

DB2 Version 9.1 for z/OS

SQL Reference



DB2 Version 9.1 for z/OS

SQL Reference



Note

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

Eleventh edition (March 2011)

This edition applies to DB2 Version 9.1 for z/OS (DB2 V9.1 for z/OS, product number 5635-DB2), DB2 9 for z/OS Value Unit Edition (product number 5697-P12), and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

© Copyright IBM Corporation 1982, 2011.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this information xvii

Who should read this information	xvii
DB2 Utilities Suite	xvii
Terminology and citations	xviii
Accessibility features for DB2 Version 9.1 for z/OS	xviii
How to send your comments	xix
How to read syntax diagrams	xx
Conventions for describing mixed data values	xxii
Industry standards	xxiii

Chapter 1. DB2 concepts 1

Structured query language.	1
Static SQL	2
Dynamic SQL	3
Deferred embedded SQL	3
Interactive SQL	3
SQL Call Level Interface and Open Database Connectivity	3
Java database connectivity and embedded SQL for Java	3
DB2 data structures	4
DB2 tables	5
DB2 indexes	5
DB2 keys	6
DB2 views	8
DB2 schemas and schema qualifiers	10
DB2 storage groups.	11
DB2 databases	12
Storage structures	14
DB2 table spaces.	14
DB2 index spaces	15
DB2 system objects	15
DB2 catalog	16
DB2 directory.	16
Active and archive logs	17
Bootstrap data set	18
Buffer pools	18
Data definition control support database	19
Resource limit facility database.	19
Work file database	19
DB2 and data integrity	19
Constraints	20
Triggers	24
Application processes, concurrency, and recovery.	25
Locking, commit, and rollback	25
Unit of work	25
Unit of recovery	26
Rolling back work	26
Packages and application plans.	27
Routines	28
Functions	28
Stored procedures	29
Sequences	30
User-defined types	30
Distributed data	31
Connections	31
Distributed unit of work	32

Remote unit of work	35
Character conversion	38
Character sets and code pages	40
Coded character sets and CCSIDS	43
Determining the encoding scheme and CCSID of a string	43
Expanding conversions	46
Contracting conversions	46

Chapter 2. Language elements 47

Characters	47
Tokens	48
Identifiers	49
SQL identifiers	49
Host identifiers	50
Restrictions for distributed access	51
Naming conventions	51
SQL path	56
Qualification of unqualified object names	57
Unqualified alias, index, JAR file, sequence, table, trigger, and view names	58
Unqualified distinct type, function, procedure, and specific names	58
Aliases and synonyms	59
Authorization, privileges, and object ownership	60
Authorization IDs, roles, and authorization names	62
Authorization IDs and schema names	63
Authorization IDs and statement preparation	63
Authorization IDs and dynamic SQL	64
Authorization IDs and remote execution	67
Data types	69
Nulls	70
Numbers	70
Character strings	73
Graphic strings	83
Binary strings	84
Large objects (LOBs)	85
Datetime values	87
Row ID values	92
XML values	93
Distinct types	94
Promotion of data types	95
Casting between data types	96
Assignment and comparison	102
Numeric assignments	105
String assignments	109
Datetime assignments	112
Row ID assignments	112
XML assignments	113
Distinct type assignments	113
Assignments to LOB locators	114
Numeric comparisons	114
String comparisons	115
Datetime comparisons	117
Row ID comparisons	117
XML comparisons	117
Distinct type comparisons	117
Rules for result data types	119
Numeric operands	119
Character and graphic string operands	120
Binary string operands	121
Datetime operands	121
Row ID operands	122
XML operands	122

Distinct type operands	122
Conversion rules for operations that combine strings	122
Constants	126
Integer constants	127
Floating-point constants	127
Decimal constants	127
Decimal floating-point constants	127
Character string constants	128
Binary string constants	129
Datetime constants	129
Graphic string constants	129
Special registers	131
General rules for special registers.	132
CURRENT APPLICATION ENCODING SCHEME	134
CURRENT CLIENT_ACCTNG	135
CURRENT CLIENT_APPLNAME	136
CURRENT CLIENT_USERID	136
CURRENT CLIENT_WRKSTNNAME	136
CURRENT DATE	137
CURRENT DEBUG MODE	137
CURRENT DECFLOAT ROUNDING MODE	138
CURRENT DEGREE	139
CURRENT LOCALE LC_CTYPE	140
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	140
CURRENT MEMBER	140
CURRENT OPTIMIZATION HINT	141
CURRENT PACKAGE PATH	141
CURRENT PACKAGESET	142
CURRENT PATH	142
CURRENT PRECISION	143
CURRENT REFRESH AGE	144
CURRENT ROUTINE VERSION	145
CURRENT RULES	145
CURRENT SCHEMA	147
CURRENT SERVER	147
CURRENT SQLID	148
CURRENT TIME	148
CURRENT TIMESTAMP	149
CURRENT TIMEZONE	149
ENCRYPTION PASSWORD	149
SESSION_USER	150
USER	150
Using special registers in a user-defined function or a stored procedure	151
Column names	153
Qualified column names.	154
Correlation names.	154
Column name qualifiers to avoid ambiguity	155
Column name qualifiers in correlated references.	157
Resolution of column name qualifiers and column names	158
References to variables	159
References to host variables	160
Host variables in dynamic SQL	162
References to LOB host variables	162
References to LOB locator variables	163
References to XML host variables.	164
References to file reference variables.	165
References to stored procedure result sets	166
References to result set locator variables	167
References to built-in session variables	168
Host structures in PL/I, C, and COBOL	171
Host-variable-arrays in PL/I, C, C++, and COBOL	172

Functions	173
Types of functions	173
Function invocation	174
Function resolution	175
Expressions	180
Without operators	182
With arithmetic operators	182
With the concatenation operator	188
Scalar-fullselect	191
Datetime operands and durations	192
Datetime arithmetic in SQL	193
Precedence of operations	197
CASE expressions	199
CAST specification	202
XMLCAST specification	210
OLAP specification	212
ROW CHANGE expression	216
Sequence reference	217
Predicates	222
Basic predicate	224
Quantified predicate	226
BETWEEN predicate	229
DISTINCT predicate	230
EXISTS predicate	232
IN predicate	234
LIKE predicate	237
NULL predicate	243
XMLEXISTS predicate	244
Search conditions	247
Options affecting SQL	248
Precompiler options for dynamic statements	250
DECFLOAT rounding mode	251
Decimal point representation	251
Apostrophes and quotation marks as string delimiters	253
Katakana characters for EBCDIC	254
Mixed data in character strings	254
Formatting of datetime strings	255
SQL standard language	255
Positioned updates of columns	256
Mappings from SQL to XML	257
Mapping SQL character sets to XML character sets	257
Mapping SQL identifiers to XML names	257
Mapping SQL data values to XML data values	257
Chapter 3. Functions	259
Aggregate functions	267
AVG	269
CORRELATION	270
COUNT	271
COUNT_BIG	272
COVARIANCE or COVARIANCE_SAMP	274
MAX	275
MIN	276
STDDEV or STDDEV_SAMP	277
SUM	278
VARIANCE or VARIANCE_SAMP	279
XMLAGG	281
Scalar functions	283
ABS	284
ACOS	285
ADD_MONTHS	286

	ASCII	288
	ASCII_CHR	289
	ASCII_STR	290
	ASIN	291
	ATAN	292
	ATANH	293
	ATAN2	294
	BIGINT	295
	BINARY	297
	BLOB	299
	CCSID_ENCODING	300
	CEILING	301
	CHAR	302
	CHARACTER_LENGTH	310
	CLOB	312
	COALESCE	315
	COLLATION_KEY	317
	COMPARE_DECFLOAT	320
	CONCAT	322
	CONTAINS	323
	COS	326
	COSH	327
	DATE	328
	DAY	330
	DAYOFMONTH	331
	DAYOFWEEK	332
	DAYOFWEEK_ISO	333
	DAYOFYEAR	334
	DAYS	335
	DBCLOB	336
	DECFLOAT	340
	DECFLOAT_SORTKEY	342
	DECIMAL or DEC	344
	DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, and DECRYPT_DB	346
	DEGREES	350
	DIFFERENCE	351
	DIGITS	352
	DOUBLE_PRECISION or DOUBLE	353
	DSN_XMLVALIDATE	355
	DSN_XMLVALIDATE	357
	EBCDIC_CHR	361
	EBCDIC_STR	362
	ENCRYPT_TDES	363
	EXP	366
	EXTRACT	367
	FLOAT	369
	FLOOR	370
	GENERATE_UNIQUE	371
	GETHINT	373
	GETVARIABLE	374
	GRAPHIC	376
	HEX	380
	HOURL	381
	IDENTITY_VAL_LOCAL	382
	IFNULL	387
	INSERT	388
	INTEGER or INT	392
	JULIAN_DAY	394
	LAST_DAY	395
	LCASE	396
	LEFT	397

	LENGTH	401
	LN	402
	LOCATE	403
	LOCATE_IN_STRING	406
	LOG10	409
	LOWER	410
	LPAD	413
	LTRIM	415
	MAX	416
	MICROSECOND	417
	MIDNIGHT_SECONDS	418
	MIN	419
	MINUTE	420
	MOD	421
	MONTH	423
	MONTHS_BETWEEN	424
	MQPUBLISH	426
	MQPUBLISHXML	429
	MQREAD	431
	MQREADCLOB	433
	MQREADXML	435
	MQRECEIVE	437
	MQRECEIVECLOB	439
	MQRECEIVEXML	441
	MQSEND	443
	MQSENDXML	445
	MQSENDXMLFILE	447
	MQSENDXMLFILECLOB	449
	MQSUBSCRIBE	451
	MQUNSUBSCRIBE	453
	MULTIPLY_ALT	455
	NEXT_DAY	456
	NORMALIZE_DECFLOAT	458
	NORMALIZE_STRING	459
	NULLIF	461
	OVERLAY	462
	POSITION	466
	POSSTR	469
	POWER	471
	QUANTIZE	472
	QUARTER	474
	RADIANS	475
	RAISE_ERROR	476
	RAND	477
	REAL	478
	REPEAT	480
	REPLACE	482
	RID	485
	RIGHT	486
	ROUND	488
	ROUND_TIMESTAMP	490
	ROWID	493
	RPAD	494
	RTRIM	496
	SCORE	498
	SECOND	501
	SIGN	502
	SIN	503
	SINH	504
	SMALLINT	505
	SOUNDEX	507

	SOAPHTTPC and SOAPHTTPV	508
	SOAPHTTPNC and SOAPHTTPNV	509
	SPACE	511
	SQRT	512
	STRIP	513
	SUBSTR	515
	SUBSTRING	517
	TAN	522
	TANH	523
	TIME	524
	TIMESTAMP	525
	TIMESTAMPADD	527
	TIMESTAMP_FORMAT	529
	TIMESTAMP_ISO	534
	TIMESTAMPDIFF	535
	TO_CHAR	538
	TO_DATE	539
	TOTALORDER	540
	TRANSLATE	541
	TRUNCATE or TRUNC	544
	TRUNC_TIMESTAMP	546
	UCASE	549
	UNICODE	550
	UNICODE_STR	551
	UPPER	552
	VALUE	555
	VARBINARY	556
	VARCHAR	558
	VARCHAR_FORMAT	564
	VARGRAPHIC	569
	WEEK	573
	WEEK_ISO	574
	XMLATTRIBUTES	575
	XMLCOMMENT	576
	XMLCONCAT	577
	XMLDOCUMENT	578
	XMLELEMENT	579
	XMLFOREST	584
	XMLNAMESPACES	587
	XMLPARSE	589
	XMLPI	591
	XMLQUERY	592
	XMLSERIALIZE	596
	XMLTEXT	599
	YEAR	600
	Table functions	600
	ADMIN_TASK_LIST	601
	ADMIN_TASK_STATUS	606
	MQREADALL	609
	MQREADALLCLOB	611
	MQREADALLXML	613
	MQRECEIVEALL	615
	MQRECEIVEALLCLOB	618
	MQRECEIVEALLXML	621
	XMLTABLE	624
	Chapter 4. Queries	629
	Authorization	630
	subselect	632
	select-clause	633
	from-clause	638

where-clause	648
group-by-clause	649
having-clause	651
order-by-clause	652
fetch-first-clause	655
Examples of subselects	657
fullselect	662
Character conversion in set operations and concatenations	666
Selecting the result CCSID	667
select-statement	669
common-table-expression	670
update-clause	673
read-only-clause	674
optimize-clause	675
isolation-clause	676
queryno-clause	678
SKIP LOCKED DATA	679
Examples of select statements	680

Chapter 5. Statements 683

How SQL statements are invoked	688
Embedding a statement in an application program	689
Dynamic preparation and execution	690
Static invocation of a SELECT statement	691
Dynamic invocation of a SELECT statement	692
Interactive invocation	692
SQL diagnostics information	692
Detecting and processing error and warning conditions in host language applications	693
SQL comments	695
ALLOCATE CURSOR	696
ALTER DATABASE	698
ALTER FUNCTION (external)	701
ALTER FUNCTION (SQL scalar)	719
ALTER INDEX	725
ALTER PROCEDURE (external)	741
ALTER PROCEDURE (SQL - external)	752
ALTER PROCEDURE (SQL - native)	758
ALTER SEQUENCE	781
ALTER STOGROUP	786
ALTER TABLE	789
ALTER TABLESPACE	841
ALTER TRUSTED CONTEXT	855
ALTER VIEW	867
ASSOCIATE LOCATORS	868
BEGIN DECLARE SECTION	872
CALL	874
CLOSE	885
COMMENT	887
COMMIT	896
CONNECT	899
CREATE ALIAS	905
CREATE AUXILIARY TABLE	908
CREATE DATABASE	912
CREATE FUNCTION	915
CREATE FUNCTION (external scalar)	916
CREATE FUNCTION (external table)	940
CREATE FUNCTION (sourced)	958
CREATE FUNCTION (SQL scalar)	972
CREATE GLOBAL TEMPORARY TABLE	982
CREATE INDEX	988
CREATE PROCEDURE	1015

	CREATE PROCEDURE (external)	1016
	CREATE PROCEDURE (SQL - external)	1035
	CREATE PROCEDURE (SQL - native)	1046
	CREATE ROLE	1065
	CREATE SEQUENCE	1066
	CREATE STOGROUP	1074
	CREATE SYNONYM	1077
	CREATE TABLE	1079
	CREATE TABLESPACE	1128
	CREATE TRIGGER	1151
	CREATE TRUSTED CONTEXT	1167
	CREATE TYPE	1177
	CREATE VIEW	1184
	DECLARE CURSOR	1191
	DECLARE GLOBAL TEMPORARY TABLE	1202
	DECLARE STATEMENT	1216
	DECLARE TABLE	1217
	DECLARE VARIABLE	1221
	DELETE	1224
	DESCRIBE	1237
	DESCRIBE CURSOR	1238
	DESCRIBE INPUT	1240
	DESCRIBE OUTPUT	1243
	DESCRIBE PROCEDURE	1250
	DESCRIBE TABLE	1253
	DROP	1256
	END DECLARE SECTION	1273
	EXCHANGE	1274
	EXECUTE	1275
	EXECUTE IMMEDIATE	1280
	EXPLAIN	1283
	FETCH	1290
	FREE LOCATOR	1316
	GET DIAGNOSTICS	1317
	GRANT	1333
	GRANT (collection privileges)	1336
	GRANT (database privileges)	1337
	GRANT (function or procedure privileges)	1340
	GRANT (package privileges)	1345
	GRANT (plan privileges)	1348
	GRANT (schema privileges)	1349
	GRANT (sequence privileges)	1351
	GRANT (system privileges)	1352
	GRANT (table or view privileges)	1355
	GRANT (type or JAR file privileges)	1359
	GRANT (use privileges)	1361
	HOLD LOCATOR	1363
	INCLUDE	1365
	INSERT	1367
	LABEL	1384
	LOCK TABLE	1386
	MERGE	1388
	OPEN	1400
	PREPARE	1405
	REFRESH TABLE	1422
	RELEASE (connection)	1424
	RELEASE SAVEPOINT	1427
	RENAME	1428
	REVOKE	1432
	REVOKE (collection privileges)	1437
	REVOKE (database privileges)	1439

REVOKE (function or procedure privileges)	1442
REVOKE (package privileges)	1447
REVOKE (plan privileges).	1449
REVOKE (schema privileges).	1451
REVOKE (sequence privileges)	1453
REVOKE (system privileges)	1455
REVOKE (table or view privileges).	1458
REVOKE (type or JAR file privileges)	1461
REVOKE (use privileges)	1463
ROLLBACK	1465
SAVEPOINT	1468
SELECT	1470
SELECT INTO.	1471
SET CONNECTION.	1475
SET CURRENT APPLICATION ENCODING SCHEME	1477
SET CURRENT DEBUG MODE	1478
SET CURRENT DECFLOAT ROUNDING MODE	1480
SET CURRENT DEGREE	1482
SET CURRENT LOCALE LC_CTYPE	1483
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	1485
SET CURRENT OPTIMIZATION HINT	1487
SET CURRENT PACKAGE PATH	1488
SET CURRENT PACKAGESET	1492
SET CURRENT PRECISION	1494
SET CURRENT REFRESH AGE	1495
SET CURRENT ROUTINE VERSION	1497
SET CURRENT RULES.	1499
SET CURRENT SQLID	1500
SET ENCRYPTION PASSWORD	1502
SET host-variable assignment.	1504
SET PATH	1507
SET SCHEMA.	1510
SET transition-variable assignment	1513
SIGNAL.	1516
TRUNCATE	1517
UPDATE	1521
VALUES.	1536
VALUES INTO	1537
WHENEVER	1539
 Chapter 6. SQL control statements for native SQL procedures.	 1541
References to SQL parameters and SQL variables	1542
References to SQL condition names.	1543
References to SQL cursor names.	1543
References to labels	1543
Nested compound statements and scope of names	1544
SQL-procedure-statement	1546
assignment-statement	1548
CALL statement	1550
CASE statement	1552
compound-statement	1554
FOR statement	1563
GET DIAGNOSTICS statement	1565
GOTO statement	1566
IF statement	1568
ITERATE statement	1569
LEAVE statement.	1571
LOOP statement	1573
REPEAT statement	1575
RESIGNAL statement	1577
RETURN statement	1580

SIGNAL statement	1582
WHILE statement	1586

Appendix. Additional information for DB2 SQL 1587

Limits in DB2 for z/OS	1588
Reserved schema names and reserved words	1594
Reserved schema names	1595
Reserved words	1596
Characteristics of SQL statements in DB2 for z/OS	1600
Actions allowed on SQL statements	1601
SQL statements allowed in external functions and stored procedures	1605
SQL statements allowed in SQL procedures	1608
SQL control statements for external SQL procedures	1610
References to SQL parameters and SQL variables	1611
SQL-procedure-statement	1612
assignment-statement (SQL control statements for external routines).	1613
CALL statement	1615
CASE statement	1617
compound-statement	1620
GET DIAGNOSTICS statement	1626
GOTO statement	1627
IF statement	1629
ITERATE statement	1631
LEAVE statement.	1632
LOOP statement	1633
REPEAT statement	1635
RESIGNAL statement	1636
RETURN statement	1639
SIGNAL statement	1641
WHILE statement	1645
SQL communication area (SQLCA).	1646
Description of SQLCA fields	1647
The included SQLCA	1652
The REXX SQLCA	1654
SQL descriptor area (SQLDA)	1656
Description of SQLDA fields	1658
Unrecognized and unsupported SQLTYPES	1669
The included SQLDA	1670
Identifying an SQLDA in C or C++.	1674
The REXX SQLDA	1675
DB2 catalog tables	1677
Table spaces and indexes	1679
New and changed catalog tables	1692
SYSIBM.IPLIST table	1697
SYSIBM.IPNAMES table	1698
SYSIBM.LOCATIONS table	1701
SYSIBM.LULIST table	1703
SYSIBM.LUMODES table	1704
SYSIBM.LUNAMES table	1705
SYSIBM.MODESELECT table.	1708
SYSIBM.SYSAUXRELS table	1709
SYSIBM.SYSCHECKDEP table	1710
SYSIBM.SYSCHECKS table	1711
SYSIBM.SYSCHECKS2 table	1712
SYSIBM.SYSCOLAUTH table.	1713
SYSIBM.SYSCOLDIST table	1715
SYSIBM.SYSCOLDISTSTATS table	1717
SYSIBM.SYSCOLDIST_HIST table	1719
SYSIBM.SYSCOLSTATS table.	1721
SYSIBM.SYSCOLUMNS table.	1723
SYSIBM.SYSCOLUMNS_HIST table	1732

	SYSIBM.SYSCONSTDEP table	1736
	SYSIBM.SYSCONTEXT table	1737
	SYSIBM.SYSCONTEXTAUTHIDS table	1739
	SYSIBM.SYSCOPY table	1740
	SYSIBM.SYSCTXTTRUSTATTRS table	1746
	SYSIBM.SYSDATABASE table	1747
	SYSIBM.SYSDATATYPES table	1749
	SYSIBM.SYSDBAUTH table	1751
	SYSIBM.SYSDBRM table	1754
	SYSIBM.SYSDEPENDENCIES table.	1756
	SYSIBM.SYSDUMMY1 table	1758
	SYSIBM.SYSENVIRONMENT table.	1759
	SYSIBM.SYSFIELDS table	1761
	SYSIBM.SYSFOREIGNKEYS table	1763
	SYSIBM.SYSINDEXES table	1764
	SYSIBM.SYSINDEXES_HIST table	1769
	SYSIBM.SYSINDEXPART table	1771
	SYSIBM.SYSINDEXPART_HIST table	1775
	SYSIBM.SYSINDEXSPACESTATS table	1777
	SYSIBM.SYSINDEXSTATS table	1782
	SYSIBM.SYSINDEXSTATS_HIST table.	1784
	SYSIBM.SYSJARCLASS_SOURCE table	1785
	SYSIBM.SYSJARCONTENTS table	1786
	SYSIBM.SYSJARDATA table	1787
	SYSIBM.SYSJAROBJECTS table	1788
	SYSIBM.SYSJAVAOPTS table	1789
	SYSIBM.SYSJAVAPATHS table	1790
	SYSIBM.SYSKEYCOLUSE table	1791
	SYSIBM.SYSKEYS table	1792
	SYSIBM.SYSKEYTARGETS table.	1793
	SYSIBM.SYSKEYTARGETSTATS table	1796
	SYSIBM.SYSKEYTARGETS_HIST table	1798
	SYSIBM.SYSKEYTGTDIST table	1801
	SYSIBM.SYSKEYTGTDISTSTATS table.	1803
	SYSIBM.SYSKEYTGTDIST_HIST table.	1805
	SYSIBM.SYSLOBSTATS table	1807
	SYSIBM.SYSLOBSTATS_HIST table.	1808
	SYSIBM.SYSOBJROLEDEP table.	1809
	SYSIBM.SYSPACKAGE table	1810
	SYSIBM.SYSPACKAUTH table	1818
	SYSIBM.SYSPACKDEP table	1820
	SYSIBM.SYSPACKLIST table	1822
	SYSIBM.SYSPACKSTMT table	1823
	SYSIBM.SYSPARMS table	1826
	SYSIBM.SYSPKSYSTEM table	1830
	SYSIBM.SYSPLAN table	1832
	SYSIBM.SYSPLANAUTH table	1837
	SYSIBM.SYSPLANDEP table	1839
	SYSIBM.SYSPLSYSTEM table.	1840
	SYSIBM.SYSRELS table.	1841
	SYSIBM.SYSRESAUTH table	1843
	SYSIBM.SYSROLES table	1845
	SYSIBM.SYSROUTINEAUTH table.	1846
	SYSIBM.SYSROUTINES table.	1848
	SYSIBM.SYSROUTINESTEXT table.	1858
	SYSIBM.SYSROUTINES_OPTS table	1859
	SYSIBM.SYSROUTINES_SRC table	1861
	SYSIBM.SYSSCHEMAAUTH table	1862
	SYSIBM.SYSSEQUENCEAUTH table	1863
	SYSIBM.SYSSEQUENCES table	1865
	SYSIBM.SYSSEQUENCESDEP table	1867

	SYSIBM.SYSSTMT table	1868
	SYSIBM.SYSSTOGROUP table	1872
	SYSIBM.SYSSTRINGS table	1874
	SYSIBM.SYSSYNONYMS table	1877
	SYSIBM.SYSTABAUTH table	1878
	SYSIBM.SYSTABCONST table	1881
	SYSIBM.SYSTABLEPART table	1882
	SYSIBM.SYSTABLEPART_HIST table	1887
	SYSIBM.SYSTABLES table	1890
	SYSIBM.SYSTABLESPACE table	1896
	SYSIBM.SYSTABLESPACESTATS table	1901
	SYSIBM.SYSTABLES_HIST table	1905
	SYSIBM.SYSTABSTATS table	1907
	SYSIBM.SYSTABSTATS_HIST table	1908
	SYSIBM.SYSTRIGGERS table	1909
	SYSIBM.SYSUSERAUTH table	1911
	SYSIBM.SYSVIEWDEP table	1914
	SYSIBM.SYSVIEWS table	1915
	SYSIBM.SYSVOLUMES table	1917
	SYSIBM.SYSXMLRELS table	1918
	SYSIBM.SYSXMLSTRINGS table	1919
	SYSIBM.USERNAMES table	1920
	SYSIBM.XSRCOMPONENT table	1921
	SYSIBM.XSROBJECTS table	1922
	SYSIBM.XSROBJECTCOMPONENTS table	1924
	SYSIBM.XSROBJECTGRAMMAR table	1925
	SYSIBM.XSROBJECTHIERARCHIES table	1926
	SYSIBM.XSROBJECTPROPERTY table	1927
	SYSIBM.XSRPROPERTY table	1928
	EXPLAIN tables	1929
	PLAN_TABLE	1930
	DSN_DETCOST_TABLE	1943
	DSN_FILTER_TABLE	1949
	DSN_FUNCTION_TABLE	1952
	DSN_PGRANGE_TABLE	1955
	DSN_PGROUP_TABLE	1957
	DSN_PREDICAT_TABLE	1961
	DSN_PTASK_TABLE	1966
	DSN_QUERYINFO_TABLE	1969
	DSN_QUERY_TABLE	1973
	DSN_SORTKEY_TABLE	1975
	DSN_SORT_TABLE	1978
	DSN_STATEMENT_CACHE_TABLE	1981
	DSN_STATEMNT_TABLE	1985
	DSN_STRUCT_TABLE	1989
	DSN_VIEWREF_TABLE	1992
	Tables that are used by accelerators	1995
	Table spaces and indexes for accelerators	1996
	SYSACCEL.SYSACCELERATORS table	1997
	SYSACCEL.SYSACCELIPLIST table	1998
	Using the catalog in database design	1998
	Retrieving catalog information about DB2 storage groups	1998
	Retrieving catalog information about a table	1999
	Retrieving catalog information about partition order	1999
	Retrieving catalog information about aliases	2000
	Retrieving catalog information about columns	2000
	Retrieving catalog information about indexes	2001
	Retrieving catalog information about views	2001
	Retrieving catalog information about authorizations	2002
	Retrieving catalog information about primary keys	2002
	Retrieving catalog information about foreign keys	2003

Retrieving catalog information about check pending	2003
Retrieving catalog information about check constraints	2004
Retrieving catalog information about LOBs	2004
Retrieving catalog information about user-defined functions and stored procedures	2005
Retrieving catalog information about triggers	2005
Retrieving catalog information about sequences	2006
Adding and retrieving comments	2006
Verifying the accuracy of the database definition	2007
Sample user-defined functions	2007
ALTDATE	2009
ALTTIME	2012
CURRENCY	2014
DAYNAME	2016
MONTHNAME	2017
TABLE_LOCATION	2018
TABLE_NAME	2020
TABLE_SCHEMA	2022
WEATHER	2024
Information resources for DB2 for z/OS and related products	2027
How to obtain DB2 information	2033
How to use the DB2 library	2035
Notices	2039
Programming Interface Information	2041
General-use Programming Interface and Associated Guidance Information	2042
Product-sensitive Programming Interface and Associated Guidance Information.	2043
Trademarks	2043
Glossary	2045
Index	2091

About this information

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

This information assumes that your DB2 subsystem is running in Version 9.1 new-function mode. Generally, new functions that are described, including changes to existing functions, statements, and limits, are available only in new-function mode. Two exceptions to this general statement are new and changed utilities and optimization enhancements, which are also available in conversion mode unless stated otherwise.

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

Who should read this information

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

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

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

DB2 Utilities Suite

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

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

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

These informational APARs are periodically updated.

Related information

DB2 utilities packaging (Utility Guide)

Terminology and citations

In this information, DB2 Version 9.1 for z/OS is referred to as "DB2 for z/OS." In cases where the context makes the meaning clear, DB2 for z/OS is referred to as "DB2." When this information refers to titles of DB2 for z/OS books, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to *IBM DB2 Version 9.1 for z/OS SQL Reference*.)

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

The following terms are used as indicated:

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

OMEGAMON®

Refers to any of the following products:

- IBM Tivoli® OMEGAMON XE for DB2 Performance Expert on z/OS
- IBM Tivoli OMEGAMON XE for DB2 Performance Monitor on z/OS
- IBM DB2 Performance Expert for Multiplatforms and Workgroups
- IBM DB2 Buffer Pool Analyzer for z/OS

C, C++, and C language

Represent the C or C++ programming language.

CICS® Represents CICS Transaction Server for z/OS.

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

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

RACF®

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

Accessibility features for DB2 Version 9.1 for z/OS

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

Accessibility features

The following list includes the major accessibility features in z/OS products, including DB2 Version 9.1 for z/OS. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size

Tip: The Information Management Software for z/OS Solutions Information Center (which includes information for DB2 Version 9.1 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

You can access DB2 Version 9.1 for z/OS ISPF panel functions by using a keyboard or keyboard shortcut keys.

For information about navigating the DB2 Version 9.1 for z/OS ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

Online documentation for DB2 Version 9.1 for z/OS is available in the Information Management Software for z/OS Solutions Information Center, which is available at the following website: <http://publib.boulder.ibm.com/infocenter/imzic>

IBM and accessibility

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

How to send your comments

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

- Send your comments by email to db2zinfo@us.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title or a help topic title).
- You can send comments from the web. Visit the DB2 for z/OS - Technical Resources website at:

<http://www.ibm.com/support/docview.wss?rs=64&uid=swg27011656>

This website has an online reader comment form that you can use to send comments.

- You can also send comments by using the **Feedback** link at the footer of each page in the Information Management Software for z/OS Solutions Information Center at <http://publib.boulder.ibm.com/infocenter/imzic>.

How to read syntax diagrams

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

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

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

The ►— symbol indicates the beginning of a statement.

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

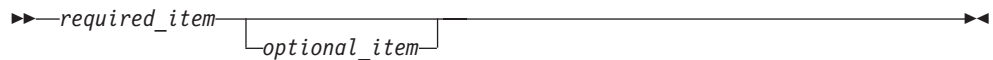
The ►— symbol indicates that a statement is continued from the previous line.

The —► symbol indicates the end of a statement.

- Required items appear on the horizontal line (the main path).



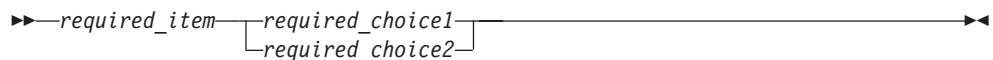
- Optional items appear below the main path.



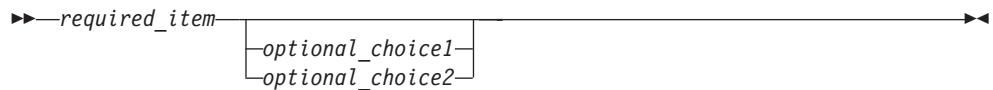
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



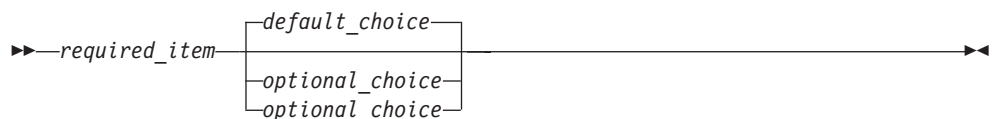
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



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



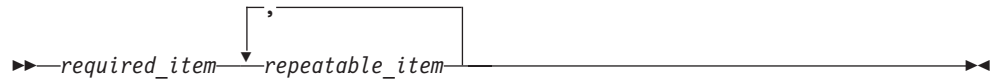
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



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



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

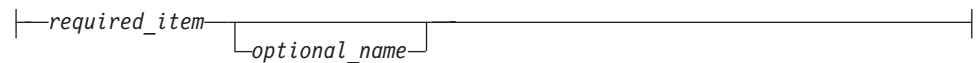


A repeat arrow above a stack indicates that you can repeat the items in the stack.

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



fragment-name:



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

Conventions for describing mixed data values

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

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

Convention	Representation
S _O	"shift-out" control character (X'0E'), used only for EBCDIC data
S _I	"shift-in" control character (X'0F'), used only for EBCDIC data
sbc-string	SBCS string of zero or more single-byte characters
dbc-string	DBCS string of zero or more double-byte characters
'	DBCS apostrophe
G	DBCS uppercase G

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

Industry standards

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

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

Chapter 1. DB2 concepts

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

The following topics provide information on these concepts:

- “Structured query language”
- “DB2 schemas and schema qualifiers” on page 10
- “DB2 tables” on page 5
- “DB2 indexes” on page 5
- “DB2 keys” on page 6
- “Constraints” on page 20
- “Triggers” on page 24
- “Storage structures” on page 14
- “DB2 storage groups” on page 11
- “DB2 databases” on page 12
- “DB2 catalog” on page 16
- “DB2 views” on page 8
- “Sequences” on page 30
- “Routines” on page 28
- “Application processes, concurrency, and recovery” on page 25
- “Packages and application plans” on page 27
- “Distributed data” on page 31
- “Character conversion” on page 38

Structured query language

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

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

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

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

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

In accordance with the relational model of data:

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

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

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

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

As the following example illustrates, SQL is generally intuitive. 

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

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

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

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



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

Static SQL

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

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

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

Dynamic SQL

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

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

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

Deferred embedded SQL

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

Like a static statement, it is embedded within an application, but like a dynamic statement, it is prepared during the execution of the application. Although prepared at run time, a deferred embedded SQL statement is processed with bind-time rules such that the authorization ID and qualifier determined at bind time for the plan or package owner are used. Deferred embedded SQL statements are used for DB2 private protocol access to remote data.

Interactive SQL

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

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

SQL Call Level Interface and Open Database Connectivity

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

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

Java database connectivity and embedded SQL for Java

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

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

DB2 data structures

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

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

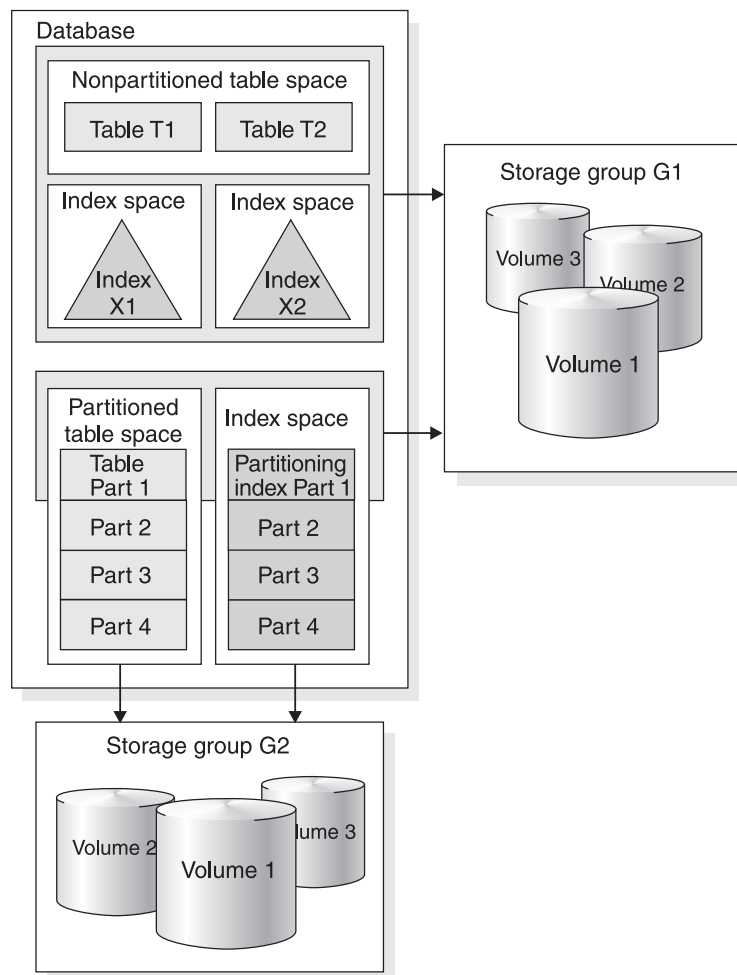


Figure 2. A hierarchy of DB2 structures

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

Databases

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

Storage groups

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

Table spaces

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

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

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

Indexes

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

DB2 tables

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

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

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

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

DB2 supports several different types of tables:

- Auxiliary tables
- Base tables
- Clone tables
- Empty tables
- Materialized query tables
- Result tables
- Temporary tables
- XML tables

DB2 indexes

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

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

DB2 can use indexes to ensure uniqueness and to improve performance by clustering data, partitioning data, and providing efficient access paths to data for

queries. In most cases, access to data is faster with an index than with a scan of the data. For example, you can create an index on the DEPTNO column of the sample DEPT table to easily locate a specific department and avoid reading through each row of, or *scanning*, the table.

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

The main purposes of indexes are:

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

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

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

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

DB2 keys

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

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

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

to the value of the primary key.” This rule means that each component of the value of the foreign key must be equal to the corresponding component of the value of the primary key.

DB2 supports several types of keys.

Unique keys

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

The columns of a unique key cannot contain null values.

Primary keys

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

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

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

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

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

Parent keys

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

Foreign keys

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

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

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

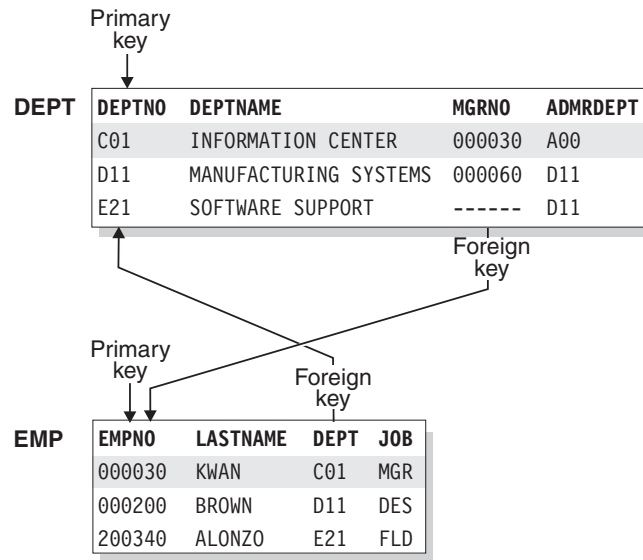


Figure 3. Relationship between DEPT and EMP tables

Figure notes: Each table has a primary key:

- DEPTNO in the DEPT table
- EMPNO in the EMP table

Each table has a foreign key that establishes a relationship between the tables:

- The values of the foreign key on the DEPT column of the EMP table match values in the DEPTNO column of the DEPT table.
- The values of the foreign key on the MGRNO column of the DEPT table match values in the EMPNO column of the EMP table when an employee is a manager.

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

DB2 views

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


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

You can create a view that:

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

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

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

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

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

Base table, **EMP**:

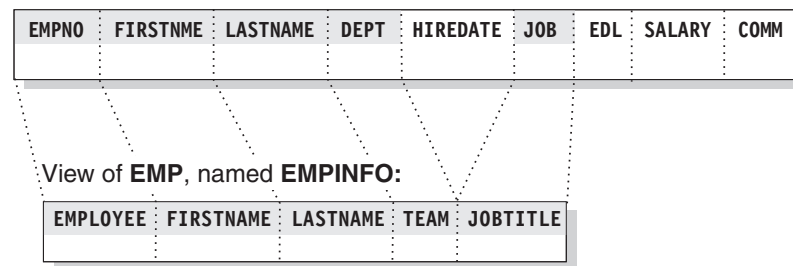


Figure 4. A view of the EMP table

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

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

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

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

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

```
CREATE VIEW EMPINFO (EMPLOYEE, FIRSTNAME, LASTNAME, TEAM, JOBTITLE)
  AS SELECT EMPNO, FIRSTNAME, LASTNAME, DEPT, JOB
  WHERE DEPT = 'A00' OR DEPT = 'C01'
  FROM EMP;
```

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

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

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

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

DB2 schemas and schema qualifiers

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

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

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

When a table, index, table space, distinct type, function, stored procedure, or trigger is created, it is given a qualified two-part name. The first part is the schema name (or the qualifier), which is either implicitly or explicitly specified. The default schema is the authorization ID of the owner of the plan or package. The second part is the name of the object.

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

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

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

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

Table 1. Schema qualifier and owner for objects

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

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

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

Table 2. Schema qualifier and owner for additional objects

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

DB2 storage groups

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

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

Within the storage group, DB2 does the following actions:

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

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

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

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

You have several options for managing DB2 data sets:

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

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

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

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

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

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

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

DB2 databases

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

Whenever a table space is created, it is explicitly or implicitly assigned to an existing database. If you create a table space and do not specify a database name, the table space is created in the default database, DSNDB04. In this case, DB2

implicitly creates a database or uses an existing implicitly created database for the table. All users who have the authority to create table spaces or tables in database DSND04 have authority to create tables and table spaces in an implicitly created database. If the table space is implicitly created, and you do not specify the IN clause in the CREATE TABLE statement, DB2 implicitly creates the database to which the table space is assigned.

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

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

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

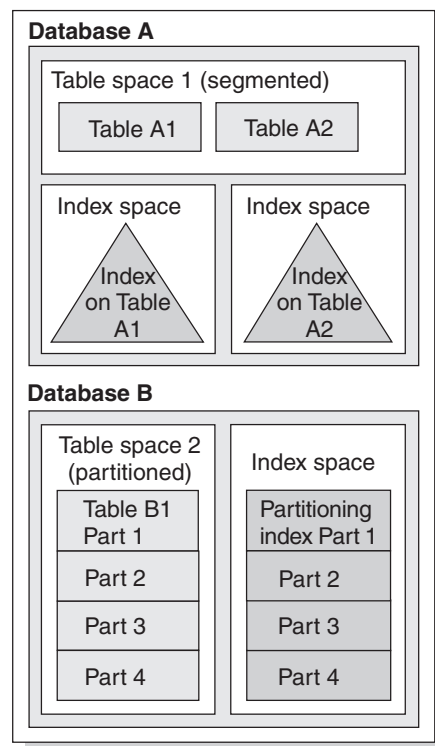


Figure 5. Data structures in a DB2 database

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

Declared global temporary tables are now stored in the WORKFILE database. The TEMP database is no longer used.

Reasons to define a database

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

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

When one user is executing a CREATE, ALTER, or DROP statement for a table, no other user can access the database that contains that table. QMF™ users, especially, might do a great deal of data definition; the QMF operations SAVE DATA and ERASE *data-object* are accomplished by creating and dropping DB2 tables. For maximum concurrency, create a separate database for each QMF user.

- The internal database descriptors (DBDs) might become inconveniently large. DBDs grow as new objects are defined, but they do not immediately shrink when objects are dropped; the DBD space for a dropped object is not reclaimed until the MODIFY RECOVERY utility is used to delete records of obsolete copies from SYSIBM.SYSCOPY. DBDs occupy storage and are the objects of occasional input and output operations. Therefore, limiting the size of DBDs is another reason to define new databases.

Storage structures

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

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

DB2 table spaces

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

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

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

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

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

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

Universal table spaces

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

Partitioned table spaces

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

Segmented table spaces

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

Large object table spaces

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

Simple table spaces

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

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

XML table spaces

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

DB2 index spaces

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

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

DB2 system objects

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

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

DB2 catalog

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

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

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

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

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

Table information

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

Column information

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

Authorization information

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

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

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

DB2 directory

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

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

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

Table 3. Directory table spaces

Table space name	Description
SCT02	Contains the internal form of SQL statements that are contained in an application. When you bind a plan, DB2 creates a structure in SCT02.
SPT01 Skeleton package	Contains the internal form of SQL statements that are contained in a package.
SYSSPUXA	Contains the contents of a package selection.
SYSSPUXB	Contains the contents of a package explain block.
SYSLGRNX Log range	Tracks the opening and closing of table spaces, indexes, or partitions. By tracking this information and associating it with relative byte addresses (RBAs) as contained in the DB2 log, DB2 can reduce recovery time by reducing the amount of log that must be scanned for a particular table space, index, or partition.
SYSUTILX System utilities	Contains a row for every utility job that is running. The row persists until the utility is finished. If the utility terminates without completing, DB2 uses the information in the row when you restart the utility.
DBD01 Database descriptor (DBD)	<p>Contains internal information, called <i>database descriptors</i> (DBDs), about the databases that exist within the DB2 subsystem.</p> <p>Each database has exactly one corresponding DBD that describes the database, table spaces, tables, table check constraints, indexes, and referential relationships. A DBD also contains other information about accessing tables in the database. DB2 creates and updates DBDs whenever their corresponding databases are created or updated.</p>

Active and archive logs

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

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

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

You can choose either single logging or dual logging.

- A single active log contains up to 93 active log data sets.

- With dual logging, the active log has twice the capacity for active log data sets, because two identical copies of the log records are kept.

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

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

Bootstrap data set

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

Specifically, the BSDS contains:

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

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

The BSDS can be duplexed to ensure availability.

Buffer pools

Buffer pools are areas of virtual storage in which DB2 temporarily stores pages of table spaces or indexes. Access to data in this temporary storage is faster than accessing data on a disk.

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

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

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

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

Data definition control support database

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

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

Resource limit facility database

The *resource limit facility database* (DSNRLST) is a facility that lets you control the amount of processor resources that are used by dynamic SELECT statements.

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

You can establish a single limit for all users, different limits for individual users, or both. You can choose to have these limits applied before the statement is executed (this is called *predictive governing*), or while a statement is running (sometimes called *reactive governing*). You can even use both modes of governing. You define these limits in one or more resource limit specification tables (RLST). **GUIP**

Work file database

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

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

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

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

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

DB2 and data integrity

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

Constraints

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

Constraints fall into the following three types:

- Unique constraints
- Referential constraints
- Check constraints

Unique constraints

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

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

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

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

Referential constraints

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

Referential integrity is the state in which all values of all foreign keys are valid. Referential integrity is based on *entity integrity*. Entity integrity requires that each entity have a unique key. For example, if every row in a table represents relationships for a unique entity, the table should have one column or a set of columns that provides a unique identifier for the rows of the table. This column (or set of columns) is called the parent key of the table. To ensure that the parent key does not contain duplicate values, a unique index must be defined on the column or columns that constitute the parent key. Defining the parent key is called entity integrity.

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

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

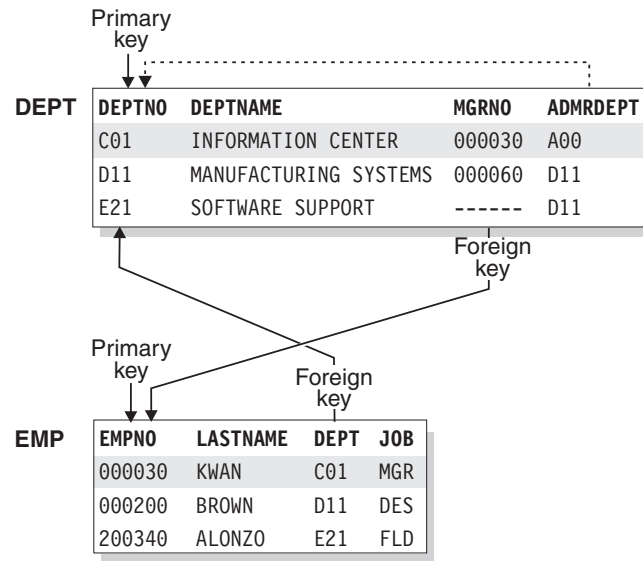


Figure 6. Referential integrity of DEPT and EMP tables

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

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

You can define a primary key on one or more columns. A primary key that includes two or more columns is called a *composite key*. A foreign key can also include one or more columns. When a foreign key contains multiple columns, the corresponding primary key must be a composite key. The number of foreign key columns must be the same as the number of columns in the parent key, and the data types of the corresponding columns must be compatible. (The sample project activity table, DSN8910.PROJACT, is an example of a table with a primary key on multiple columns, PROJNO, ACTNO, and ACSTDATE.)

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

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

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

DB2 enforces referential constraints when the following actions occur:

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

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

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

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

parent key

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

parent table

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

dependent table

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

descendent table

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

referential cycle

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

parent row

A row that has at least one dependent row.

dependent row

A row that has at least one parent row.

descendent row

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

self-referencing row

A row that is a parent of itself.

self-referencing table

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

The following rules provide referential integrity:

insert rule

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

update rule

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

delete rule

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

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

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

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

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

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

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

Check constraints

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

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

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

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

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

Triggers

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

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

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

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

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

You can define triggers with the CREATE TRIGGER statement.

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

Triggers move the business rule application logic into the database, which results in faster application development and easier maintenance. The business rule is no longer repeated in several applications, and the rule is centralized to the trigger.

DB2 checks the validity of the changes that any application makes to the salary column, and you are not required to change application programs when the logic changes.

Application processes, concurrency, and recovery

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

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

Locking, commit, and rollback

More than one application process might request access to the same data at the same time. Furthermore, under certain circumstances, an SQL statement can execute concurrently with a utility on the same table space. *Locking* is used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously.

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

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

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

Unit of work

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

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

A unit of work is initiated when an application process is initiated. A unit of work is also initiated when the previous unit of work is ended by something other than the end of the application process. A unit of work is ended by a commit operation, a full rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes made within the unit of work it ends. While these changes remain uncommitted, other application processes are unable to perceive them unless they are running with an isolation level of uncommitted

read. The changes can still be backed out. Once committed, these database changes are accessible by other application processes and can no longer be backed out by a rollback. Locks acquired by DB2 on behalf of an application process that protects uncommitted data are held at least until the end of a unit of work.

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

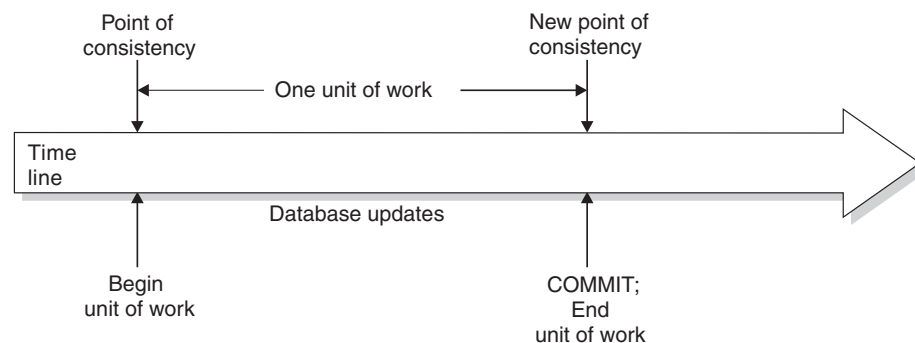


Figure 7. Unit of work with a commit operation

Unit of recovery

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

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

Rolling back work

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

Rolling back all changes

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

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

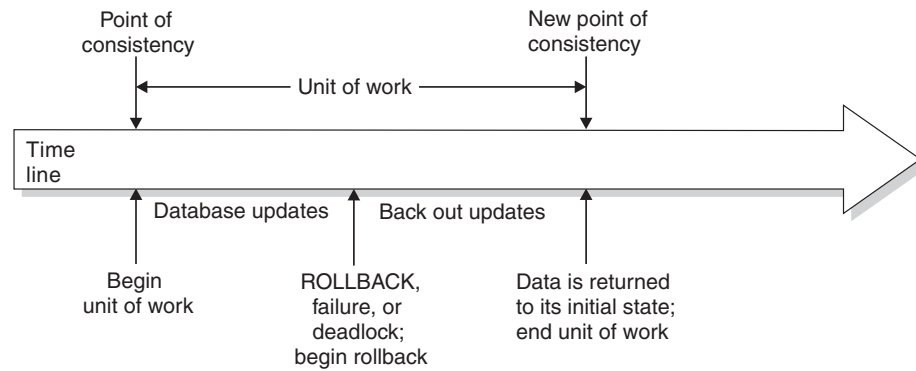


Figure 8. Rolling back all changes from a unit of work

Rolling back selected changes using savepoints

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

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

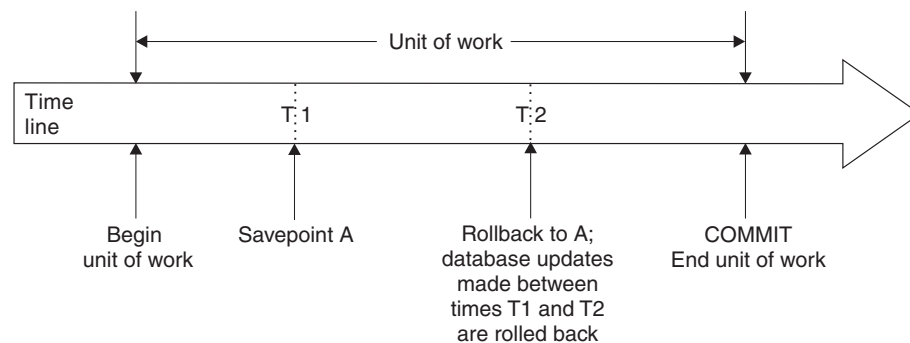


Figure 9. Rolling back changes to a savepoint within a unit of work

Packages and application plans

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

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

An application plan contains one or both of the following elements:

- A list of package names
- The bound form of SQL statements

Most DB2 applications require an application plan. Packages make application programs more flexible and easier to maintain. For example, when you use packages, you do not need to bind the entire plan again when you change one SQL statement.

Example: The following figure shows an application plan that contains two packages. Suppose that you decide to change the SELECT statement in package AA to select data from a different table. In this case, you need to bind only

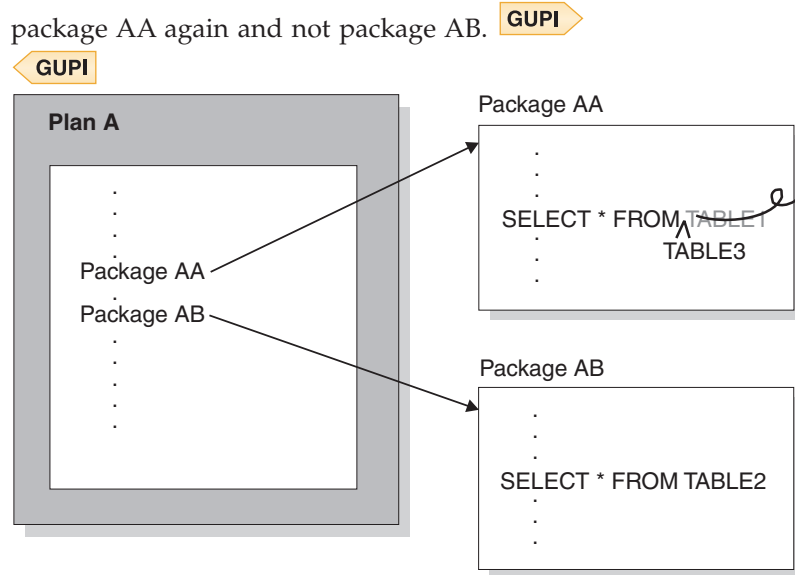


Figure 10. Application plan and packages

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

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

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

Routines

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

Functions

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

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

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

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

Stored procedures

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

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

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

External SQL procedures

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

Native SQL procedures

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

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

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

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

You define procedures by using the CREATE PROCEDURE statement.

Sequences

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

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

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

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

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

A sequence is created with a CREATE SEQUENCE statement. A sequence can be referenced using a *sequence-reference*. A sequence reference can appear most places that an expression can appear. A sequence reference can specify whether the value to be returned is a newly generated value, or the previously generated value.

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

User-defined types

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

A *distinct type* is a user-defined type that shares its internal representation with a built-in data type (its source type), but is considered to be a separate and

incompatible data type for most operations. A distinct type is created with an SQL CREATE TYPE statement. A distinct type can be used to define a column of a table, or a parameter of a routine.

For more information, see “CREATE TYPE” on page 1177 and “Distinct types” on page 94.

Distributed data

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

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

Distributed relational databases are built on formal requester-server protocols and functions. An *application requester* component supports the application end of a connection. It transforms an application's database request into communication protocols that are suitable for use in the distributed database network. These requests are received and processed by an *application server* component at the database server end of the connection. Working together, the application requester and application server handle the communication and location considerations so that the application is isolated from these considerations and can operate as if it were accessing a local database.

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

Connections

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

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

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

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

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

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

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

Distributed unit of work

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

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

Connection management

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

At any time:

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

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

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

The following figure shows the state transitions:

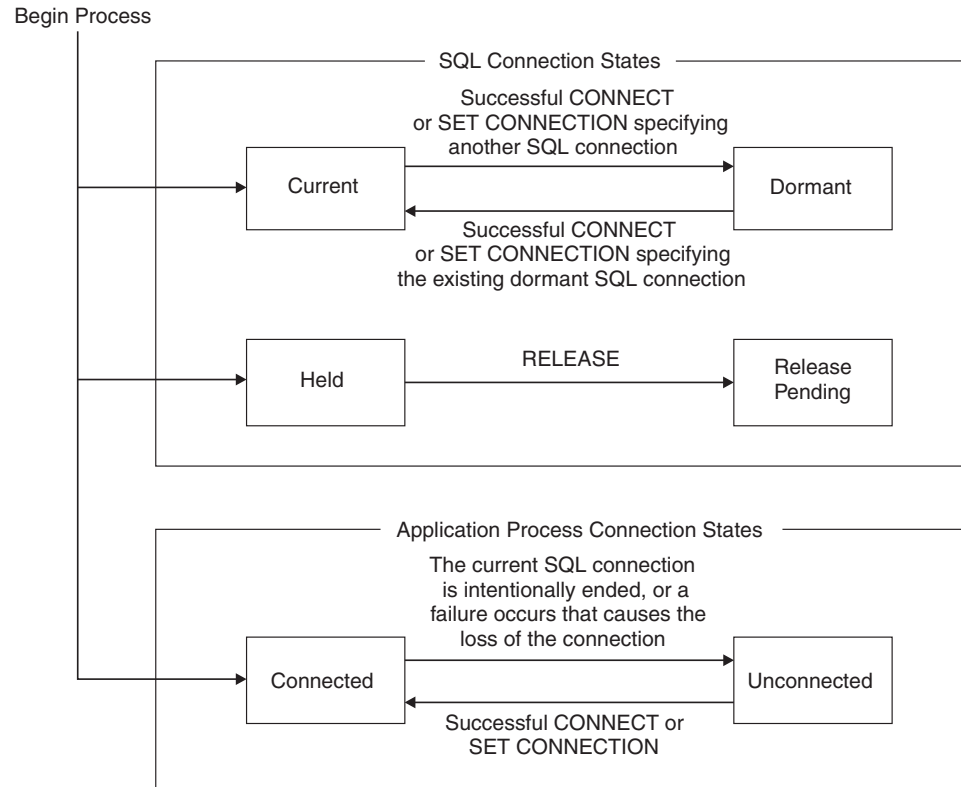


Figure 11. State transitions for an SQL connection and an application process connection in a distributed unit of work

SQL connection states

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

If an application process successfully executes a `CONNECT` statement:

- The current connection is placed in the dormant and held state.
- The new connection is placed in the current and held state.
- The location name is added to the set of existing connections.

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

An SQL connection in the dormant state is placed in the current state using:

- The `SET CONNECTION` statement, or
- The `CONNECT` statement, if the `SQLRULES(DB2) bind` option is in effect.

When an SQL connection is placed in the current state, the previously-current SQL connection, if any, is placed in the dormant state. No more than one SQL

connection in the set of existing connections of an application process can be current at any time. Changing the state of an SQL connection from current to dormant or from dormant to current has no effect on its held or release-pending state.

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

Application process connection states

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Remote unit of work

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

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

executes a commit or rollback operation on the current server to end that unit of work. This restricts the situations in which a CONNECT statement can be executed.

Connection management

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

An application process is in one of four states at any time:

- Connectable and connected
- Unconnectable and connected
- Connectable and unconnected
- Unconnectable and unconnected

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

Figure 12 shows the state transitions:

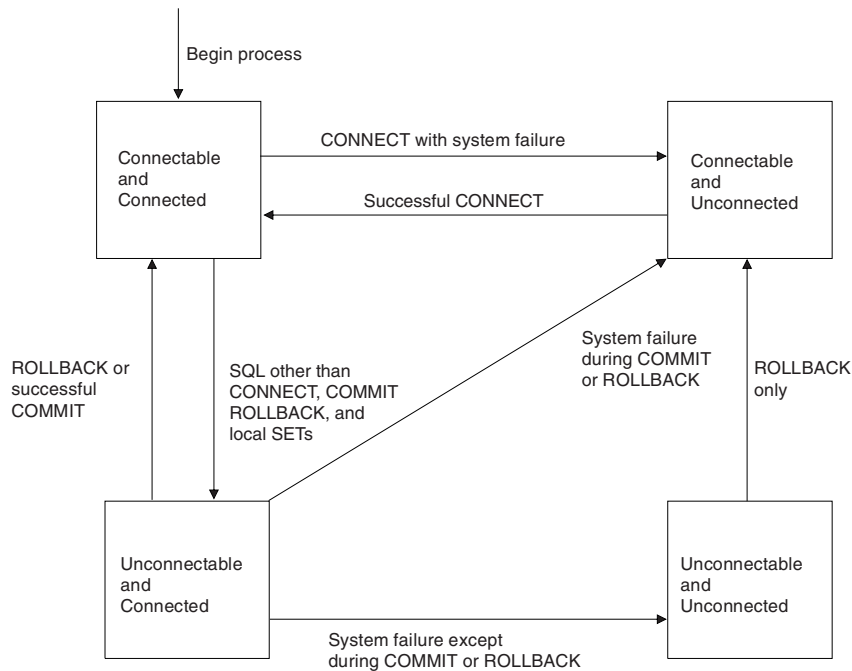


Figure 12. State transitions for an application process connection in a remote unit of work

In the following descriptions of application process connections, CONNECT can mean:

- CONNECT TO
- CONNECT RESET
- CONNECT *authorization*

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

Consecutive CONNECT statements can be executed successfully because CONNECT does not remove an application process from the connectable state. A CONNECT statement does not initiate a new unit of work; a unit of work is initiated by the first SQL statement that accesses data. CONNECT cannot execute successfully when it is preceded by any SQL statement other than CONNECT, COMMIT, RELEASE, ROLLBACK, or SET CONNECTION. To avoid an error, execute a commit or rollback operation before a CONNECT statement is executed.

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

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

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

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

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

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

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

Character conversion

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

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

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

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

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

- An overriding CCSID is specified.

For example, an SQL statement with a descriptor, which requires an SQLDA. In the SQLDA, the CCSID is in the SQLNAME field for languages other than REXX, and in the SQLCCSID field for REXX. (For more information, see “SQL descriptor area (SQLDA)” on page 1656). A DECLARE VARIABLE statement can also be issued to associate a CCSID with the host variables into which data is retrieved from a table.

- The value of the ENCODING bind option or the APPLICATION ENCODING SCHEMA option of the CREATE PROCEDURE or ALTER PROCEDURE statement for a native SQL procedure (static SQL statements) or the CURRENT APPLICATION ENCODING SCHEME special register (for dynamic SQL) is different than encoding scheme of the data being retrieved.
- A mixed character string is assigned to an SBCS column or host variable.
- An SQL statement refers to data that is defined with different CCSIDs.

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

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

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

character set

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

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

code page

A set of assignments of characters to code points. For example, in EBCDIC, “A” is assigned code point X'C1', and “B” is assigned code point X'C2'. In

Unicode UTF-8, “A” is assigned code point X'41', and “B” is assigned code point X'42'. Within a code page, each code point has only one specific meaning.

code point

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

coded character set

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

coded character set identifier (CCSID)

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

EBCDIC

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

encoding scheme

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

- ASCII
- EBCDIC
- Unicode

substitution character

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

Unicode

A universal encoding scheme for written characters and text that enables the exchange of data internationally. It provides a character set standard that can be used all over the world. It provides the ability to encode all characters used for the written languages of the world and treats alphabetic characters, ideographic characters, and symbols equivalently because it specifies a numeric value and a name for each of its characters. It includes punctuation marks, mathematical symbols, technical symbols, geometric shapes, and dingbats. DB2 supports these two encoding forms:

- UTF-8: Unicode Transformation Format, a 8 bit encoding form designed for ease of use with existing ASCII-based systems. UTF-8 can encode any of the Unicode characters. A UTF-8 character is 1,2,3, or 4 bytes in length. A UTF-8 data string can contain any combination of SBCS and MBCS data, including supplementary characters. The CCSID value for data in UTF-8 format is 1208.

- UTF-16: Unicode Transformation Format, a 16 bit encoding form designed to provide code values for over a million characters and a superset of UCS-2. UTF-16 can encode any of the Unicode characters. In UTF-16 encoding, characters are 2 bytes in length, except for supplementary characters, which take two 2 byte string units per character. The CCSID value for data in UTF-16 format is 1200.

Character data (CHAR, VARCHAR, and CLOB) is encoded in Unicode UTF-8. Character strings are also used for mixed data (that is a mixture of single-byte characters and multi-byte characters) and for data that is not associated with any character set (called bit data). Graphic data (GRAPHIC, VARGRAPHIC, and DBCLOB) is encoded in Unicode UTF-16. For a comparison of some UTF-8 and UTF-16 code points for some sample characters, see Figure 15 on page 42. This table shows how a UTF-8 character can be 1 to 4 bytes in length, a non-supplementary UTF-16 character is 2 bytes in length, and how a supplementary character in either UTF-8 or UTF-16 takes two 2 byte code points.

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

“Conversion rules for string assignment” on page 111

“Conversion rules for operations that combine strings” on page 122

“Character conversion in set operations and concatenations” on page 666

Character sets and code pages

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

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

	0	1	2	3	4	5		E	F
0				0	@	P		Â	
1				1	A	Q		À	α
2			†	2	B	R		Å	β
3				3	C	S		Á	γ
4				4	D	T		Ã	σ
5			%	5	E	U		Ä	ε
E			.	>	N			¼	Ö
F			/	*	O			®	

character set ss1
(in code page pp1)

	0	1		A	B	C	D	E	F
0					#				0
1					\$	A	J		1
2				s	%	B	K	S	2
3				t	¬	C	L	T	3
4				u	*	D	M	U	4
5				v	(E	N	V	5
E					!	:	Â	}	
F				À	ç	;	Á	{	

character set ss1
(in code page pp2)

For Unicode, there is only one CCSID for UTF-8 and only one CCSID for UTF-16. The following figure shows how the first 127 single code points for UTF-8 are the same as ASCII with a CCSID of 367. For example, in both UTF-8 and ASCII CCSID 367, an A is X'41' and a 1 is X'31'.

First 127 code points for UTF-8 code page

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

code point: 2F

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

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

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

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

Character encoding for IBM systems is described in *Character Data Representation Architecture Reference and Registry*. For more information on Unicode, see the Unicode standards web site at <http://www.unicode.org>.

Coded character sets and CCSIDS

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

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

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

- DB2 catalog table SYSIBM.SYSSTRINGS
Each row in the catalog table describes a conversion from one coded character set to another.
- z/OS support for Unicode
For more information about the conversion services that are provided, including a complete list of the IBM-supplied conversion tables, see *z/OS Support for Unicode: Using Conversion Services*.

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

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

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

Determining the encoding scheme and CCSID of a string

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

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

The CCSID that is associated with a string value depends on the SQL statement in which the data is referenced and the type of expression. Table 4 on page 44 describes the rules for determining the CCSID that is associated with a string value. Use the Type 1 rules when the SQL statement meets the following conditions:

- The SQL statement operates with a single set of CCSIDs (SBCS, mixed, and graphic). An SQL statement that does not contain any of the following items operates with a single set of CCSIDs:
 - References to columns from multiple tables or views that are defined with CCSIDs from more than one set of CCSIDs (SBCS, mixed, and graphic)

- Graphic hexadecimal (GX) or hexadecimal Unicode (UX) string constants
- References to the XMLCLOB built-in function
- Cast specifications with a CCSID clause
- User-defined table functions
- The SQL statement is not one of the following statements:
 - CALL statement
 - SET assignment statement
 - SET special register
 - VALUE statement
 - VALUES INTO statement

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

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

Source of the string data	Type 1 rules	Type 2 rules
String constant	<p>If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the string constant.</p> <p>Otherwise, the default EBCDIC encoding scheme is used for the string constant.</p> <p>The CCSID is the appropriate character string CCSID for the encoding scheme.</p>	The CCSID is the appropriate character string CCSID for the application encoding scheme. ¹
Hexadecimal string constant (X'...')	<p>If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the string constant.</p> <p>Otherwise, the default EBCDIC encoding scheme is used for the string constant.</p> <p>The CCSID is the appropriate graphic string CCSID for the encoding scheme.</p>	The CCSID is the appropriate character string CCSID for the application encoding scheme. ¹
Graphic string constant (G'...')	<p>If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the graphic string constant.</p> <p>Otherwise, the default EBCDIC encoding scheme is used for the graphic string constant.</p> <p>The CCSID is the graphic string CCSID for the encoding scheme.</p>	The CCSID is the graphic string CCSID for the application encoding scheme. ¹
Graphic hexadecimal constant (GX'...')	Not applicable.	The CCSID is the graphic string CCSID for the application encoding scheme, which must be ASCII or EBCDIC.
Hexadecimal Unicode string constant (UX'...')	Not applicable.	The CCSID is 1200 (UTF-16).

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

Source of the string data	Type 1 rules	Type 2 rules
Special register	<p>If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the special register.</p> <p>Otherwise, the default EBCDIC encoding scheme is used for the special register.</p> <p>The CCSID is the appropriate character string CCSID for the encoding scheme.</p>	The CCSID is the appropriate CCSID for the application encoding scheme. ¹
Column of a table	The CCSID is the CCSID that is associated with the column of the table.	The CCSID is the CCSID that is associated with the column of the table.
Column of a view	The CCSID is the CCSID of the column of the result table of the fullselect of the view definition.	The CCSID is the CCSID of the column of the result table of the fullselect of the view definition.
Expression	The CCSID is the CCSID of the result of the expression.	The CCSID is the CCSID of the result of the expression.
Result of a built-in function	<p>If the description of the function, in Chapter 3, "Functions," on page 259, indicates what the CCSID of the result is, the CCSID is that CCSID.</p> <p>Otherwise, if the description of the function refers to this table for the CCSID, the CCSID is the appropriate CCSID of the CCSID set that is used by the statement for the data type of the result.</p>	<p>If the description of the function, in Chapter 3, "Functions," on page 259, indicates what the CCSID of the result is, the CCSID is that CCSID.</p> <p>Otherwise, if the description of the function refers to this table for the CCSID, the CCSID is the appropriate CCSID of the application encoding scheme for the data type of the result.¹</p>
Parameter of a user-defined routine	The CCSID is the CCSID that was determined when the function or procedure was created.	The CCSID is the CCSID that was determined when the function or procedure was created.
The expression in the RETURN statement of a CREATE statement for a user-defined SQL scalar function	If the expression in the RETURN statement is string data, the encoding scheme is the same as for the parameters of the function. The CCSID is determined from the encoding scheme and the attributes of the data.	The CCSID is determined from the CCSID of the result of the expression specified in the RETURN statement.
String host variable	<p>If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the host variable.</p> <p>Otherwise, the default EBCDIC encoding scheme is used for the host variable.</p> <p>The CCSID is the appropriate CCSID for the data type of the host variable.</p>	<p>At package prepare time, the CCSID is the appropriate CCSID for the data type of the host variable for the application encoding scheme.</p> <p>At run time, the CCSID specified in the declare variable statement, or as an override in the SQLDA. Otherwise, the CCSID is the appropriate CCSID for the data type of the host variable for the application encoding scheme.</p>
Note: If the context is within a check constraint or trigger package, the CCSID is the appropriate CCSID for Unicode instead of the application encoding scheme.		

The following examples show how these rules are applied.

Example 1: Assume that the default encoding scheme for the installation is EBCDIC and that the installation does not support mixed and graphic data. The following

statement conforms to the rules for Type 1 in Table 4 on page 44. Therefore, the X'40' is interpreted as EBCDIC SBCS data because the statement references a table that is in EBCDIC. The CCSID for X'40' is the default EBCDIC SBCS CCSID for the installation.

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

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

Example 2: The following statement references data from two different tables that use different encoding schemes. This statement does not conform to the rules for Type 1 statements in Table 4 on page 44. Therefore, the rules for Type 2 statements are used. The CCSID for X'40' is dependent on the current application encoding scheme. Assuming that the current application encoding scheme is EBCDIC, X'40' represents a single EBCDIC blank.

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

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

Expanding conversions

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

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

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

Contracting conversions

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

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

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

Chapter 2. Language elements

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

The following topics provide information about these language elements:

- “Characters”
- “Tokens” on page 48
- “Identifiers” on page 49
- “Naming conventions” on page 51
- “SQL path” on page 56
- “Qualification of unqualified object names” on page 57
- “Aliases and synonyms” on page 59
- “Authorization IDs, roles, and authorization names” on page 62
- “Data types” on page 69
- “Promotion of data types” on page 95
- “Casting between data types” on page 96
- “Assignment and comparison” on page 102
- “Rules for result data types” on page 119
- “Constants” on page 126
- “Special registers” on page 131
- “Column names” on page 153
- “References to variables” on page 159
- “Host structures in PL/I, C, and COBOL” on page 171
- “Host-variable-arrays in PL/I, C, C++, and COBOL” on page 172
- “Functions” on page 173
- “Expressions” on page 180
- “Predicates” on page 222
- “Search conditions” on page 247
- “Options affecting SQL” on page 248
- “Mappings from SQL to XML” on page 257

Characters

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

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

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

Tokens

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

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

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

Examples:

1 .1 +2 SELECT E 3

- A *delimiter token* is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark (?) is also a delimiter token when it serves as a parameter marker, as explained in "PREPARE" on page 1405.

Examples:

, 'string' "fld1" = .

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

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

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

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

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

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

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

simple comments

Simple comments are introduced with two consecutive hyphens (--).

Simple comments cannot continue past the end of the line. For additional information, see “SQL comments” on page 695.

bracketed comments

Bracketed comments are introduced with `/*` and end with `*/`. A bracketed comment can continue past the end of the line. For additional information, see “SQL comments” on page 695.

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

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

```
select * from DSN8910.EMP where lastname = 'Smith';  
SELECT * FROM DSN8910.EMP WHERE LASTNAME = 'Smith';
```

Identifiers

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

See “Limits in DB2 for z/OS” on page 1588 for the identifier length limits that DB2 imposes.

SQL identifiers

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

Ordinary identifiers

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

An ordinary identifier should not be a reserved word. If a reserved word is used as an identifier in SQL, it must be specified in uppercase and must be a delimited identifier or specified in a host variable. For a list of reserved words, see “Reserved schema names and reserved words” on page 1594.

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

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

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

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

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

Example: The following example is an ordinary identifier:

SALARY

Delimited identifiers

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

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

- Dynamic SQL when the field SQL STRING DELIMITER on installation panel DSNTIPF is set to the quotation mark (") and either of these conditions is true:
 - DYNAMICRULES run behavior applies. For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 64.
 - DYNAMICRULES bind, define, or invoke behavior applies and installation panel field USE FOR DYNAMIC RULES is YES.

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

However, for COBOL application programs, if DYNAMICRULES run behavior does not apply and installation panel field USE FOR DYNAMICRULES is NO, a COBOL compiler option specifies whether the escape character is the quotation mark or apostrophe.

- Static SQL in COBOL application programs. A COBOL compiler option specifies whether the escape character is the quotation mark (") or the apostrophe (').

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

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

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

"VIEW"

Host identifiers

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

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

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

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

Restrictions for distributed access

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

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

Naming conventions

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

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

alias-name

A qualified or unqualified name that designates an alias, table, or view. The unqualified form of *alias-name* is an SQL identifier. An unqualified *alias-name* in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

See “Aliases and synonyms” on page 59 for additional information about aliases.

authorization-name

An SQL identifier that designates a set of privileges. It can also designate a user, a group of users, or a role. For a user or a group of users, DB2 does not control this property. For a role, DB2 does control this property. See “Authorization IDs, roles, and authorization names” on page 62 for the distinction between an authorization name and an authorization ID.

aux-table-name

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

bpname

A name that identifies a buffer pool. The following list shows the names of the different buffer pool sizes.

4KB BP0, BP1, BP2, ..., BP49

8KB BP8K0, BP8K1, BP8K2, ..., BP8K9

16KB BP16K0, BP16K1, BP16K2, ..., BP16K9

32KB BP32K, BP32K1, BP32K2, ..., BP32K9

built-in-type

A qualified or unqualified name that identifies an IBM-supplied data type. A qualified name is SYSIBM followed by a period and the name of the built-in data type. An unqualified name has an implicit qualifier, the schema name, which is determined by the rules in “Qualification of unqualified object names” on page 57.

catalog-name

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

|

|

|

|

clone-table-name

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

collection-id

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

column-name

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

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

constraint-name

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

correlation-name

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

|

|

context-name

An unqualified SQL identifier that designates a trusted context.

cursor-name

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

database-name

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

descriptor-name

A host identifier that designates an SQL descriptor area (SQLDA). See “References to host variables” on page 160 for a description of a host identifier. A descriptor name never includes an indicator variable.

distinct-type-name

A qualified or unqualified name that designates a distinct type.

|

|

|

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

An unqualified distinct type name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the distinct type appears as described by the rules in “Qualification of unqualified object names” on page 57.

external-program-name

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

function-name

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

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

An unqualified function name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears as described by the rules in “Qualification of unqualified object names” on page 57.

host-label

A token that designates a label in a host program.

host-variable

A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, as explained in “References to host variables” on page 160.

index-name

A qualified or unqualified name that designates an index.

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

An unqualified index name is an SQL identifier with an implicit qualifier. The implicit qualifier is an authorization ID, which is determined by the context in which the unqualified name appears as described by the rules in “Qualification of unqualified object names” on page 57.

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

label An SQL identifier that designates a label in an SQL procedure. A label must not be a delimited identifier that includes lowercase letters or special characters.

location-name

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

package-name

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

parameter-name

An SQL identifier that designates a parameter in a procedure or function. A *parameter-name* in an SQL procedure must not be SQLCODE or SQLSTATE.

plan-name

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

procedure-name

A qualified or unqualified name that designates a stored procedure.

A fully qualified procedure name is a three-part name. The first part is a location name that identifies the DBMS at which the procedure is stored.

The second part is the schema name of the stored procedure. The third part is an SQL identifier. A period must separate each of the parts in a qualified name.

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

A one part, or unqualified, procedure name is an SQL identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears, as described by the rules in “Qualification of unqualified object names” on page 57.

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

profile-name

An SQL identifier that corresponds to a RACF profile name.

program-name

An SQL identifier that designates an exit routine.

role-name

An unqualified SQL identifier that designates a role. The identifier cannot be SYSADM, SYSCTRL, or PUBLIC.

routine-version-id

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

savepoint-name

An unqualified SQL identifier that designates a savepoint.

schema-name

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

seclabel-name

A string that corresponds to the value of the RACF security label. It is recommended that name not include national characters (@ (X'7C'), # (X'7B'), or \$ (X'5B')), as an error can occur for some conversions from Unicode.

sequence-name

A qualified or unqualified name that designates a sequence.

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

A one-part or unqualified sequence name is an SQL identifier with an implicit qualifier. The implicit qualifier is an authorization ID, which is determined by the context in which the unqualified name appears as described by the rules in “Qualification of unqualified object names” on page 57.

server-name

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

specific-name

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

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

An unqualified specific name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears as described by the rules in “Qualification of unqualified object names” on page 57.

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

SQL-condition-name

An SQL identifier that designates a condition in an SQL procedure.

SQL-label

An SQL identifier that designates a label in an SQL procedure.

SQL-parameter-name

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

SQL-variable-name

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

statement-name

An SQL identifier that designates a prepared SQL statement.

stogroup-name

An SQL identifier that designates a storage group.

synonym

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

See “Aliases and synonyms” on page 59 for additional information about synonyms.

table-name

A qualified or unqualified name that designates a table.

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

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

A one-part or unqualified table name is an SQL identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second is a schema name, which is determined by the rules set forth in “Qualification of unqualified object names” on page 57. For a declared temporary table, the qualifier (the second part in a three-part name and the first part in a two-part name) must be SESSION. For complete details on specifying a name when a declared temporary table is defined and then later referring to that declared temporary table in other SQL statements, see “DECLARE GLOBAL TEMPORARY TABLE” on page 1202.

table-space-name

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

trigger-name

A qualified or unqualified name that designates a trigger.

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

An unqualified trigger name is an SQL identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the unqualified name appears as described by the rules in “Qualification of unqualified object names” on page 57.

view-name

A qualified or unqualified name that designates a view.

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

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

A one-part or unqualified view name is an SQL identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second is a schema name, which is determined by the context in which the unqualified name appears as described by the rules in “Qualification of unqualified object names” on page 57.

XML-attribute-name

An identifier that is used as an XML attribute name.

XML-element-name

An identifier that is used as an XML element name.

SQL path

The *SQL path* is an ordered list of schema names. DB2 uses the path to resolve the schema name for unqualified data type names (both built-in types and distinct types), function names, and stored procedure names that appear in any context other than as the main object of an ALTER, CREATE, DROP, COMMENT, GRANT, RENAME, or REVOKE statement.

Searching through the path from left to right, DB2 implicitly qualifies the object name with the first schema name in the SQL path that contains the same object with the same unqualified name for which the user has appropriate authorization. For functions, DB2 uses a process called function resolution in conjunction with the SQL path to determine which function to choose because several functions with the same name and number of parameters but different parameter data types might be defined in the same schema or other schemas in the SQL path. (For details, see “Function resolution” on page 175.) For procedures, DB2 selects a matching procedure name only if the number of parameters is also the same.

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

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

The PATH option establishes the SQL path used to resolve:

- Unqualified data type and function names in static SQL statements
- Unqualified procedure names in SQL CALL statements that specify the procedure name as an identifier token (CALL *procedure-name*)

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

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

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

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

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

If schema SYSIBM, SYSPROC, or SYSFUN is not explicitly specified in the SQL path, the schema is implicitly assumed at the front of the path; if all are not specified, they are assumed in the order of SYSIBM, SYSPROC, and SYSFUN. For example, assume that the SQL path is explicitly specified as SYSIBM, GEORGIA, SMITH. As implicitly assumed schemas, SYSPROC and SYSFUN are added to the beginning of the explicit path effectively making the path:

SYSPROC, SYSFUN, SYSIBM, GEORGIA, SMITH

For more information about the SQL path for dynamic SQL, see “CURRENT PATH” on page 142 and “SET PATH” on page 1507.

Qualification of unqualified object names

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

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

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

The default schema is determined as follows:

- For static SQL statements, the default schema is the identifier specified in the QUALIFIER option of the BIND subcommand or the CREATE PROCEDURE or ALTER PROCEDURE statement (for a native SQL procedure). If this option is not in effect for the plan, package, or native SQL procedure, the default schema is the authorization ID of the owner of the plan, package, or native SQL procedure.
- For dynamic SQL statements, the behavior as specified by the combination of the DYNAMICRULES option and the run-time environment determines the default schema. (For a list of these behaviors and the DYNAMICRULES values that determine them, see Table 6 on page 64).
 - If *DYNAMICRULES run behavior* applies, the default schema is the schema in the CURRENT SCHEMA special register. Run behavior is the default.
 - If *bind behavior* applies, the default schema is the identifier that is implicitly or explicitly specified in the QUALIFIER option, as explained for static SQL statements.
 - If *define behavior* applies, the default schema is the owner of the function or stored procedure (the owner is the definer).
 - If *invoke behavior* applies, the default schema is the authorization ID of the invoker of the function or stored procedure.

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

Unqualified distinct type, function, procedure, and specific names

The qualification of unqualified distinct type, function, stored procedure, and specific names depends on the SQL statement in which the unqualified name appears.

- If an unqualified name is the main object of an ALTER, CREATE, COMMENT, DROP, GRANT, or REVOKE statement, the name is implicitly qualified with a schema name as follows:
 - In a static statement, the implicit schema name is the identifier specified in the QUALIFIER option of the BIND subcommand or the CREATE PROCEDURE or ALTER PROCEDURE statement (for a native SQL procedure). If this option is not in effect for the plan, package, or procedure, the implicit qualifier is the authorization ID of the owner of the plan, package, or procedure.
 - In a dynamic statement, the implicit schema name is the schema in the CURRENT SCHEMA special register.
- Otherwise, the implicit schema name for the unqualified name is determined as follows:
 - For distinct type names, DB2 searches the SQL path and selects the first schema in the path such that the distinct type exists in the schema and the user has authorization to use the distinct type.

- For function names, DB2 uses the SQL path in conjunction with function resolution, as described in “Function resolution” on page 175.
- For stored procedure names in CALL statements, DB2 searches the SQL path and selects the first schema in the path such that the schema contains a procedure with the same name and number of parameters and the user has authorization to use the procedure.
- For stored procedure names in ASSOCIATE LOCATORS and DESCRIBE PROCEDURE statements, DB2 does not use the SQL path because an implicit schema name is not generated for these statements.

For information about the SQL path, see “SQL path” on page 56.

Aliases and synonyms

A table or view can be referred to in an SQL statement by its name, by an alias that has been defined for its name, or by a synonym that has been defined for its name. Thus, aliases and synonyms can be thought of as alternative names for tables and views.

An alias can be defined at a local server and can refer to a table or view that is at the current server or a remote server. The alias name can be used wherever the table name or view name can be used to refer to the table or view in an SQL statement. The rules for forming an alias name are the same as the rules for forming a table name or a view name, as explained below. A fully qualified alias name (a three-part name) can refer to an alias at a remote server. However, the table or view identified by the alias at the remote server must exist at the remote server.

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

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

Statements that use three-part names and refer to distributed data result in either DRDA access to the remote site or DB2 private protocol access. DRDA access for three-part names is used when the plan or package that contains the query to distributed data is bound with bind option DBPROTOCOL(DRDA), or the value of field DATABASE PROTOCOL on installation panel DSNTIP5 is DRDA and bind option DBPROTOCOL(PRIVATE) was not specified when the plan or package was bound. When an application program uses three-part name aliases for remote objects and DRDA access, the application program must be bound at each location that is specified in the three-part names. Also, each alias needs to be defined at the local site. An alias at a remote site can refer to yet another server as long as a referenced alias eventually refers to a table or view.

The option of referencing a table or view by an alias or a synonym is not explicitly shown in the syntax diagrams or mentioned in the description of SQL statements. But they can be referred to in an SQL statement, with two exceptions: a local alias cannot be used in the CREATE ALIAS statement, and a synonym cannot be used

in the CREATE SYNONYM statement. If an alias is used in the CREATE SYNONYM statement, it must identify a table or view at the current server. The synonym is defined on the name of that table or view. If a synonym is used in the CREATE ALIAS statement, the alias is defined on the name of the table or view identified by the synonym.

The effect of using an alias or a synonym in an SQL statement is that of text substitution. For example, if A is an alias for table Q.T, one of the steps involved in the preparation of SELECT * FROM A is the replacement of 'A' by 'Q.T'. Likewise, if S is a synonym for Q.T, one of the steps involved in the preparation of SELECT * FROM S is the replacement of 'S' by 'Q.T'.

The differences between aliases and synonyms are as follows:

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

Authorization, privileges, and object ownership

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

The two forms of authorization are administrative authority and privileges.

Administrative authority

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

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

Privileges

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

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

Object ownership

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

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

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

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

Authorization IDs, roles, and authorization names

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

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

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

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

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

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

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

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

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

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

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

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

```
CREATE TABLE TDEPT LIKE DSN8910.DEPT;  
GRANT SELECT ON TDEPT TO KEENE;
```

Also assume that your site has not replaced the default exit routine for connection authorization and that you have not executed the SET CURRENT SQLID

statement. Thus, when the GRANT statement is prepared and executed by SPUFI, the SQL authorization ID is SMITH. KEENE is an authorization name that is specified in the GRANT statement.

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

Authorization IDs and schema names

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

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

- CREATEIN privilege
- ALTERIN privilege
- DROPIN privilege

Authorization IDs and statement preparation

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

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

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


The authorization ID that is used for the authorization checking of embedded SQL statements is that of the owner of the plan or package. If the application is bound in a trusted context using the ROLE AS OBJECT OWNER clause, the authorization ID that is used for authorization checking is the role that owns the plan or package, otherwise the authorization ID is the authorization ID of the owner of the plan or package. If an embedded SQL statement refers to tables or views at a DB2 subsystem other than the one at which the plan or package is bound, the authorization checking is deferred until run time. For more information on this, see “Authorization IDs and remote execution” on page 67.

If VALIDATE(BIND) is specified, the privileges required to use or manipulate objects at the DB2 subsystem at which the plan or package is bound must exist at

bind time. If the privileges or the referenced objects do not exist and SQLERROR(NOPACKAGE) is in effect, the bind operation is unsuccessful. If SQLERROR(CONTINUE) is specified, then the bind is successful and any statements in error are flagged. If any statements in error are flagged, an error will occur when you attempt to execute them at run time.

If a plan or package is bound with VALIDATE(RUN), authorization checking is still performed at bind time, but the referenced objects and the privileges required to use these objects need not exist at this time. If any privilege required for a statement does not exist at bind time, an authorization check is performed whenever the statement is first executed within a unit of work, and all privileges required for the statement must exist at that time. If any privilege does not exist, execution of the statement is unsuccessful. When the authorization check is performed at run time, it is performed against the plan or package owner, not the SQL authorization ID. For the effect of this option on cursors, see “DECLARE CURSOR” on page 1191.

Related reference

 The DSN command and its subcommands (DB2 Commands)

Authorization IDs and dynamic SQL

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

This discussion applies to dynamic SQL statements that refer to objects at the current server. For those that refer to objects elsewhere, see “Authorization IDs and remote execution” on page 67.

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

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

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

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

Table 6. How DYNAMICRULES and the run-time environment determine dynamic SQL statement behavior (continued)

DYNAMICRULES value	Behavior of dynamic SQL statements	
	Stand-alone program environment	User-defined function or stored procedure environment
INVOKEBIND	Bind behavior	Invoke behavior
Note: BIND and RUN values can be specified for packages, plans, and native SQL procedures. The other values can be specified for packages and native SQL procedures but not for plans.		

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

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

Run behavior

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

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

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

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

Implicit qualification: As explained under “Qualification of unqualified object names” on page 57, when an SQL statement is dynamically prepared, the values of the CURRENT SCHEMA special register is used as the implicit qualifier. For example, it is used as the implicit qualifier for all unqualified tables, aliases, views, indexes, and sequences.

Bind behavior

The same rules that are used to determine the authorization ID for static (embedded) statements are used for dynamic statements. DB2 uses the authorization ID of the owner of the package or plan for authorization checking of dynamic SQL statements, as explained in detail under “Authorization IDs and statement preparation” on page 63.

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

Implicit qualification: The identifier specified in the QUALIFIER option of the bind command that is used to bind the SQL statements, or the CREATE PROCEDURE or ALTER PROCEDURE statement that is used to create a version of an SQL procedure is the implicit qualifier for all unqualified tables, views, aliases, indexes, and sequences. If the QUALIFIER option was not used when the plan, package, or native SQL procedure was created or last changed, the implicit qualifier is the owner of the plan, package, or native SQL procedure.

Define behavior

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

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

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

Invoke behavior

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

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

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

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

Related reference

➞ Privileges required for using dynamic SQL statements (DB2 Administration Guide)

➞ BIND and REBIND options (DB2 Commands)

Authorization IDs and remote execution

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

DRDA access with DB2 for z/OS only

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

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

A user who invokes an application that has a plan at the local DB2 must have the EXECUTE privilege on the plan recorded in the DB2 catalog of the requester. If the application uses a package bound at a server other than the local DB2 and the package is not a user-defined function, stored procedure, or trigger package, the plan owner must have the EXECUTE privilege on the package recorded in the DB2 catalog of the server. The plan needs no other privilege to execute the package. EXECUTE authority is also required to use a package that is a user-defined function, stored procedure, or trigger package; however, the plan owner is not the required holder of the privilege, as explained in *DB2 Administration Guide*. In the case of trigger packages, the authorization ID of the SQL statement that activates the trigger must have the EXECUTE privilege on the trigger. Again, all these privileges must be recorded in the DB2 catalog of the server.

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

processing when it first connects to a server or when it reuses a CICS or IMS thread that has a different primary authorization ID. For details on connection processing, see *DB2 Administration Guide*.

If an application uses Recoverable Resources Manager Services attachment facility (RRSAF) and has no plan, authority to execute the package is determined in the same way as when the requester is not DB2 for z/OS, which is described next under “DRDA access with a server or requester other than DB2.”

DRDA access with a server or requester other than DB2

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

DB2 for z/OS as the server: If the requester is not a DB2 for z/OS subsystem, there is no DB2 application plan involved. In this case, the privilege set of the authorization ID, which is determined by the DYNAMICRULES behavior, must have the EXECUTE privilege on the package. Dynamic SQL statements in the package are executed according to the DYNAMICRULES behavior, as described in “Authorization IDs and dynamic SQL” on page 64.

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

DB2 private protocol access

Any statement that refers to a table or view at a DB2 subsystem other than the current server and is bound with bind option DBPROTOCOL(PRIVATE) is executed using DB2 private protocol access.

Such statements are processed as deferred embedded SQL statements. The additional cost of the dynamic bind occurs once for every unit of work where the statement is executed. Authorization to execute such statements is checked against the owner of a plan or package. Authorization IDs for executing dynamic statements are handled just as they are for DRDA access. In either case, the pertinent privileges must be recorded in the catalog of the DBMS that executes the statement.

Authorization ID translations

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

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

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

Related concepts

 Communications database for the server (DB2 Administration Guide)

 Communications database for the requester (DB2 Administration Guide)

Other security measures

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

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

Data types

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

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

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

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

Figure 16 on page 70 shows the built-in data types that DB2 supports.

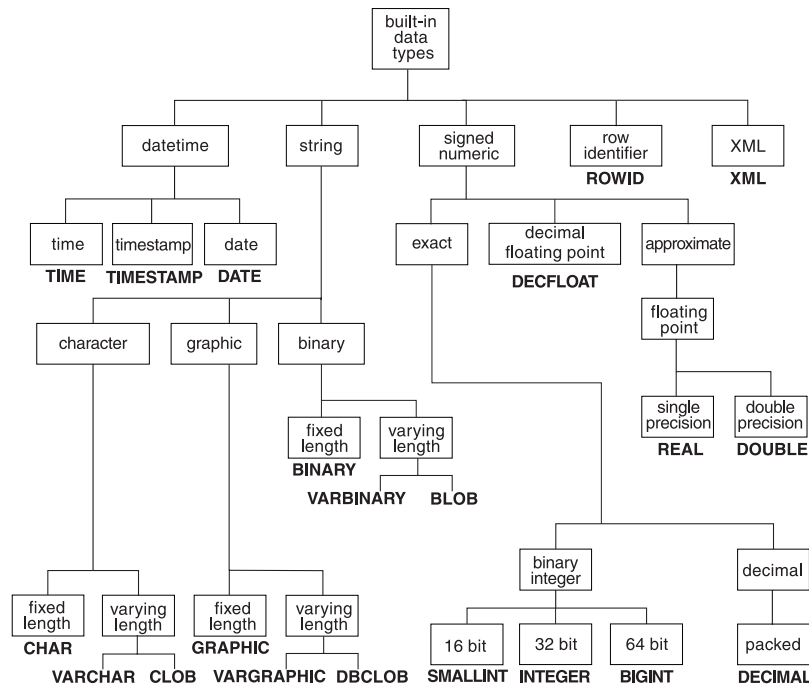


Figure 16. Built-in data types supported by DB2

Nulls

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

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

Numbers

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

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

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

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

Small integer (SMALLINT)

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

Large integer (INTEGER)

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

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

Big integer (BIGINT)

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

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

Single precision floating-point (REAL)

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

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

Double precision floating-point (DOUBLE or FLOAT)

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

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

Decimal (DECIMAL or NUMERIC)

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

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

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

Decimal floating-point (DECFLOAT)

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

The maximum precision is 34 digits.

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

In addition to the finite numbers, decimal floating point numbers are able to represent one of the following named special values:

- Infinity - a value that represents a number whose magnitude is infinitely large.

- Quiet NaN - a value that represents undefined results which does not cause an invalid number condition.
- Signaling NaN - a value that represents undefined results which will cause an invalid number condition if used in any numerical operation.

When a number has one of these special values, its coefficient and exponent are undefined. The sign of an infinity is significant (that is, it is possible to have both positive and negative infinity). The sign of a NaN has no meaning for arithmetic operations. INF can be used in place of INFINITY.

Numeric host variables

Numeric host variables can be defined in all languages with a few exceptions.

Binary integer variables can be defined in all host languages.

Floating-point variables can be defined in all host languages. All languages, except Java, support System/390[®] floating-point format. Assembler, C, C++, PL/I, and Java also support IEEE floating-point format. In assembler, C, C++, and PL/I programs, the precompiler option FLOAT tells DB2 whether floating-point variables contain data in System/390 floating-point format or IEEE floating-point format. The contents of floating-point host variables must match the format that is specified with the FLOAT SQL processing option.

Decimal variables can be defined in all host languages except Fortran.

In COBOL, decimal numbers can be represented in the following formats:

- Packed decimal format, denoted by USAGE PACKED-DECIMAL or COMP-3
- External decimal format, denoted by USAGE DISPLAY with SIGN LEADING SEPARATE
- NATIONAL decimal format denoted by USAGE NATIONAL and SIGN LEADING SEPARATE

Decimal floating-point variables can be defined in Java.

String representations of numeric values

String representations of numeric values can be used in some contexts. A valid string representation of a numeric value must conform to the rules for numeric constants.

The encoding scheme in use determines what type of strings can be used for string representation of numeric values. For ASCII and EBCDIC, a string representation of a numeric value must be a character string. For UNICODE, a string representation of a numeric value can be either a character string or a graphic string. Thus, the only time a graphic string can be used for a numeric value is when the encoding scheme is UNICODE.

When a decimal, decimal floating-point, or floating-point number is cast to a string (for example, using a CAST specification), the implicit decimal point is replaced by the default decimal separator character that is in effect when the statement is prepared. When a string is cast to a decimal, decimal floating-point, or floating-point value (for example, using a CAST specification), the default decimal separator character in effect when the statement was prepared is used to interpret the string.

For more information, see “Constants” on page 126.

Subnormal numbers and underflow

The decimal floating-point data type has a set of non-zero numbers that fall outside the range of normal decimal floating-point values. These numbers are called subnormal.

Non-zero numbers whose adjusted exponents are less than E_{\min}^4 are called subnormal numbers. These subnormal numbers are accepted as operands for all operations and can result from any operation. If a result is subnormal before any rounding occurs, the subnormal condition is returned.

For a subnormal result, the minimum values of the exponent becomes $E_{\min} - (\text{precision} - 1)$, called E_{tiny} , where precision is the working precision. If necessary, the result will be rounded to ensure that the exponent is no smaller than E_{tiny} . If the result becomes inexact during rounding, an underflow condition is returned. A subnormal result does not always return the underflow condition but will always return the subnormal condition.

When a number underflows to zero during a calculation, its exponent will be E_{tiny} . The maximum value of the exponent is unaffected.

The maximum value of the exponent for subnormal numbers is the same as the minimum value of the exponent which can arise during operations that do not result in subnormal numbers. This occurs where the length of the coefficient in decimal digits is equal to the precision.

Character strings

A *character string* is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is called the *empty string*. The empty string should not be confused with the null value.

Default CCSIDs

The value of the field MIXED DATA (on installation panel DSNTIPF) determines the default CCSIDs for a character string.

The following table shows how the value of the field MIXED DATA (on installation panel DSNTIPF) determines the default CCSIDs for a character string.

Table 7. Default CCSIDs for character strings

Encoding scheme	Value of MIXED DATA field	Default attribute
ASCII or EBCDIC	NO	Character: SBCS The value of the ASCII CODED CHAR SET or EBCDIC CODED CHAR SET field on installation panel determines the system CCSID for SBCS data.
ASCII or EBCDIC	YES	Character: MIXED The value of the ASCII CODED CHAR SET or EBCDIC CODED CHAR SET field on installation panel DSNTIPF determines the system CCSID for SBCS data, MIXED, and graphic data.

Table 7. Default CCSIDs for character strings (continued)

Encoding scheme	Value of MIXED DATA field	Default attribute
Unicode	Not applicable	Character: MIXED The CCSIDs are: <ul style="list-style-type: none"> • 367 for SBCS data • 1208 for MIXED data • 1200 for graphic data

Fixed-length character strings

When fixed-length character string distinct types, columns, and variables are defined, the length attribute is specified, and all values have the same length. For a fixed-length character string, the length attribute must be between 1 and 255 inclusive.

Varying-length character strings

The types of varying-length character strings are VARCHAR and *character large object* (CLOB). A CLOB is a type of LOB. A CLOB column is useful for storing large amounts of character data, such as documents written with a single character set.

When varying-length character strings, distinct types, columns, and variables are defined, the maximum length is specified and this length becomes the length attribute except for C NUL-terminated strings. Actual values might have a smaller value. For varying-length character strings, the length specifies the number of bytes.

For a VARCHAR string, the length attribute must be between 1 and 32704. For a VARCHAR column, the maximum for the length attribute is determined by the record size that is associated with the table, as described in Maximum record size the description of the CREATE TABLE statement. For a CLOB string, the length attribute must be between 1 and 2 147 483 647 inclusive. (2 147 483 647 is 2 gigabytes minus 1 byte.) For more information about CLOBs, see “Large objects (LOBs)” on page 85.

Character string variables

Character string variables follow specific rules for use in host languages.

- Fixed-length character string variables can be used in all languages except REXX and Java. In C, CHAR string variables are limited to a length of 1.
- Varying-length character string variables can be used in all host languages with the following exceptions:
 - Fortran: varying-length non-LOB character strings cannot be used.
 - Assembler, C, and COBOL: varying-length non-LOB strings are simulated as described in *DB2 Application Programming and SQL Guide*. In C, NUL-terminated strings can also be used.
 - REXX: CLOBs and DBCLOBs cannot be used.

Character string encoding schemes

The method of representing DBCS and MBCS characters within a mixed string differs among the encoding schemes.

Each character string is further defined as one of the following subtypes:

Bit data

Data that is not associated with a coded character set and, therefore, is never converted. The CCSID for bit data is X'FFFF' (65535). The bytes do not represent characters.

Bit data is a form of character data. The pad character is a blank for assignments to bit data; the pad character is X'00' for assignments to binary data. It is recommended that binary data be used instead of character for bit data.

If both operands in a predicate are EBCDIC, both operands are padded with X'40'. Otherwise, both operands are padded with X'20'. For example, if both operands are ASCII, or if one operand is ASCII and the other operand is EBCDIC, both are padded with X'20'.

SBCS data

Data in which every character is represented by a single byte. Each SBCS string has an associated CCSID. If necessary, an SBCS string is converted before it is used in an operation with a character string that has a different CCSID.

Mixed data

Data that can contain a mixture of characters from a single-byte character set (SBCS) and a multiple-byte character set (MBCS). Each mixed string has an associated CCSID. If necessary, a mixed string is converted before an operation with a character string that has a different CCSID. If a mixed data string contains an MBCS character, it cannot be converted to SBCS data.

EBCDIC mixed data can contain shift characters, which are not MBCS data.

When the encoding scheme is Unicode or the DB2 installation is defined to support mixed data, DB2 recognizes MBCS sequences within mixed data string when performing character sensitive operations. These operations include parsing, character conversion, and the pattern matching specified by the LIKE predicate.

Character strings with a CLOB data type can only be SBCS or MIXED. BLOB should be used for binary strings.

The method of representing DBCS and MBCS characters within a mixed string differs among the encoding schemes.

- ASCII reserves a set of code points for SBCS characters and another set as the first half of DBCS characters. When it encounters the first half of a DBCS character, the system reads the next byte in order to obtain the complete character.
- EBCDIC makes use of two special code points:
 - A shift-out character (X'0E') to introduce a string of DBCS characters.
 - A shift-in character (X'0F') to end a string of DBCS characters.

DBCS sequences within mixed data strings are recognized as the string is read from left to right. At any time, the reading of the string is in SBCS mode or DBCS mode. In SBCS mode, which is the initial mode, any byte other than a shift-out is interpreted as an SBCS character. When a shift-out is read, the mode switches to DBCS mode. In DBCS mode, the next byte and every second byte after that byte is interpreted as the first byte of a DBCS character unless it is a shift character. If the byte is a shift-out, an error occurs. If the byte is a shift-in, the mode returns to SBCS mode. An error occurs if the mode is still DBCS mode

after processing the last byte of the string. Because of the shift characters, EBCDIC mixed data requires more storage than ASCII mixed data.

- UTF-8 is a varying-length encoding of byte sequences. The high bits indicate the part of the sequence to which a byte belongs. The first byte indicates the number of bytes to follow in a byte sequence.

Examples

The same mixed date character string can be represented as character and hexadecimal data in different encoding schemes.

For the same mixed data character string, the following table shows character and hexadecimal representations of the character string in different encoding schemes. In EBCDIC, the shift-out and shift-in characters are needed to delineate the double-byte characters.

Table 8. Example of a character string in different encoding schemes

Data type and encoding scheme	Character representation	Hexadecimal representation (with spaces separating each character)
9 bytes in ASCII	gen ki	8CB3 67 65 6E 8B43 6B 69
13 bytes in EBCDIC	gen ki	0E 4695 0F 87 85 95 0E 45B9 0F 92 89
11 bytes in Unicode UTF-8	gen ki	E58583 67 65 6E E6B097 6B 69

Because of the differences of the representation of mixed data strings in ASCII, EBCDIC, and Unicode, mixed data is not transparently portable. To minimize the effects of these differences, use varying-length strings in applications that require mixed data and operate on ASCII, EBCDIC, and Unicode data.

String unit specifications

The ability to specify string units for certain built-in functions and on the CAST specification allows you to process string data in a more "character-based manner" than a "byte-based manner". The string unit determines the length in which the operation is to occur. You can specify CODEUNITS32, CODEUNITS16, or OCTETS as the units for the operation.

CODEUNITS32

Specifies that Unicode UTF-32 is the units for the operation. CODEUNITS32 is useful when an application wants to process data in a simple fixed-length format and needs the same answer regardless of the storage format of the data (ASCII, EBCDIC, UTF-8, or UTF-16). Although the answers are in terms of CODEUNITS32, the data is not converted to UTF-32 to perform the function.

CODEUNITS16

Specifies that Unicode UTF-16 is the units for the operation. CODEUNITS16 is useful when an application wants to know how many double-byte characters are in a string.

OCTETS

Specifies that bytes are the units for the operation. OCTETS is often used when an application is interested in allocation buffer space or when operations need to use simple byte processing.

Determining the length of a string by counting in string units (CODEUNITS16 or CODEUNITS32) or bytes (OCTETS) can result in different answers. When OCTETS is specified, the length of a string is determined by simply counting the number of bytes in the string. Counting by CODEUNITS16 or CODEUNITS32 gives the same answer unless the data contains supplementary characters. For information about the difference between CODEUNITS16 and CODEUNITS32 when the data contains supplementary characters, see “Difference between CODEUNITS16 and CODEUNITS32” on page 78.

Example: Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'. The first two queries, which count the length of the string in CODEUNITS32 and CODEUNITS16, returns the same value, 6. The third query, which counts the length of the string in OCTETS, returns the value 7. These values are the length of the string as expressed in the string units that are specified.

```
SELECT CHARACTER_LENGTH(NAME, CODEUNITS32)
FROM T1 WHERE NAME = 'Jürgen';
SELECT CHARACTER_LENGTH(NAME, CODEUNITS16)
FROM T1 WHERE NAME = 'Jürgen';
SELECT CHARACTER_LENGTH(NAME, OCTETS)
FROM T1 WHERE NAME = 'Jürgen';
```

The following table shows the UTF-8, UTF-16, and UTF-32 representations of 'Jürgen'.

Format	Representation of the name 'Jürgen'
UTF-8	x'4AC3BC7267656E'
UTF-16	x'004A00FC007200670065006E'
UTF-32	x'0000004A000000FC0000007200000067000000650000006E'

The bold highlighting in the table demonstrates how the representation of the character *ü* in 'Jürgen' differs between the three string units:

- The UTF-8 representation of the character *ü* is X'C3BC'. In UTF-8, characters that are not in the Latin-1 subset (essentially a through z, A through Z, and 0 through 9), such as accented characters or Japanese characters, are represented by multiple bytes.
- The UTF-16 representation of the character *ü* is X'00FC'. In UTF-16, each character is represented in 2 bytes. UTF-16 supplementary characters take two 2-byte code points.
- The UTF-32 representation of the character *ü* is X'000000FC'. In UTF-32, each character is represented in 4 bytes.

Specifying the string units on a built-in function does not affect the data type or the CCSID of the result of the function. If necessary, DB2 converts the data to Unicode for evaluation when CODEUNITS32 or CODEUNITS16 is specified. DB2 always evaluates the data in the encoding scheme of the output data when OCTETS is specified. For more information about the data types and CCSIDs of the results of functions, see the description of each function.

Differences between the way that characters are represented in ASCII, EBCDIC, and Unicode can affect the results of your queries.

Example: Assume that NAME is a VARCHAR(128) column, encoded in EBCDIC (CCSID 37), that contains the value 'Mit freundlichen Grüßen, Jürgen'. The following query returns the string 'Mit freundlichen Grüß':


```
SELECT SUBSTRING(C1,1,21,CODEUNITS16)
FROM T1 WHERE C1 = 'Mit freundlichen Grüßen, Jürgen';
```

The following table shows the result data in more detail:

Format	Representation of 'Mit freundlichen Grüß'
EBCDIC	D489A340869985A4958493898388859540C799 DC59
UTF-8	4D697420667265756E646C696368656E204772 C3BCC39F
UTF-16	004D0069007400200066007200650075006E0064006C0069006300680065006E002000470072 00FC00DF

The bold highlighting in the table shows that the representation of the characters *ü* and *ß* in UTF-8 and UTF-16 each require two bytes. If OCTETS had been specified on the SUBSTRING function to have the string evaluated in UTF-8 bytes instead of EBCDIC OCTETS or CODEUNITS16, the result would have been 'Mit freundlichen Grü'. The character *ß* would have been lost.

Difference between CODEUNITS16 and CODEUNITS32:

CODEUNITS16 and CODEUNITS32 return the same answer unless the data contains supplementary characters.

A supplementary character is represented as two UTF-16 code units or one UTF-32 code unit. In UTF-8, a non-supplementary character is represented by 1 to 3 bytes and a supplementary character is represented by 4 bytes. In UTF-16, a non-supplementary character is represented by one CODEUNIT16 code unit or 2 bytes, and a supplementary character is represented by two CODEUNIT16 code units or 4 bytes. In UTF-32, a character is represented by one CODEUNIT32 code unit or 4 bytes. Thus, CODEUNITS16 and CODEUNITS32 return different answers when the data contains supplementary characters.

Example 1: The following table shows the hexadecimal values for the mathematical bold capital A and the Latin capital letter A. The mathematical bold capital A is a supplementary character that is represented by 4 bytes in UTF-8, UTF-16, and UTF-32.

Character	UTF-8 representation	UTF-16 representation	UTF-32 representation
Unicode value \u1D400 - 'A'	X'F09D9080'	X'D835DC00'	X'0001D400'
MATHEMATICAL BOLD CAPITAL A			
Unicode value \u0041 - 'A'	X'41'	X'0041'	X'00000041'
LATIN CAPITAL LETTER A			

Assume that C1 is a VARCHAR(128) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080'). The following similar queries return different answers:

```
-- Query:                                     -- Returns the value:
SELECT CHARACTER_LENGTH(C1,CODEUNITS32) FROM T1;      -- 1
SELECT CHARACTER_LENGTH(C1,CODEUNITS16) FROM T1;      -- 2
SELECT CHARACTER_LENGTH(C1,OCTETS) FROM T1;           -- 4
```


Example 2: Assume that C1 is a VARCHAR(128) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080'). The following similar queries return different answers.

-- Query:	-- Returns the value:
SELECT HEX(SUBSTRING(C1,1,1,CODEUNITS32)) FROM T1;	-- X'F09D9080'
SELECT HEX(SUBSTRING(C1,1,1,CODEUNITS16)) FROM T1;	-- X'20'
SELECT HEX(SUBSTRING(C1,1,2,CODEUNITS16)) FROM T1;	-- X'F09D9080'
SELECT HEX(SUBSTRING(C1,1,1,OCTETS)) FROM T1;	-- X'20'
SELECT HEX(SUBSTR(C1,1,1)) FROM T1;	-- X'F0'

The value X'20' is the pad (blank) character.

Determining the length attribute of the final result:

When CODEUNITS32, CODEUNITS16, or OCTETS is specified for a function or the CAST specification, the length attribute of the final result string is calculated by applying specific formulas depending on which function is specified.

To determine the final result of a function or the CAST specification, DB2 might need to use an intermediate result string if CODEUNITS32 or CODEUNITS16 is specified, depending on the encoding scheme of the data:

- ASCII and EBCDIC data require the use of a UTF-16 intermediate result string when either CODEUNITS32 or CODEUNITS16 is specified.
- UTF-8 data requires the use of a UTF-16 intermediate result string only when CODEUNITS16 is specified.

Regardless of whether an intermediate string is used, when CODEUNITS32, CODEUNITS16, or OCTETS is specified for a function or the CAST specification, the length attribute of the final result string is calculated by applying the formulas that are described in the following table. The length attributes that are calculated at each step in the formulas are measured in bytes, unless indicated otherwise.

Table 9. Formulas for the length attribute of the final result string

Function	Determination of the length attribute of the string ¹
CAST specification	Follow these three steps to determine the length attribute of the final result:
CHAR	<p>1. Length of the intermediate string (IML)</p> <p>When CODEUNITS32 or CODEUNITS16 is specified:</p> <ul style="list-style-type: none"> • If the source string is not in Unicode CCSID 1200, 1208, or 367, convert the source string to CCSID 1200, using the formulas in Table 35 on page 126 to determine the result length of the intermediate string (IML). • If source string is in Unicode CCSID 1208 or 367, and CODEUNITS16 is specified, convert the source string to CCSID 1200, using the formulas in Table 35 on page 126 to determine the result length of the intermediate string (IML). • Otherwise, the intermediate string is the same as the source string. <p>When OCTETS is specified:</p> <ul style="list-style-type: none"> • If the CCSID of the source string is different from the CCSID of the result of the function, convert the source string to the CCSID of the result of the function, using the formulas in Table 35 on page 126 to determine the result length of the intermediate string (IML). • Otherwise, the intermediate string is the same as the source string. <p>Exception: For the GRAPHIC and VARGRAPHIC function, if the source string is EBCDIC, the source is widened with prefix X'42' before the source string is converted to CCSID 1200 and the length of the intermediate string is determined.</p>
CLOB	
DBCLOB	
GRAPHIC	
VARCHAR	
VARGRAPHIC	
	<p>2. Result length attribute of the intermediate string (rl)</p> <p>The result length (rl) of the intermediate string depends on whether a <i>length</i> argument was explicitly specified.</p> <p>If <i>length</i> was not specified, the result length (rl) attribute is:</p> $rl = IML$ <p>If <i>length</i> was specified, the result length (rl) attribute is:</p> <pre> IF (ol * n) < r_IML THEN rl = ol * n ELSE IF intermediate string is in CCSID 1200 (UTF-16) THEN rl = MIN(ol * n , IML + (r * 2)) ELSE rl = MIN(ol * n , IML + r) </pre> <p>Where:</p> <ul style="list-style-type: none"> • ol = original <i>length</i> argument, expressed in the specified string units • n = 4 bytes for CODEUNITS32 2 bytes for CODEUNITS16 • IML = length of the intermediate string • r_IML = IML rounded up to next multiple of n • r = ol - (r_IML/n), expressed in the specified string units <p>The calculation for r is an estimate of the difference between the <i>length</i> argument and the estimated number of characters of the input argument, expressed in the specified string units.</p>
	<p>3. Length of the final result string (the result of the function)</p> <p>The result length attribute of the final string is determined by converting the result length (rl) of the intermediate string to the CCSID of the result of the function, using the formulas in Table 35 on page 126, if CCSID conversion is necessary. Otherwise, the result length attribute of the final string is rl.</p>

Table 9. Formulas for the length attribute of the final result string (continued)

Function	Determination of the length attribute of the string ¹
CHARACTER_LENGTH	<p>Follow these three steps to determine the length attribute of the final result:</p> <ol style="list-style-type: none"> 1. Length of the intermediate string (IML) The length of the intermediate string (IML) is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).) 2. Result length attribute of the intermediate string (r1) The result length (r1) attribute is always 4 (the length of an integer): $r1 = 4$ 3. Length of the final result string (the result the function) The length of the final result of the function is always an integer.
LOCATE LOCATE_IN_STRING POSITION	<p>Follow these three steps to determine the length attribute of the final result:</p> <ol style="list-style-type: none"> 1. Length of the intermediate string (IML) The length of the intermediate string (IML) is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).) For the LOCATE, LOCATE_IN_STRING, and POSITION functions, this applies to both the <i>source-string</i> and <i>search-string</i>. If the CCSIDs of intermediate strings for the converted <i>source-string</i> and <i>search-string</i> differ, the intermediate string for the <i>search-string</i> is converted to the CCSID of intermediate string for the <i>source-string</i>. 2. Result length attribute of the intermediate string (r1) The result length (r1) attribute of the intermediate string depends on whether the start and length arguments are constants. If the <i>start</i> and <i>length</i> arguments are both constants, the result length attribute is: $r1 = L1 - \text{MIN} (\text{MAX} (0, L1 - (V2 - 1) * n), V3 * m) + L4$ If at least one argument (the <i>start</i> or <i>length</i> argument) is not a constant, the result length attribute is: $r1 = L1 + L4$ Where: <ul style="list-style-type: none"> L1 and L4 are the length attributes of the intermediate strings of the <i>source-string</i> and <i>insert-string</i>, respectively. V2 and V3 are the <i>start</i> and <i>length</i> values, respectively, expressed in the specified string units. $m =$ 1 if the intermediate string of the <i>source-string</i> is not CCSID 1200 (UTF-16) 2 if the intermediate string of the <i>source-string</i> is CCSID 1200 (UTF-16) $n =$ 4 bytes for CODEUNITS32 2 bytes for CODEUNITS16 3. Length of the final result string (the result the function) The length of the final result is the same as the length of the final result for the CAST specification. (See Length attribute of the final result string (the result of the function).)
INSERT OVERLAY	<p>Follow these three steps to determine the length attribute of the final result:</p> <ol style="list-style-type: none"> 1. Length of the intermediate string (IML) The length of the intermediate string (IML) for both the <i>source-string</i> and the <i>insert-string</i> is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).) If the CCSIDs of the intermediate strings for the converted <i>source-string</i> and <i>insert-string</i> differ, the intermediate string for the <i>insert-string</i> is converted to the CCSID of the intermediate string for the <i>source-string</i>. 2. Result length attribute of the intermediate string (r1) The result length (r1) attribute of the intermediate string depends on whether the start and length arguments are constants. If the <i>start</i> and <i>length</i> arguments are both constants, the result length attribute is: $r1 = L1 - \text{MIN} (\text{MAX} (0, L1 - (V2 - 1) * n), V3 * m) + L4$ If at least one argument (the <i>start</i> or <i>length</i> argument) is not a constant, the result length attribute is: $r1 = L1 + L4$ Where: <ul style="list-style-type: none"> L1 and L4 are the length attributes of the intermediate strings of the <i>source-string</i> and <i>insert-string</i>, respectively. V2 and V3 are the <i>start</i> and <i>length</i> values, respectively, expressed in the specified string units. $m =$ 1 if the intermediate string of the <i>source-string</i> is not CCSID 1200 (UTF-16) 2 if the intermediate string of the <i>source-string</i> is CCSID 1200 (UTF-16) $n =$ 4 bytes for CODEUNITS32 2 bytes for CODEUNITS16 3. Length of the final result string (the result the function) The length of the final result is the same as the length of the final result for the CAST specification. (See Length attribute of the final result string (the result of the function).)

Table 9. Formulas for the length attribute of the final result string (continued)

Function	Determination of the length attribute of the string ¹
LEFT	Follow these three steps to determine the length attribute of the final result:
RIGHT	<p>1. Length of the intermediate string (IML) The length of the intermediate string (IML) is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).)</p> <p>2. Result length attribute of the intermediate string (rl) The result length (rl) attribute is the same as the length of the intermediate string: $rl = IML$</p> <p>3. Length of the final result string (the result of the function) The result length attribute of the final string is determined by converting the result length (rl) of the intermediate string to the CCSID of the result of the function, using the formulas in Table 35 on page 126, if CCSID conversion is necessary. Otherwise, the result length attribute of the final string is rl. The result length attribute of the final string is: $MIN(\text{length of source string, length of CCSID converted string})$</p>
SUBSTRING	<p>Follow these three steps to determine the length attribute of the final result:</p> <p>1. Length of the intermediate string (IML) The length of the intermediate string (IML) is determined the same way as for the CAST specification. (See Length of the intermediate string (IML).)</p> <p>2. Result length attribute of the intermediate string (rl) The result length (rl) of the intermediate string depends on whether a <i>length</i> argument was explicitly specified. If <i>length</i> was not specified, the result length (rl) attribute is: $rl = IML$ If <i>length</i> was specified, the result length (rl) attribute is: $rl = MIN(ol * n, IML)$ Where: <ul style="list-style-type: none"> ol = original <i>length</i> argument, expressed in the specified string units n = 4 bytes for CODEUNITS32 2 bytes for CODEUNITS16 IML = length of the intermediate string </p> <p>3. Length of the final result string (the result of the function) The length of the final result string is the same as for LEFT built-in function.</p>
Note:	
1. The final value of the calculation for each length attribute (IML, rl, and the final result of the function) is limited by the maximum length of the function or by the maximum length of the corresponding data type of the result, whichever is applicable. Each length attribute is expressed in terms of bytes.	

Example 1: Assume that T1 is a table encoded in EBCDIC and C1 is a CHAR(26) column (SBCS data with EBCDIC CCSID 37). The CHAR function is invoked in the following statement:

```
SELECT CHAR(C1,10,CODEUNITS32) as COL1 FROM T1;
```

DB2 uses an intermediate string to evaluate the function and determines the intermediate and final result string lengths using these steps:

1. C1, which is SBCS EBCDIC 37 data, is converted to Unicode 1200 (UTF-16). The result length of the conversion (using the formula from Table 35 on page 126, $X * 2$) is $26 * 2$. Thus, the length of the intermediate string is 52 bytes (IML = 52).
2. The CHAR function is evaluated against the first 10 UTF-32 characters in this string. The result length attribute is 40 bytes ($r1 = o1 * n$ or $10 * 4$) because $o1 * n < r_IML$ or $40 < 52$.
3. The 40 bytes of the string are converted back to SBCS EBCDIC 37. The result length of the conversion (using the formula from Table 35 on page 126, $X * .5$) is $40 * .5$. Thus, the length of the final result of the functions is 20 bytes.

Example 2: This example is similar to the first example, except that the specified length for the function is 20 instead of 10. Assume that T1 is a table encoded in EBCDIC and C1 is a CHAR(26) column (SBCS data with EBCDIC CCSID 37). The CHAR function is invoked in the following statement:

```
SELECT CHAR(C1,20,CODEUNITS32) as COL1 FROM T1;
```

DB2 uses an intermediate string to evaluate the function and determines the intermediate and final result string lengths using these steps:

1. C1, which is SBCS EBCDIC 37 data, is converted to Unicode 1200 (UTF-16). The result length of the conversion (using the formula from Table 35 on page 126, $X * 2$) is $26 * 2$. Thus, the length of the intermediate result string is 52 bytes (IML = 52).
2. The CHAR function is evaluated against the first 20 UTF-32 characters in this intermediate string. However, because the estimated number of characters in the intermediate string, as expressed in the specified string units, is only 13 characters (r_IML/n or $52/4$), the intermediate string must be padded with 7 padding characters to satisfy the 20 characters that are requested ($r = o1 - (r_IML/n)$ or $20 - 13$). In Unicode 1200 (UTF-16), each padding character takes 2 bytes.
The result length attribute is then calculated to be 66 bytes ($r1 = \text{MIN}(o1 * n, \text{IML} + (r * 2))$ or $\text{MIN}(20 * 4, 52 + 14)$) because $o1 * n < r_IML$ or $80 < 52$ is not true.
3. The 66 bytes of the string are converted back to SBCS EBCDIC 37. The result length of the conversion (using the formula from Table 35 on page 126, $X * .5$) is $66 * .5$. Thus, the length of the final result of the function is 33 bytes.

Graphic strings

A *graphic string* is a sequence of double-byte characters.

The length of the string is the number of characters in the sequence. Like character strings, graphic strings can be empty. An empty string should not be confused with the null value.

Fixed-length graphic strings

When fixed-length graphic string distinct types, columns, and variables are defined, the length attribute is specified and all values have the same length. For a fixed-length graphic string, the length attribute must be between 1 and 127 inclusive. A fixed-length graphic string column can also be called a GRAPHIC column.

Varying-length graphic strings

The types of varying-length graphic strings are VARGRAPHIC and *double-byte character large object (DBCLOB)*. DBCLOB is a type of LOB. A DBCLOB column is

useful for storing large amounts of double-byte character data, such as documents written with a single double-byte character set.

When varying-length graphic strings, distinct types, columns, and variables are defined, the maximum length is specified and this length becomes the length attribute. Actual values might have a smaller value. For a varying-length graphic string, the length attribute must be between 1 and 16352.

For a varying-length graphic string column, the maximum for the length attribute is determined by the record size associated with the table, as described Maximum record size in the description of the CREATE TABLE statement. For a DBCLOB string, the length attribute must be between 1 and 1 073 741 823 inclusive. In UTF-16, although supplementary characters use two 2-byte code points, supplementary characters are still considered double-byte characters. For more information about DBCLOBs, see “Large objects (LOBs)” on page 85.

Graphic string variables

Graphic string variables must follow certain rules.

Variables with a graphic string type cannot be defined in Fortran. In addition, graphic string variables follow these rules:

- Fixed-length graphic string host variables can be defined in all host languages, except REXX and Java. In C, fixed-length graphic-string variables are limited to a length of 1.
- Varying-length graphic string variables can be defined in all host languages, with the exception of DBCLOBs which cannot be used in REXX.

Graphic string encoding schemes

Each graphic string can be further defined as either double-byte data or Unicode data.

Double-byte data

Data in which every character is represented by a character from the double-byte character set (DBCS) that does not include shift-out or shift-in characters. Each double-byte graphic string has an associated ASCII or EBCDIC CCSID.

Unicode data

Data that contains characters represented by two bytes, except supplementary characters, which take two 2-byte code points per character. Each Unicode graphic string is encoded using UTF-16. The CCSID for UTF-16 is 1200.

String units in built-in functions

When working with graphic strings, you can specify the string unit in which the operation is to take place for certain built-in functions and the CAST specification. The string unit determines the length in which the operation is to occur.

For more information about string units, see “String unit specifications” on page 76.

Binary strings

A *binary string* is a sequence of bytes.

The length of a binary string is the number of bytes in the sequence. Binary strings are not associated with any CCSID. There are three binary string data types: BINARY, VARBINARY (BINARY VARYING) and BLOB (BINARY LARGE OBJECT).

Fixed-length binary strings

The type of fixed-length binary strings is BINARY. When fixed-length binary string distinct types, columns, and variables are defined, the length attribute is specified, and all values have the same length. For a fixed-length binary string, the length attribute must be between 1 and 255 inclusive.

Varying-length binary strings

The types of varying-length binary strings are VARBINARY (BINARY VARYING) and BLOB (BINARY LARGE OBJECT)

When varying-length binary strings, distinct types, columns, and variables are defined, the maximum length is specified and this length becomes the length attribute. Actual length values might have a smaller value than the length attribute value. For varying-length binary strings, the actual length specifies the number of bytes in the string.

For a VARBINARY string, the length attribute must be between 1 and 32704. For a VARBINARY string column, the maximum for the length attribute is determined by the record size that is associated with the table, as described in "Maximum record size" on the description of the CREATE TABLE statement. Like a varying-length character string, varying-length binary string could be an empty string.

A binary string column is useful for storing non-character data, such as encoded or compressed data, pictures, voice, and mixed media. Another use is to hold structured data for exploitation by distinct types, user-defined functions, and stored procedures. Note, that although binary strings and FOR BIT DATA character strings might be used for similar purposes, the two data types are not compatible. The BINARY, BLOB, VARBINARY built-in functions and CAST specification can be used to change a FOR BIT DATA character string into a binary string.

Large objects (LOBs)

The term *large object (LOB)* refers to any of the following data types: CLOB, DBCLOB, or BLOB.

CLOB A *character large object (CLOB)* is a varying-length string with a maximum length of 2 147 483 647 bytes (2 gigabytes minus 1 byte). A CLOB is designed to store large SBCS data or mixed data, such as lengthy documents. For example, you can store information such as an employee resume, the script of a play, or the text of novel in a CLOB. Alternatively, you can store such information in UTF-8 in a mixed CLOB. A CLOB is a varying-length character string.

DBCLOB

A *double-byte character large object (DBCLOB)* is a varying-length string with a maximum length of 1 073 741 823 double-byte characters. A DBCLOB is designed to store large DBCS data. For example, you could store the information mentioned for CLOB (an employee resume, the script for a play, or the text of a novel) in UTF-16 in a DBCLOB. A DBCLOB is a varying-length graphic string.

BLOB A *binary large object (BLOB)* is a varying-length string with a maximum

length of 2 147 483 647 bytes (2 gigabytes minus 1 byte). A BLOB is designed to store non-traditional data such as pictures, voice, and mixed media. BLOBs can also store structured data for use by distinct types and user-defined functions. A BLOB is a binary string.

Although BLOB strings and FOR BIT DATA character strings might be used for similar purposes, the two data types are not compatible. The BLOB function can be used to change a FOR BIT DATA character string into a BLOB string.

Restrictions using LOBs

With a few exceptions, you can use LOBs in the same contexts in which you can use other varying-length strings.

The following table shows the contexts in which LOBs cannot be used.

Table 10. Contexts in which LOBs cannot be used

Context of usage	LOB (CLOB, DBCLOB, or BLOB)
A GROUP BY clause	Not allowed
An ORDER BY clause	Not allowed
A CREATE INDEX statement	Not allowed
A SELECT DISTINCT statement	Not allowed
A subselect of a set operation except UNION ALL	Not allowed
Predicates	Cannot be used in any predicate except EXISTS, LIKE, and NULL. This restriction includes a <i>simple-when-clause</i> in a CASE expression. <i>expression</i> WHEN <i>expression</i> in a <i>simple-when-clause</i> is equivalent to a predicate with <i>expression=expression</i> .
The definition of primary, unique, and foreign keys	Not allowed
Check constraints	Not allowed

Manipulating LOBs using locators

A LOB locator is a host variable with a value that represents a single LOB value in the database server. LOB locators provide a mechanism for you to easily manipulate very large objects in application programs without having to store the entire LOB value on the client machine where the application program might be running.

Because LOB values can be very large, the transfer of these values from the database server to host variables in client application programs can be time consuming. Also, application programs typically process LOB values a piece at a time, rather than as a whole. For these cases, the application can use a *large object locator* (LOB locator) to reference the LOB value.

For example, when selecting a LOB value, an application program could handle the value in either of these two ways:

- Select the entire LOB value and place it into an equally large host variable. This method is acceptable if the application program is going to process the entire LOB value at once.
- Select the LOB value into a LOB locator. Then, using the LOB locator, the application program can issue subsequent database operations on the LOB value (such as using it as a parameter to the scalar functions SUBSTR, CONCAT, COALESCE, LENGTH, doing an assignment, searching the LOB value with

LIKE or POSSTR, or using it as a parameter to a user-defined function or procedure) by supplying the LOB locator value as input. The resulting output of the LOB locator operation, for example, the amount of data that is assigned to a client host variable, would then typically be a small subset of the input LOB value.

LOB locators can also represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```
SUBSTR(lob_value_1 CONCAT lob_value_2 CONCAT lob_value_3 , 42, 6000000)
```

For non-locator-based host variables in an application program, when a null value is selected into that host variable, the indicator variable is set to -1, signifying that the value is null. For LOB locators, however, the meaning of indicator variables is slightly different. Because a LOB locator host variable itself can never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value (the server does not track null values with valid LOB locators).

A LOB locator represents a value, not a row or location in the database. Therefore, after a value is selected into a LOB locator, no action that is subsequently performed on the original row or table will affect the value that is referenced by the LOB locator. The value associated with a LOB locator is valid until the transaction ends, or until the LOB locator is explicitly freed, whichever comes first.

A LOB locator is also not a database type, and it is never stored in the database. As a result, it cannot participate in views or check constraints. However, values for the SQLTYPE field of the SQLDA exist for LOB locators so that they can be described within an SQLDA structure that is used by FETCH, OPEN, CALL and EXECUTE statements.

For more information about manipulating LOBs with LOB locators, see *DB2 Application Programming and SQL Guide*.

Datetime values

Datetime values are neither strings nor numbers. Nevertheless, datetime values can be used in certain arithmetic and string operations and are compatible with certain strings.

Moreover, strings can represent datetime values, as discussed in “String representations of datetime values” on page 89.

Date

A *date* is a three-part value (year, month, and day) designating a point in time using the Gregorian calendar, which is assumed to have been in effect from the year 1 A.D.

⁴ The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to 28, 29, 30, or 31, depending on the month and year.

4. Historical dates do not always follow the Gregorian calendar. Dates between 1582-10-04 and 1582-10-15 are accepted as valid dates although they never existed in the Gregorian calendar.

The internal representation of a date is a string of 4 bytes. Each byte consists of two packed decimal digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.

The length of a DATE column as described in the catalog is the internal length, which is 4 bytes. The length of a DATE column as described in the SQLDA is the external length, which is 10 bytes unless a date exit routine was specified when your DB2 subsystem was installed. (Writing a date exit routine is described in *DB2 Administration Guide*.) In that case, the string format of a date can be up to 255 bytes in length. Accordingly, DCLGEN⁵ defines fixed-length string variables for DATE columns with a length equal to the value of the field LOCAL DATE LENGTH on installation panel DSNTIP4, or a length of 10 bytes if a value for the field was not specified.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB or DBCLOB.

Time

A *time* is a three-part value (hour, minute, and second) designating a time of day using a 24-hour clock. The range of the hour part is 0 to 24. The range of the minute and second parts is 0 to 59. If the hour is 24, the minute and second parts are both zero.

The internal representation of a time is a string of 3 bytes. Each byte consists of two packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column as described in the catalog is the internal length which is 3 bytes. The length of a TIME column as described in the SQLDA is the external length which is 8 bytes unless a time exit routine was specified when the DB2 subsystem was installed. (Writing a time exit routine is described in *DB2 Administration Guide*.) In that case, the string format of a time can be up to 255 bytes in length. Accordingly, DCLGEN⁵ defines fixed-length string variables for TIME columns with a length equal to the value of the field LOCAL TIME LENGTH on installation panel DSNTIP4, or a length of 8 bytes if a value for the field was not specified.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB or DBCLOB.

Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) that represents a date and time. The time portion of a timestamp includes a fractional specification of microseconds.

The internal representation of a timestamp is a string of 10 bytes, each of which consists of two packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds.

The length of a TIMESTAMP column as described in the catalog is the internal length which is 10 bytes. The length of a TIMESTAMP column as described in the SQLDA is the external length which is 26 bytes. DCLGEN therefore defines 26-byte, fixed-length string variables for TIMESTAMP columns.

5. DCLGEN is a DB2 DSN subcommand for generating table declarations for designated tables or views. The declarations are stored in z/OS data sets, for later inclusion in DB2 source programs.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB or DBCLOB.

Datetime host variables

Character-string host variables are normally used to contain date, time, and timestamp values. However, date, time, and timestamp host variables can also be specified in Java as `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`, respectively.

String representations of datetime values

Dates, times, and timestamp values can be represented by strings. For many host languages, there are no special SQL constants for datetime values and, except for Java, no host variables with a data type of date, time, or timestamp. Thus, to be retrieved, a datetime value must be assigned to a string variable.

Values whose data types are date, time, or timestamp are represented in an internal form that is transparent to the user of SQL. But dates, times, and timestamps can also be represented by strings. These representations directly concern the SQL user because for many host languages there are no special SQL constants for datetime values and, except for Java, no host variables with a data type of date, time, or timestamp. Thus, to be retrieved, a datetime value must be assigned to a string variable.

Each datetime value is assigned an encoding scheme. This encoding scheme is used when the datetime value is converted from its internal form to the string representation in the form of the mixed CCSID if the field MIXED DATA is YES on installation panel DSNTIPF. Otherwise the SBCS CCSID of the assigned encoding scheme is used. For Unicode, the mixed CCSID is always used. The following table shows how the encoding scheme is determined:

Table 11. The encoding scheme of datetime values

Datetime expression	Result encoding scheme
Columns	The same encoding scheme as the table that contains the column
Host variables	If the statement references: <ul style="list-style-type: none"> • A single encoding scheme - The same encoding scheme • Multiple encoding schemes - The application encoding scheme
Special registers	If the statement references: <ul style="list-style-type: none"> • A single encoding scheme - The same encoding scheme • Multiple encoding schemes - The application encoding scheme
Expressions	If the statement references: <ul style="list-style-type: none"> • A single encoding scheme - The same encoding scheme • Multiple encoding schemes - The application encoding scheme

For ASCII and EBCDIC, a string representation of a datetime value must be a character string. For Unicode, a string representation of a datetime value can be either a character string or a graphic string. Thus, the only time a graphic string can be used for a datetime value is when the encoding scheme is Unicode.

In host languages other than Java, a datetime value must be assigned to a string variable. When a date or time is assigned to a string variable, the string format is determined by a precompiler option or subsystem parameter. When a string representation of a datetime value is used in other operations, it is converted to a datetime value. However, this can be done only if the string representation is recognized by DB2 or an exit provided by the installation and the other operand is a compatible datetime value. An input string representation of a date or time with LOCAL specified must have an actual length that is not greater than 255 bytes.

Datetime values that are represented by strings can appear in contexts requiring values whose data types are date, time, or timestamp. A string representation of a date, time or timestamp can be passed as an argument to the DATE, TIME, or TIMESTAMP function to obtain a datetime value. A CAST specification can also be used to turn a character representation of a date, time, or timestamp into a datetime value.

Date strings:

A string representation of a date is a string that starts with a digit and has a length of at least 8 characters. Trailing blanks can be included, leading blanks are not allowed, and leading zeros can be omitted in the month and day portions.

The following table shows the valid string formats for dates. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use. For an installation-defined date string format, the format and length must have been specified when DB2 was installed. They cannot be listed here.

Table 12. Formats for string representations of dates

Format name	Abbreviation	Date format	Example
International Standards Organization	ISO	yyyy-mm-dd	1987-10-12
IBM USA standard	USA	mm/dd/yyyy	10/12/1987
IBM European standard	EUR	dd.mm.yyyy	12.10.1987
Japanese industrial standard Christian era	JIS	yyyy-mm-dd	1987-10-12
Installation-defined	LOCAL	Any installation- defined form	—

Time strings:

A string representation of a time is a string that starts with a digit, and has a length of at least 4 characters. Trailing blanks can be included, leading blanks are not allowed, and leading zeros can be omitted in the hour part of the time; seconds can be omitted entirely. If you choose to omit seconds, an implicit specification of 0 seconds is assumed. Thus 13.30 is equivalent to 13.30.00.

The following table shows the valid string formats for times. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use. In the case of an installation-defined time string format, the format and length must have been specified when your DB2 subsystem was installed. They cannot be listed here.

Table 13. Formats for string representations of times

Format name	Abbreviation	Time format	Example
International Standards Organization ¹	ISO ¹	hh:mm:ss	13.30.05
IBM USA standard	USA	hh:mm AM or PM	1:30 PM
IBM European standard	EUR	hh:mm:ss	13.30.05
Japanese industrial standard Christian era	JIS	hh:mm:ss	13:30:05
Installation-defined	LOCAL	Any installation-defined form	—

Note: 1. This is an earlier version of the ISO format. JIS can be used to get the current ISO format.

In the USA format:

- The minutes can be omitted, thereby specifying 00 minutes. For example, 1 PM is equivalent to 1:00 PM.
- The letters A, M, and P can be lowercase.
- A single blank must precede the AM or PM.
- The hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM.

Using the ISO format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

- 12:01 AM through 12:59 AM correspond to 00.01.00 through 00.59.00
- 01:00 AM through 11:59 AM correspond to 01.00.00 through 11.59.00
- 12:00 PM (noon) through 11:59 PM correspond to 12.00.00 through 23.59.00
- 12:00 AM (midnight) corresponds to 24.00.00
- 00:00 AM (midnight) corresponds to 00.00.00

Timestamp strings:

A string representation of a timestamp is a string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-mm-dd-hh.mm.ss.nnnnnnn*. Trailing blanks can be included, leading blanks are not allowed, and leading zeros can be omitted in the month, day, and hour part of the timestamp; trailing zeros can be truncated or omitted entirely from microseconds. If you choose to omit any digit of the microseconds portion, an implicit specification of 0 is assumed. Thus, '1990-3-2-8.30.00.10' is equivalent to '1990-03-02-08.30.00.100000'. A timestamp whose time part is '24.00.00.000000' is also accepted.

SQL statements also support the ODBC or JDBC string representation of a timestamp as an input value only. The ODBC and JDBC string representation of a timestamp has the form *yyyy-mm-dd hh:mm:ss.nnnnnnn*.

LOCAL date and time exits: For LOCAL, the date exit for ASCII data is DSNXVDTA, the date exit for EBCDIC is DSNXVDTX, and the date exit for Unicode is DSNXVDTU. For LOCAL, the time exit for ASCII data is DSNXVTMA, the time exit for EBCDIC is DSNXVTMX, and the time exit for Unicode is DSNXVTMU.

Restrictions on the use of local datetime formats

Certain restrictions apply on the use of date and time values as input, as output, and for use in binding a package.

The following rules apply to the character-string representation of dates and times:

For input: In distributed operations, DB2 as a server uses its local date or time routine to evaluate host variables and constants. This means that character-string representation of dates and times can be:

- One of the standard formats
- A format recognized by the server's local date/time exit

For output: With DRDA access, DB2 as a server returns date and time host variables in the format defined at the server. With DB2 private protocol access, DB2 as a server returns date and time host variables in the format defined at the requesting system. To have date and time host variables returned in another format, use `CHAR(date-expression, XXXX)` where XXXX is JIS, EUR, USA, ISO, or LOCAL to explicitly specify the specific format.

For BIND PACKAGE COPY: When binding a package using the COPY option, DB2 uses the ISO format for output values unless the SQL statement explicitly specifies a different format. Input values can be specified in the format described previously.

Row ID values

A *row ID* is a value that uniquely identifies a row in a table. A column or a host variable can have a row ID data type.

A ROWID column enables queries to be written that navigate directly to a row in the table because the column implicitly contains the location of the row. Each value in a ROWID column must be unique. Although the location of the row might change, for example across a table space reorganization, DB2 maintains the internal representation of the row ID value permanently. When a row is inserted into the table, DB2 generates a value for the ROWID column unless one is supplied. If a value is supplied, it must be a valid row ID value that was previously generated by DB2 and the column must be defined as GENERATED BY DEFAULT. Users cannot update the value of a ROWID column.

The internal representation of a row ID value is transparent to the user. The value is never subject to character conversion because it is considered to contain BIT data. The length of a ROWID column as described in the LENGTH column of catalog table SYSCOLUMNS is the internal length, which is 17 bytes. The length as described in the LENGTH2 column of catalog table SYSCOLUMNS is the external length, which is 40 bytes.

A ROWID column can be either user-defined or implicitly generated by DB2. You can use the CREATE TABLE statement or the ALTER TABLE statement to define a ROWID column. If you define a LOB column in a table and the table does not have a ROWID column, DB2 implicitly generates a ROWID column. DB2:

- Creates the column with a name of `DB2_GENERATED_ROWID_FOR_LOBSnn`. DB2 appends *nn* only if the column name already exists in the table, replacing *nn* with '00' and incrementing by '1' until the name is unique within the row.
- Defines the column as GENERATED ALWAYS.
- Appends the column to the end of the row after all the other explicitly defined columns.

An implicitly hidden ROWID column is called a *hidden ROWID column*. A hidden ROWID column is not visible in SQL statements unless you refer to the column directly by name. For example, assume that DB2 generated a hidden ROWID column named DB2_GENERATED_ROWID_FOR_LOBS for table MYTABLE. The result table for a SELECT * statement for table MYTABLE would not contain that ROWID column. However, the result table for SELECT COL1, DB2_GENERATED_ROWID_FOR_LOBS would include the hidden ROWID column.

If you add a ROWID column to a table that already has a hidden ROWID column, DB2 ensures that the corresponding values in each column are identical. If the ROWID column that you add is defined as GENERATED BY DEFAULT, DB2 changes the attribute of the hidden ROWID column to GENERATED BY DEFAULT.

In a distributed data environment, the row ID data type is not supported for DB2 private protocol access. For information about using row IDs, see *DB2 Application Programming and SQL Guide*.

XML values

An XML value represents well-formed XML in the form of an XML document, XML content, or a sequence of XML nodes.

An XML value that is stored in a table as the value of a column that is defined with the XML data type must be a well-formed XML document. XML values are processed in an internal representation that is not comparable to any string value. The only predicates that can be applied to the XML data type are the XMLEXISTS predicate and the NULL predicate.

An XML value can be transformed into a serialized string value that represents the XML document by using the XMLSERIALIZE function. Similarly, a string value that represents an XML document can be transformed to an XML value by using the XMLPARSE function.

The XML data type has a variable length and allows for a wide range of sizes. Although data of this type has no defined maximum length, it does have an effective maximum length limit when treated as a serialized string value that represents XML. The maximum effective length is the same as the DB2 limit for a LOB data value. DB2 treats XML string data in a similar manner as LOB data to accommodate very large XML values. Thus, XML values are constrained by the same maximum length limit as LOB data. Unlike the LOB data type which has a LOB locator type, there is no XML locator type.

Restrictions when using XML values: With a few exceptions, you can use XML values in the same contexts in which you can use other data type. XML values cannot be used in the following contexts:

- SELECT lists that are preceded by the DISTINCT clause
- GROUP BY clauses
- ORDER BY clauses
- A subselect of a fullselect with a set operator that is not UNION ALL
- Basic predicates, quantified predicates, BETWEEN predicates, DISTINCT predicates, IN predicates, or LIKE predicates
- Aggregate functions with the DISTINCT keyword
- Primary, unique, or foreign keys

- CREATE TYPE statements

No host languages have any built-in support for an XML data type.

Distinct types

A *distinct type* is a user-defined data type that shares its internal representation with a built-in data type (its *source type*), but is considered to be a separate and incompatible data type for most operations.

For example, the semantics for a picture type, a text type, and an audio type that all use the built-in data type BLOB for their internal representation are quite different. A distinct type is created with the SQL statement CREATE TYPE.

For example, the following statement creates a distinct type named AUDIO:

```
CREATE TYPE AUDIO AS BLOB (1M);
```

Although AUDIO has the same representation as the built-in data type BLOB, it is a separate data type that is not comparable to a BLOB or to any other data type. This inability to compare AUDIO to other data types allows functions to be created specifically for AUDIO and assures that these functions cannot be applied to other data types.

The name of a distinct type is qualified with a schema name. The implicit schema name for an unqualified name depends on the context in which the distinct type appears. If an unqualified distinct type name is used:

- In a CREATE TYPE statement or the object of DROP, COMMENT, GRANT, or REVOKE statement, DB2 uses the normal process of qualification by authorization ID to determine the schema name.
- In any other context, DB2 uses the SQL path to determine the schema name. DB2 searches the schemas in the path, in sequence, and selects the first schema in the path such that the distinct type exists in the schema and the user has authorization to use the data type. For a description of the SQL path, see “SQL path” on page 56.

A distinct type does not automatically acquire the functions and operators of its source type because they might not be meaningful. (For example, it might make sense for a “length” function of the AUDIO type to return the length in seconds rather than in bytes.) Instead, distinct types support *strong typing*. Strong typing ensures that only the functions and operators that are explicitly defined on a distinct type can be applied to that distinct type. However, a function or operator of the source type can be applied to the distinct type by creating an appropriate user-defined function. The user-defined function must be sourced on the existing function that has the source type as a parameter. For example, the following series of SQL statements shows how to create a distinct type named MONEY based on data type DECIMAL(9,2), how to define the + operator for the distinct type, and how the operator might be applied to the distinct type:

```
CREATE TYPE MONEY AS DECIMAL(9,2);
CREATE FUNCTION "+" (MONEY,MONEY)
  RETURNS MONEY
  SOURCE SYSIBM."+"(DECIMAL(9,2),DECIMAL(9,2));
CREATE TABLE SALARY_TABLE
  (SALARY MONEY,
   COMMISSION MONEY);
SELECT SALARY + COMMISSION FROM SALARY_TABLE;
```


A distinct type is subject to the same restrictions as its source type. For example, if a CLOB value is not allowed as input to a function, you cannot specify a distinct type that is based on a CLOB as input.

The comparison operators are automatically generated for distinct types, except those that are based on a CLOB, DBCLOB, or BLOB. In addition, DB2 automatically generates functions for every distinct type that support casting from the source type to the distinct type and from the distinct type to the source type. For example, for the AUDIO type created above, these are generated cast functions:

```
FUNCTION schema-name.BLOB (schema-name.AUDIO) RETURNS SYSIBM.BLOB (1M)
FUNCTION schema-name.AUDIO (SYSIBM.BLOB (1M)) RETURNS schema-name.AUDIO
```

In a distributed data environment, distinct types are not supported for DB2 private protocol access.

Promotion of data types

Data types can be classified into groups of related data types. Within such groups, an order of precedence exists in which one data type is considered to precede another data type. This precedence enables DB2 to support the *promotion* of one data type to another data type that appears later in the precedence order.

For example, DB2 can promote the data type CHAR to VARCHAR and the data type INTEGER to DOUBLE PRECISION; however, DB2 cannot promote a CLOB to a VARCHAR.

DB2 considers the promotion of data types when:

- Performing function resolution (see “Function resolution” on page 175)
- Casting distinct types (see “Casting between data types” on page 96)
- Assigning built-in data types to distinct types (see “Distinct type assignments” on page 113)

For each data type, the following table shows the precedence list (in order) that DB2 uses to determine the data types to which the data type can be promoted. The table indicates that the best choice is the same data type and not promotion to another data type. The table also shows data types that are considered equivalent during the promotion process. For example, CHARACTER and GRAPHIC are considered to be equivalent data types.

Table 14. Precedence of data types

Data type ^{1,2}	Data type precedence list (in best-to-worst order)
SMALLINT ³	SMALLINT, INTEGER, BIGINT, decimal, real, double, DECFLOAT
INTEGER ³	INTEGER, BIGINT, decimal, real, double, DECFLOAT
BIGINT ³	BIGINT, decimal, real, double, DECFLOAT
decimal ³	decimal, real, double, DECFLOAT
real	real, double, DECFLOAT
double	double, DECFLOAT
DECFLOAT	DECFLOAT
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB or DBCLOB

Table 14. Precedence of data types (continued)

Data type ^{1,2}	Data type precedence list (in best-to-worst order)
CLOB or DBCLOB	CLOB or DBCLOB
BINARY	BINARY, VARBINARY, BLOB
VARBINARY	VARBINARY, BLOB
BLOB	BLOB
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
ROWID	ROWID
XML	XML
A distinct type	The same distinct type

Notes:

- The data types in lowercase letters represent the following data types:
 - decimal** DECIMAL(*p,s*) or NUMERIC(*p,s*)
 - real** REAL or FLOAT(*n*) where *n* is not greater than 21
 - double** DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is greater than 21
- Other synonyms for the listed data types are considered to be the same as the synonym listed.
- Real and double are checked for function resolution purposes only. Additionally, the number of significant digits (even for DECFLOAT(16)), and the exponent range of DECFLOAT exceeds that of real and double (double has 16 significant digits). Therefore, DECFLOAT values will not be promoted to real or double.

Casting between data types

There are many occasions when a value with a given data type needs to be *cast* (changed) to a different data type or to the same data type with a different length, precision, or scale.

Data type promotion is one example where the promotion of one data type to another data type requires that the value be cast to the new data type. A data type that can be changed to another data type is *castable* from the source data type to the target data type.

The casting of one data type to another can occur implicitly or explicitly. You can use the function notation syntax or CAST specification syntax to explicitly cast a data type. DB2 might implicitly cast data types during assignments that involve a distinct type (see “Distinct type assignments” on page 113). In addition, when you create a sourced user-defined function, the data types of the parameters of the source function must be castable to the data types of the function that you are creating (see “CREATE FUNCTION” on page 915).

If truncation occurs when a character or graphic string is cast to another data type, a warning occurs if any non-blank characters are truncated. This truncation behavior is similar to retrieval assignment of character or graphic strings. See “Retrieval assignment” on page 110.

If truncation occurs when casting to a binary string, an error is returned.

For casts that involve a distinct type as either the data type to be cast to or from, Table 15 shows the supported casts.

For casting a parameter marker or NULL value to the XML data type, the CAST specification can be used. XML input can also be specified for the CAST specification when the result data type is XML.

Table 15. Supported casts when a distinct type is involved

Data type ...	Is castable to data type ...
Distinct type <i>DT</i>	Source data type of distinct type <i>DT</i>
Source data type of distinct type <i>DT</i>	Distinct type <i>DT</i>
Distinct type <i>DT</i>	Distinct type <i>DT</i>
Data type <i>A</i>	Distinct type <i>DT</i> where <i>A</i> is promotable to the source data type of distinct type <i>DT</i> (see “Promotion of data types” on page 95)
INTEGER	Distinct type <i>DT</i> if <i>DT</i> 's source data type is SMALLINT
DOUBLE	Distinct type <i>DT</i> if <i>DT</i> 's source data type is REAL
VARCHAR	Distinct type <i>DT</i> if <i>DT</i> 's source data type is CHAR or GRAPHIC
VARGRAPHIC	Distinct type <i>DT</i> if <i>DT</i> 's source data type is GRAPHIC or CHAR
VARBINARY	Distinct type <i>DT</i> if <i>DT</i> 's source data type is BINARY

When a distinct type is involved in a cast, a cast function that was generated when the distinct type was created is used. How DB2 chooses the function depends on whether function notation or CAST specification syntax is used. (For details, see “Function resolution” on page 175 and “CAST specification” on page 202, respectively.) Function resolution is similar for both. However, in CAST specification, when an unqualified distinct type is specified as the target data type, DB2 first resolves the schema name of the distinct type and then uses that schema name to locate the cast function.

For casts between built-in data types, the following table shows the supported casts.

Table 16. Supported casts between built-in data types

	To data type ¹															
	S	M	I	D	E		V	G	V						T	
	A	N	B	E	C	D	A	R	A	R	D	B	B		I	
	L	T	I	C	F	O	R	A	A	B	I	I			E	
	L	E	G	I	L	R	U	C	C	P	P	C	N	N	S	R
	I	G	I	M	O	E	B	H	H	L	H	L	A	A	T	O
Cast from data type –	N	E	N	A	A	A	L	A	A	O	I	I	O	R	A	W
	T	R	T	L	T	L	E	R	R	B	C	C	B	Y	P	X
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	Y							
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	Y							
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	Y							
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	Y							

Table 16. Supported casts between built-in data types (continued)

Cast from data type –	To data type ¹																			
	S	M	I	D	E	D	V	G	R	G	D	B	R	A	L	D	T	S	R	O
	A	N	B	E	C	O	A	R	R	A	B	I	N	A	L	A	I	A	W	X
	L	T	I	C	F	R	C	H	L	H	L	N	A	R	O	T	M	P	D	L
	I	G	I	M	O	E	B	H	C	P	C	A	R	Y	Y	B	E	E		
	N	E	N	A	A	A	L	A	A	O	I	I	O	C	B					
	T	R	T	L	T	L	E	R	R	B	C	C	B	Y	Y	B	E	E	P	D
DECFLOAT	Y	Y	Y	Y	Y	Y	Y	Y	Y											
REAL	Y	Y	Y	Y	Y	Y	Y	Y	Y											
DOUBLE	Y	Y	Y	Y	Y	Y	Y	Y	Y											
CHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
VARCHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CLOB							Y	Y	Y	Y	Y	Y	Y	Y	Y					
GRAPHIC	Y	Y	Y	Y	Y	Y	Y	Y ²	Y ²	Y ²	Y	Y	Y	Y	Y	Y	Y ³	Y ³	Y ³	
VARGRAPHIC	Y	Y	Y	Y	Y	Y	Y	Y ²	Y ²	Y ²	Y	Y	Y	Y	Y	Y	Y	Y	Y	
DBCLOB							Y ²	Y ²	Y ²	Y	Y	Y	Y	Y	Y					
BINARY															Y	Y	Y			
VARBINARY															Y	Y	Y			
BLOB															Y	Y	Y			
DATE							Y	Y									Y			
TIME							Y	Y									Y			
TIMESTAMP							Y	Y									Y	Y	Y	
ROWID							Y	Y						Y	Y	Y				Y
XML																				Y

Note:

1. Other synonyms for the listed data types are considered to be the same as the synonym listed. Some exceptions exist when the cast involves character string data if the subtype is FOR BIT DATA.
2. The result length for these casts is 3 * LENGTH(graphic string).
3. These data types are castable between each other only if the data is Unicode.

Table 17 shows where to find information about the rules that apply when casting to the identified target data types.

Table 17. Rules for casting to a data type

Target data type	Rules
SMALLINT	"SMALLINT" on page 505
INTEGER	"INTEGER or INT" on page 392
BIGINT	"BIGINT" on page 295
DECIMAL	"DECIMAL or DEC" on page 344
NUMERIC	"DECIMAL or DEC" on page 344

Table 17. Rules for casting to a data type (continued)

Target data type	Rules
REAL	"REAL" on page 478
DOUBLE	"DOUBLE_PRECISION or DOUBLE" on page 353
DECFLOAT	"DECFLOAT" on page 340
CHAR	"CHAR" on page 302
VARCHAR	"VARCHAR" on page 558
CLOB	"CLOB" on page 312
GRAPHIC	"GRAPHIC" on page 376
VARGRAPHIC	"VARGRAPHIC" on page 569
DBCLOB	"DBCLOB" on page 336
BINARY	"BINARY" on page 297
VARBINARY	"VARBINARY" on page 556
BLOB	"BLOB" on page 299
DATE	"DATE" on page 328
TIME	"TIME" on page 524
TIMESTAMP	See "TIMESTAMP" on page 525, where one operand is specified.
ROWID	"ROWID" on page 493

Casting non-XML values to XML values

Table 18. Supported Casts from Non-XML Values to XML Values

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
SMALLINT	Y	xs:short
INTEGER	Y	xs:int
BIGINT	Y	xs:long
DECIMAL	Y	xs:decimal
DECFLOAT	N	
REAL	N	
FLOAT	Y	xs:double
DOUBLE	Y	xs:double
CHAR	Y	xs:string
VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string
DBCLOB	Y	xs:string
BINARY	N	
VARBINARY	N	
BLOB	N	
character type FOR BIT DATA	N	

Table 18. Supported Casts from Non-XML Values to XML Values (continued)

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
DATE	N	
TIME	N	
TIMESTAMP	N	
ROWID	N	
distinct type	N	

When character string values are cast to XML values, the resulting xs:string atomic value cannot contain illegal XML characters. If the input character string is not in Unicode, the input characters are converted to Unicode.

Casting XML values to non-XML values

An XMLCAST from an XML value to a non-XML value can be described as two casts: an XQuery cast that converts the source XML value to a target XQuery data type that corresponds to the SQL target type, followed by a cast from the corresponding XQuery data type to the actual SQL type. The target XQuery data type is an XML schema data type like xs:decimal or xs:string, as shown in the follow table.

An XMLCAST is supported if the target type has a corresponding XQuery target type that is supported, and if there is a supported XQuery cast from the type of the source value to the corresponding XQuery target type. The target type that is used in the XQuery cast is based on the corresponding XQuery target type and might contain some additional restrictions.

The following table lists the XQuery types that result from such conversion.

Table 19. Supported Casts from XML Values to Non-XML Values

Target Data Type	Source Data Type	
	XML	Corresponding XQuery Target Type
SMALLINT	Y	xs:integer
INTEGER	Y	xs:integer
BIGINT	Y	xs:integer
DECIMAL	Y	xs:decimal
DECFLOAT	Y	xs:double
REAL	Y	xs:double
FLOAT	Y	xs:double
DOUBLE	Y	xs:double
CHAR	Y	xs:string
VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string
DBCLOB	Y	xs:string

Table 19. Supported Casts from XML Values to Non-XML Values (continued)

Target Data Type	Source Data Type	
	XML	Corresponding XQuery Target Type
BINARY	N	
VARBINARY	N	
BLOB	N	
character type FOR BIT DATA	N	
DATE	Y	xs:date
TIME	Y	xs:time
TIMESTAMP	Y	xs:dateTime
ROWID	N	
distinct type	N	

The following restrictions are in effect when a value is cast from an XQuery target data type to a target SQL data type:

- If the target type is one of the character or graphic string types, the resulting XML value is converted, if necessary, to the CCSID of the target data type using the rules described in “Conversion rules for string assignment” on page 111, before it is converted to the target type with a limited length. Truncation occurs if the specified length limit is smaller than the length of the resulting string after CCSID conversion. A warning occurs if any non-blank characters are truncated. If the target type is a fixed-length string type (CHAR or GRAPHIC) and the specified length of the target type is greater than the length of the resulting string from CCSID conversion, blanks are padded at the end. This truncation and padding behavior is similar to retrieval assignment of character or graphic strings.
- If the target type is DOUBLE or REAL and the source XML value after the XQuery cast is an xs:double value of INF, -INF, or NaN, an error is returned. If the source value is an xs:double negative zero, the value is converted to positive zero. If the source value is beyond the range of the target data type, an overflow error is returned. If the source value contains more significant digits than the precision of the target data type, the source value is rounded to the precision of the target data type.
- If the target type is DECFLOAT and the source XML value is an xs:double value of INF, -INF, or NaN, the result will be the corresponding special DECFLOAT values INF, -INF, or NaN. If the source value is an xs:double negative zero, the result is negative zero. If the target type is DECFLOAT(16) and the source value is beyond the range of DECFLOAT(16), an overflow error is returned. If the source value has more than 16 significant digits, the value is rounded according to the ROUNDING mode that is in effect. This rounding behavior is the same as what is used during the cast of DECFLOAT(34) to DECFLOAT(16).
- If the target type is DECIMAL, the resulting xs:decimal value is converted, if necessary, to the precision and scale of the target data type. The necessary number of leading zeros is added or removed. In the fractional part of the number, the necessary number of trailing zeros is added or the necessary number of digits is eliminated. This truncation behavior is similar to the behavior of the cast from DECIMAL to DECIMAL.
- If the target type is DATE, TIME, or TIMESTAMP, the resulting XML value is adjusted to UTC time and the time zone component is removed. If the target type is TIME and the resulting XML value contains a seconds component with

non-zero digits after the decimal point, those digits are truncated. If the target type is DATE or TIMESTAMP, the year part of the resulting xs:date or xs:dateTime value must be in the range of 0001 to 9999. If the target type is TIMESTAMP, the fractional second part of the xs:dateTime value must have six or fewer digits.

Assignment and comparison

The basic operations of SQL are assignment and comparison.

Assignment operations are performed during the execution of statements such as CALL, INSERT, UPDATE, MERGE FETCH, SELECT INTO, SET *host-variable* or SET *assignment-statement*, and VALUES INTO statements. In addition, when a function is invoked or a stored procedure is called, the arguments of the function or stored procedure are assigned. Comparison operations are performed during the execution of statements that include predicates and other language elements such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

The basic rule for both operations is that data types of the operands must be compatible. The compatibility rule also applies to other operations that are described under “Rules for result data types” on page 119. The following table shows the compatibility of data types for assignments and comparisons.

Table 20. Compatibility of data types for assignments and comparisons with numeric data types. Y indicates that the data types are compatible. N indicates no compatibility. For any number in a column, read the corresponding note at the bottom of the table.

Operand	Binary integer	Decimal number	Floating point	Decimal floating point
Binary integer	Y	Y	Y	Y
Decimal number	Y	Y	Y	Y
Floating point	Y	Y	Y	Y
Decimal floating point	Y	Y	Y	Y
Character string	N	N	N	N
Graphic string	N	N	N	N
Binary string	N	N	N	N
Date	N	N	N	N
Time	N	N	N	N
Timestamp	N	N	N	N
Row ID	N	N	N	N
Distinct type	1	1	1	1
XML ²	N	N	N	N

Notes:

1. A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, DB2 supports assignments between a distinct type value and its source data type. For additional information, see “Distinct type assignments” on page 113.
2. For comparison, XML can only be compared using the XMLEXISTS and NULL predicates.

Table 21. Compatibility of data types for assignments and comparisons with string data types. Y indicates that the data types are compatible. N indicates no compatibility. For any number in a column, read the corresponding note at the bottom of the table.

Operand	Character string	Graphic string	Binary string
Binary integer	N	N	N
Decimal number	N	N	N
Floating point	N	N	N
Decimal floating point	N	N	N
Character string	Y	Y ^{4,5}	N ³
Graphic string	Y ^{4,5}	Y	N
Binary string	N ³	N	Y
Date	1	1,4	N
Time	1	1,4	N
Timestamp	1	1,4	N
Row ID	N	N	N
Distinct type	2	2	2
XML ⁶	N	N	N

Notes:

- The compatibility of datetime values is limited to assignment and comparison:
 - Datetime values can be assigned to string columns and to string variables that are not LOB values, as explained in “Datetime assignments” on page 112.
 - A valid string representation of a date can be assigned to a date column or compared to a date.
 - A valid string representation of a time can be assigned to a time column or compared to a time.
 - A valid string representation of a timestamp can be assigned to a timestamp column or compared to a timestamp.
- A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, DB2 supports assignments between a distinct type value and its source data type. For additional information, see “Distinct type assignments” on page 113.
- All character strings, even those with subtype FOR BIT DATA, are not compatible with binary strings.
- On assignment and comparison from Graphic to Character, the resulting length in bytes is 3 * (LENGTH(*graphic-string*)), depending on the CCSIDs.
- Character strings with subtype FOR BIT DATA are not compatible with Graphic Data.
- Character and graphic strings, including LOBs, can be assigned to XML columns. However, XML cannot be assigned to a character or graphic string column. For comparison, XML can only be compared using the XMLEXISTS and NULL predicates.

Table 22. Compatibility of data types for assignments and comparisons with date, time, and timestamp data types. Y indicates that the data types are compatible. N indicates no compatibility. For any number in a column, read the corresponding note at the bottom of the table.

Operand	Date	Time	Timestamp
Binary integer	N	N	N
Decimal number	N	N	N
Floating point	N	N	N
Decimal floating point	N	N	N
Character string	1	1	1
Graphic string	1,3	1,3	1,3
Binary string	N	N	N

Table 22. Compatibility of data types for assignments and comparisons with date, time, and timestamp data types (continued). Y indicates that the data types are compatible. N indicates no compatibility. For any number in a column, read the corresponding note at the bottom of the table.

Operand	Date	Time	Timestamp
Date	Y	N	N
Time	N	Y	N
Timestamp	N	N	Y
Row ID	N	N	N
Distinct type	2	2	2
XML ⁴	N	N	N

Notes:

- The compatibility of datetime values is limited to assignment and comparison:
 - Datetime values can be assigned to string columns and to string variables that are not LOB values, as explained in “Datetime assignments” on page 112.
 - A valid string representation of a date can be assigned to a date column or compared to a date.
 - A valid string representation of a time can be assigned to a time column or compared to a time.
 - A valid string representation of a timestamp can be assigned to a timestamp column or compared to a timestamp.
- A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, DB2 supports assignments between a distinct type value and its source data type. For additional information, see “Distinct type assignments” on page 113.
- On assignment and comparison from Graphic to Character, the resulting length in bytes is 3 * (LENGTH(*graphic-string*)), depending on the CCSIDs.
- For comparison, XML can only be compared using the XMLEXISTS and NULL predicates.

Table 23. Compatibility of data types for assignments and comparisons with row ID, XML, and distinct data types. Y indicates that the data types are compatible. N indicates no compatibility. For any number in a column, read the corresponding note at the bottom of the table.

Operand	Row ID	XML ²	Distinct type
Binary integer	N	N	¹
Decimal number	N	N	¹
Floating point	N	N	¹
Decimal floating point	N	N	¹
Character string	N	N	¹
Graphic string	N	N	¹
Binary string	N	N	¹
Date	N	N	¹
Time	N	N	¹
Timestamp	N	N	¹
Row ID	Y	N	¹
Distinct type	¹	N	Y ¹
XML ²	N	Y	N

Notes:

- A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, DB2 supports assignments between a distinct type value and its source data type. For additional information, see “Distinct type assignments” on page 113.
- For comparison, XML can only be compared using the XMLEXISTS and NULL predicates.

Compatibility with a column that has a field procedure is determined by the data type of the column, which applies to the decoded form of its values.

A basic rule for assignment operations is that a null value cannot be assigned to:

- A column that cannot contain null values
- A non-Java host variable that does not have an associated indicator variable
For a host variable that does have an associated indicator variable, a null value is assigned by setting the indicator variable to a negative value. See “References to host variables” on page 160 for a discussion of indicator variables.
- A Java host variable that is a primitive type
For a Java host variable that is not a primitive type, the value of that variable is set to a Java null value.

Numeric assignments

The basic rule for numeric assignments is that the whole part of a decimal or integer number cannot be truncated. If necessary, the fractional part of a decimal number is truncated.

Decimal or integer to floating-point

Because floating-point numbers are only approximations of real numbers, the result of assigning a decimal or integer number to a floating-point column or variable might not be identical to the original number.

Floating-point or decimal to integer

When a single precision floating-point number is converted to integer, rounding occurs on the seventh significant digit, zeros are added to the end of the number, if necessary, starting from the seventh significant digit, and the fractional part of the number is eliminated. When a double precision floating-point or decimal number is converted to integer, the fractional part of the number is eliminated.

Example 1: The following example shows single precision floating-point numbers converted to an integer:

Floating-point number:	Results when assigned to an integer column or host variable:
2.0000045E6	2000000
2.00000555E8	200001000

Example 2: The following example shows a double precision floating-point number converted to an integer:

Floating-point number:	Results when assigned to an integer column or host variable:
2.0000045E6	2000004
2.00000555E8	200000555

Example 3: The following example shows a decimal number converted to an integer:

Decimal number:	Results when assigned to an integer column or host variable:
2000004.5	2000004
200000555.0	200000555

Decimal to decimal

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and the scale of the target.

The necessary number of leading zeros is added or eliminated, and, in the fractional part of the number, the necessary number of trailing zeros is added, or the necessary number of trailing digits is eliminated.

Decimal to DECFLOAT

When a decimal number is assigned to a DECFLOAT column or variable, the number is converted to the precision (16 or 34) of the target. Leading zeros are eliminated.

Depending on the precision and scale of the decimal number, and the precision of the target, the value might be rounded to fit.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

Integer to decimal

When an integer is assigned to a decimal column or variable, the number is converted first to a temporary decimal number and then, if necessary, to the precision and scale of the target.

The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer.

Integer to DECFLOAT

When an integer is assigned to a DECFLOAT column or variable, the number is converted first to a temporary decimal number and then to DECFLOAT.

The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer. The decimal number is then converted to DECFLOAT using the rules for "Decimal to DECFLOAT."

Floating-point to floating-point

When a single precision floating-point number is assigned to a double precision floating-point column or variable, the single precision data is padded with eight hex zeros. When a double precision floating-point number is assigned to a single precision floating-point column or variable, the double precision data is converted and rounded up on the seventh hex digit.

In assembler, C, or C++ applications that are precompiled with the FLOAT(IEEE) option, floating-point constants and values in host variables are assumed to have IEEE floating-point format. All floating-point data is stored in DB2 in System/390 floating-point format. Therefore, when the FLOAT(IEEE) precompiler option is in effect, DB2 performs the following conversions:

- When a number in short or long IEEE floating-point format is assigned to a single-precision or double-precision floating-point column, DB2 converts the number to System/390 floating-point format.
- When a single-precision or double-precision floating-point column value is assigned to a short or long floating-point host variable, DB2 converts the column value to IEEE floating-point format.

Floating-point to decimal

When a single precision floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number.

When a single precision floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 6 by rounding on the seventh decimal digit. Twenty five zeros are then appended to the number to bring the precision to 31. Because of rounding, a number less than 0.5×10^{-6} is reduced to 0.

When a double precision floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 15, and then, if necessary, truncated to the precision and scale of the target. In this conversion, zeros are added to the end of the number, if necessary, to bring the precision to 16. The number is then rounded (using floating-point arithmetic) on the sixteenth decimal digit to produce a 15-digit number. Because of rounding, a number less in magnitude than 0.5×10^{-15} is reduced to 0. If the decimal number requires more than 15 digits to the left of the decimal point, an error is reported. Otherwise, the scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

The following examples show the effect of converting a double precision floating-point number to decimal:

Example 1: The floating-point number, .123456789098765E-05 in decimal notation is, .00000123456789098765. Rounding adds 5 in the 16th position, so the number becomes .00000123456789148765 and truncates the result to .000001234567891. Zeros are then added to the end of a 31-digit result, and the number becomes .00000123456789100000000000000000.

Example 2: The floating-point number, 1.2339999999999E+01 in decimal notation is, 12.33999999999900. Rounding adds 5 in the 16th position, so the number becomes 12.33999999999905 and truncates the result to 12.3399999999990. Zeros are then added to the end of a 31-digit result and the number becomes 12.33999999999900000000000000000000.

Floating point to DECFLOAT

When a single or double precision floating-point number is assigned to a DECFLOAT column or variable, the number is first converted to a temporary string representation of the floating point number. The string representation of the number is then converted to DECFLOAT.

DECFLOAT to integer

When a DECFLOAT is assigned to a binary integer column or variable, the fractional part of the number is lost.

Example 1: The following example shows decimal floating-point numbers converted to an integer:

Decimal floating-point number:	Results when assigned to an integer column or host variable:
2.0000045E6	2000004
2.00000555E8	200000555

DECFLOAT to decimal

When a DECFLOAT value is assigned to a decimal column or variable, the DECFLOAT value is converted, if necessary, to the precision and the scale of the target.

During the assignment, the necessary number of leading zeros is added and, in the fractional part of the number, the necessary number of trailing zeros is added, or rounding occurs.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

Example 1: The following example shows decimal floating-point numbers converted to a decimal value:

Decimal floating-point number:	Results when assigned to an decimal(15,0) column or host variable:
2.0000045E6	2000005
Decimal floating-point number:	Results when assigned to an decimal(15,2) column or host variable:
2.0000045E6	2000004.50
2.00000555E8	200000555.00

DECFLOAT to floating-point

Because floating-point numbers are only approximations of real numbers, the result of assigning a DECFLOAT value to a floating-point column or variable might not be identical to the original number.

The DECFLOAT value is first converted to a string representation, and is then converted to floating-point number.

DECFLOAT(16) to DECFLOAT(34)

When a DECFLOAT(16) is assigned to a DECFLOAT(34) column or variable, the exponent of the source is converted to the corresponding exponent in the result format, and the coefficient is extended by appending zeros on the left.

DECFLOAT(34) to DECFLOAT(16)

When a DECFLOAT(34) is assigned to a DECFLOAT(16) column or variable, the exponent of the source is converted to the corresponding exponent in the result format.

The source coefficient is rounded to the precision of the target.

For static SQL statements, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

To COBOL integers

Assignment to COBOL integer variables uses the full size of the integer.

Thus, the value placed in the COBOL data item might be out of the range of values.

Example 1: If COL1 contains a value of 12345, the following statements cause the value 12345 to be placed in A, even though A has been defined with only 4 digits:

```
01 A PIC S9999 BINARY.  
EXEC SQL SELECT COL1  
      INTO :A  
      FROM TABLEX  
END-EXEC.
```

Example 2: The following COBOL statement results in 2345 being placed in A:

```
MOVE 12345 TO A.
```

String assignments

There are two types of string assignments; storage assignment and retrieval assignment.

- *Storage assignment* is when a value is assigned to a column or a parameter of a function or stored procedure.
- *Retrieval assignment* is when a value is assigned to a host variable.

The rules differ for storage and retrieval assignment.

Binary string assignment

Binary string assignment involves assignment at both the storage and the retrieval of binary strings.

Storage assignment: The length of a string that is assigned to a column or parameter of a function or procedure must not be greater than the length attribute of the column or parameter. If the string is longer than the length attribute of that column or parameter, an error is returned.

When the string is assigned to a fixed-length binary string column or parameter of a function or procedure, and the length of the string is less than the length attribute of that column or parameter, the string is padded to the right with the necessary number of binary zeros.

Retrieval assignment: The length of a string that is assigned to a host variable can be greater than the length attribute of the host variable. When a string is assigned to a host variable and the string is longer than the length attribute of the host variable, the string is truncated on the right by the necessary number of bytes. When this occurs, a warning is returned.

Character and graphic string assignment

The rules for storage and retrieval assignment apply when both the source and the target are strings.

When a datetime data type is involved, see “Datetime assignments” on page 112. For the special considerations that apply when a distinct type is involved in an assignment, especially to a host variable, see “Distinct type assignments” on page 113.

Storage assignment:

The basic rule for character storage assignment is that the length of a string that is assigned to a column or parameter of a function or stored procedure must not be greater than the length attribute of the column or the parameter.

Trailing blanks are included in the length of the string. When the length of the string is greater than the length attribute of the column or the parameter, the following actions occur:

- If all of the trailing characters that must be truncated to make a string fit the target are blanks and the string is a character or graphic string, the string is truncated and assigned without warning.
- Otherwise, the string is not assigned and an error occurs to indicate that at least one of the excess characters is non-blank.

When a string is assigned to a fixed-length column or parameter and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of SBCS or DBCS blanks. The pad character is always a blank even for columns or parameters that are defined with the FOR BIT DATA attribute.

Retrieval assignment:

The length of a string that is assigned to a host variable can be greater than the length attribute of the host variable. When the length of the string is greater than the length of the host variable, the string is truncated on the right by the necessary number of SBCS or DBCS characters.

When truncation occurs, the value W is assigned to the SQLWARN1 field of the SQLCA. Furthermore, if an indicator variable is provided and the source of the value is not a LOB, the indicator variable is set to the original length of the string. The truncation result of an improperly formed mixed string is unpredictable.

When a character string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of blanks. The pad character is always a blank even for strings defined with the FOR BIT DATA attribute.

When a string of length n is assigned to a varying-length string variable with a maximum length greater than n , the characters after the n th character of the variable are undefined.

Assignments involving mixed data strings

A mixed data string that contains MBCS characters cannot be assigned to an SBCS column, SBCS parameter, or SBCS host variable.

The following rules apply when a mixed data string is assigned to a host variable and the string is longer than the length attribute of the variable:

- If the string is not well-formed mixed data, it is truncated as if it were BIT or graphic data.
- If the string is well-formed mixed data, it is truncated on the right such that it is well-formed mixed data with a length that is the same as the length attribute of the variable and the number of characters lost is minimal.

Assignments involving C NUL-terminated strings

A C NUL-terminated string variable that is referenced in a CONNECT statement does not need to contain a NUL. Otherwise, DB2 enforces the convention that the value of a NUL-terminated string variable, either character or graphic, is NUL-terminated.

An input host variable that does not contain a NUL will cause an error. A value that is assigned to an output variable will always be NUL-terminated even if a character must be truncated to make room for the NUL.

When a string of length n is assigned to a C NUL-terminated string variable with a length greater than $n+1$, the rules depend on whether the source string is a value of a fixed-length string or a varying-length string:

- If the source is a fixed-length string and the value of field PAD NUL-TERMINATED on installation panel DSNTIP4 is YES, the string is padded on the right with $x-n-1$ blanks, where x is the length of the variable. The padded string is then assigned to the variable and a NUL is appended at the end of the variable. If the value of field PAD NUL-TERMINATED is NO, the string is assigned to the first n bytes of the variable and a NUL is appended at the end of the variable.
- If the source is a varying-length string, the string is assigned to the first n bytes of the variable and a NUL is appended at the end of the variable.

Conversion rules for string assignment

A character or graphic string that is assigned to a column or host variable is first converted, if necessary, to the coded character set of the target. Conversion is necessary only if certain conditions apply.

Conversion is necessary only if all the following are true:

- The CCSIDs of string and target are different.
- Neither CCSID is X'FFFF' (neither the string nor the target is defined as BIT data).
- The string is neither null nor empty.

An error occurs if:

- The SYSSTRINGS table is used but contains no information about the pair of CCSIDs and DB2 cannot do the conversion through z/OS support for Unicode.
- A character of the string cannot be converted and the operation is assignment to a column or to a host variable that has no indicator variable. For example, a DBCS character cannot be converted to a host variable with an SBCS CCSID.

A warning occurs if:

- A character of the string is converted to a *substitution character*. A *substitution character* is the character that is used when a character of the source character set is not part of the target character set. For example, assuming an EBCDIC target character set, if the source character set includes Katakana characters and the target character set does not, a Katakana character is converted to the EBCDIC SUB X'3F'.
- A character of the string cannot be converted and the operation is assignment to a host variable that has an indicator variable. For example, a DBCS character cannot be converted if the host variable has an SBCS CCSID. In this case, the string is not assigned to the host variable and the indicator variable is set to -2.

Datetime assignments

A value that is assigned to a date, time, or timestamp column, variable, or parameter must be a valid string representation of a date, a time, or a timestamp.

A value that is assigned to a DATE column, a DATE variable, or a DATE must be a date or a valid string representation of a date. A date can be column, a character-string column, or a character-string variable.

A value that is assigned to a TIME column, a TIME variable, or a TIME parameter must be a time or a valid string representation of a time. A time can be assigned only to a TIME column, a character-string column, or a character-string variable.

A value assigned to a TIMESTAMP column, a TIMESTAMP variable, or a TIMESTAMP parameter must be a timestamp or a valid string representation of a timestamp. A timestamp can be assigned only to a TIMESTAMP column, a character-string column, or a character-string variable.

A valid string representation of a datetime value must not be a BLOB, CLOB, or DBCLOB. A datetime value cannot be assigned to a column that has a field procedure.

When a datetime value is assigned to a character-string variable or column, it is converted to its string representation. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target varies depending on the format of the string representation. If the length of the fixed length character-string target is greater than required, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of datetime value involved, and the type of the target.

- If the target is a string column (except for BLOB, CLOB, or DBCLOB), truncation is not allowed. The length of the column must be at least 10 for a date, 8 for a time, and 19 for a timestamp.
- When the target is a host variable, the following rules apply:
 - **For a DATE:** The length of the variable must not be less than 10.
 - **For a TIME:** If the USA format is used, the length of the variable must not be less than 8. This format does not include seconds.
If the ISO, EUR, or JIS format is used, the length of the variable must not be less than 5. If the length is 5, 6, or 7, the seconds part of the time is omitted from the result and SQLWARN1 is set to 'W'. In this case, the seconds part of the time is assigned to the indicator variable if one is provided, and, if the length is 6 or 7, the value is padded with blanks so that it is a valid string representation of a time.
 - **For a TIMESTAMP:** The length of the variable must not be less than 19. If the length is between 19 and 25, the timestamp is truncated like a string, causing the omission of one or more digits of the microsecond part. If the length is 20, the trailing decimal point is replaced by a blank so that the value is a valid string representation of a timestamp.

Row ID assignments

A row ID value can be assigned only to a column, parameter, or host variable with a row ID data type.

For the value of the ROWID column, the column must be defined as GENERATED BY DEFAULT and the column must have a unique, single-column index. The value that is specified for the column must be a valid row ID value that was previously generated by DB2.

XML assignments

XML data can be assigned to a column, but when the target is not a column, the XML data type can only be assigned to another XML data type.

When the target is a column (for example, data change statements), the source can be the XML data type, or CHAR, VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, DBCLOB, BINARY, VARBINARY, or BLOB data types. When the source is not XML data, the source is implicitly parsed as if the XMLPARSE function is invoked with the STRIP WHITESPACE option. If the source data is graphic data, the encoding scheme must be Unicode.

All other data types cannot be assigned to a target of the XML data type.

Distinct type assignments

The rules that apply to the assignments of distinct types to host variables are different than the rules for all other assignments that involve distinct types.

Assignments to host variables: The assignment of distinct type to a host variable is based on the source data type of the distinct type. Therefore, the value of a distinct type is assignable to a host variable only if the source data type of the distinct type is assignable to the host variable.

Example: Assume that distinct type AGE was created with the following SQL statement:

```
CREATE TYPE AGE AS SMALLINT;
```

When the statement was executed, DB2 also generated these cast functions:

```
AGE (SMALLINT) RETURNS AGE
AGE (INTEGER) RETURNS AGE
SMALLINT (AGE) RETURNS SMALLINT
```

Next, assume that column STU_AGE was defined in table STUDENTS with distinct type AGE. Now, consider this valid assignment of a student's age to host variable HV_AGE, which has an INTEGER data type:

```
SELECT STU_AGE INTO :HV_AGE FROM STUDENTS WHERE STU_NUMBER = 200;
```

The distinct type value is assignable to host variable HV_AGE because the source data type of the distinct type (SMALLINT) is assignable to the host variable (INTEGER). If distinct type AGE had been based on a character data type such as CHAR(5), the above assignment would be invalid because a character type cannot be assigned to an integer type.

Assignments other than to host variables: A distinct type can be the source or target of an assignment. Assignment is based on whether the data type of the value to be assigned is castable to the data type of the target. (Table 15 on page 97 shows which casts are supported when a distinct type is involved). Therefore, a distinct type value can be assigned to any target other than a host variable when:

- The target of the assignment has the same distinct type, or
- The distinct type is castable to the data type of the target

Any value can be assigned to a distinct type when:

- The value to be assigned has the same distinct type as the target, or
- The data type of the assigned value is castable to the target distinct type

Example: Assume that the source data type for distinct type AGE is SMALLINT:

```
CREATE TYPE AGE AS SMALLINT;
```

Next, assume that two tables TABLE1 and TABLE2 were created with four identical column descriptions:

```
AGECOL    AGE
SMINTCOL  SMALLINT
INTCOL    INTEGER
DECCOL    DEC(6,2)
```

Using the following SQL statement and substituting various values for X and Y to insert values into various columns of TABLE1 from TABLE2, the following table shows whether the assignments are valid. DB2 uses assignment rules in this INSERT statement to determine if X can be assigned to Y.

```
INSERT INTO TABLE1 (Y)
  SELECT X FROM TABLE2;
```

Table 24. Assessment of various assignments for example INSERT statement

X (column in TABLE2)	Y (column in TABLE1)	Valid	Reason
AGECOL	AGECOL	Yes	Source and target are same distinct type
SMINTCOL	AGECOL	Yes	SMALLINT can be cast to AGE
INTCOL	AGECOL	Yes	INTEGER can be cast to AGE (because AGE's source type is SMALLINT)
DECCOL	AGECOL	No	DECIMAL cannot be cast to AGE
AGECOL	SMINTCOL	Yes	AGE can be cast to its source type of SMALLINT
AGECOL	INTCOL	No	AGE cannot be cast to INTEGER
AGECOL	DECCOL	No	AGE cannot be cast to DECIMAL

Assignments to LOB locators

When a LOB locator is used, it can refer only to LOB data. If a LOB locator is used for the first fetch of a cursor, LOB locators must be used for all subsequent fetches.

Numeric comparisons

Numbers are compared algebraically, that is, with regard to sign. For example, -2 is less than +1. When numbers of different data types are compared, certain rules are in effect as to how the comparison is performed.

If one number is an integer and the other is decimal, the comparison is made with a temporary copy of the integer, which has been converted to decimal.

When decimal numbers with different scales are compared, the comparison is made with a temporary copy of one of the numbers that has been extended with trailing zeros so that its fractional part has the same number of digits as the other number.

If one number is double precision floating-point and the other is integer, decimal, or single precision floating-point, the comparison is made with a temporary copy of the other number which has been converted to double precision floating-point. However, if a single precision floating-point number is compared with a floating-point constant, the comparison is made with a single precision form of the constant.

Two floating-point numbers are equal only if the bit configurations of their normalized forms are identical.

| If one number is DECFLOAT and the other number is integer, decimal, single
| precision floating-point, or double precision floating-point, the comparison is made
| with a temporary copy of the other number which has been converted to
| DECFLOAT.

| If one number is DECFLOAT(16) and the other number is DECFLOAT(34), the
| DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison.

| Additionally, the DECFLOAT data type supports both positive and negative zero.
| Positive and negative zero have different binary representations, but the equal (=)
| predicate will return true for comparisons of positive and negative zero.

| The functions, COMPARE_DECFLOAT and TOTALORDER can be used to perform
| comparisons at a binary level. For example, for a comparison of 2.0<>2.00.

| The DECFLOAT data type also supports the specification of negative and positive
| NaN (quiet and signaling), and negative and positive infinity. From an SQL
| perspective, infinity = infinity, NaN = NaN, and sNaN = sNaN.

| The following rules are the comparison rules for these special values:
| • Infinity compares equal only to infinity of the same sign (positive or negative)
| • NaN compares equal only to NaN of the same sign (positive or negative)
| • sNaN compares equal only to sNaN of the same sign (positive or negative)

| The ordering among the different special values is as follows: -NAN < -SNAN <
| -INFINITY < 0 < INFINITY < SNAN <NAN

String comparisons

String comparisons can occur with binary string, character strings, and graphic strings.

Binary string comparisons

| Binary string comparisons are always performed according to the binary values.

| Two binary strings are equal only if the lengths of the two strings are identical. If
| the strings are equal up to the length of the shorter string length, the shorter string
| is considered less than the longer string even when the remaining bytes in the
| longer string are hexadecimal zeros. This is illustrated in the following table:

| *Table 25. Binary string comparison where one operand is longer because of hexadecimal
| zeros*

Hexadecimal value of the first operand	relationship	Hexadecimal value of the second operand
X'4100'	<	X'410000'

Table 25. Binary string comparison where one operand is longer because of hexadecimal zeros (continued)

Hexadecimal value of the first operand	relationship	Hexadecimal value of the second operand
X'4100'	<	X'42'
X'4100'	=	X'4100'
X'4100'	>	X'41'
X'4100'	>	X'400000'

Binary strings cannot be compared to character strings (even FOR BIT DATA) unless the character string is cast to a binary string.

Character and graphic string comparisons

Two strings are compared by comparing the corresponding bytes of each string. If the strings do not have the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string.

Two strings are equal if they are both empty or if all corresponding bytes are equal. An empty string is equal to a blank string. If two strings are not equal, their relationship (that is, which has the greater value) is determined by the comparison of the first pair of unequal bytes from the left end of the strings. This comparison is made according to the collating sequence associated with the encoding scheme of the data. For ASCII data, characters A through Z (both upper and lowercase) have a greater value than characters 0 through 9. For EBCDIC data, characters A through Z (both upper and lowercase) have a lesser value than characters 0 through 9.

Varying-length strings with different lengths are equal if they differ only in the number of trailing blanks. In operations that select one value from a collection of such values, the value selected is arbitrary. The operations that can involve such an arbitrary selection are DISTINCT, MAX, MIN, and references to a grouping column. See the description of GROUP BY for further information about the arbitrary selection involved in references to a grouping column.

String comparisons with field procedures:

The rules for string comparisons with field procedures depend on the values being compared.

If a column with a field procedure is compared with the value of a variable or a constant, the variable or constant is encoded by the field procedure before the comparison is made. If the comparison operator is LIKE, the variable or constant is not encoded and the column value is decoded.

If a column with a field procedure is compared with another column, that column must have the same field procedure and both columns must have the same CCSID set. The comparison is performed on the encoded form of the values in the columns. If the encoded values are numeric, their data types must be identical; if they are strings, their data types must be compatible.

If two encoded strings of different lengths are compared, the shorter is temporarily padded with encoded blanks so that it has the same length as the other string.

In a CASE expression, if a column with a field procedure is used as the *result-expression* in a THEN or ELSE clause, all other columns that are used as *result-expressions* must have the same field procedure. Otherwise, no column used in a *result-expression* can name a field procedure.

Datetime comparisons

A DATE, TIME, or TIMESTAMP value can be compared either with another value of the same data type or with a string representation of that data type.

All comparisons are chronological, which means the further a point in time is from January 1, 0001, the greater the value of that point in time.

Comparisons that involve TIME values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds are implied.

Comparisons that involve timestamp values are chronological without regard to representations that might be considered equivalent. Thus, the following predicate is true:

```
TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
```

Row ID comparisons

A value with a row ID type can only be compared to another row ID value.

The comparison of the row ID values is based on their internal representations. The maximum number of bytes that are compared is 17 bytes, which is the number of bytes in the internal representation. Therefore, row ID values that differ in bytes beyond the 17th byte are considered to be equal.

XML comparisons

The XML data type cannot be directly compared to any data type, including the XML data type. The method for doing comparison is through the use of the XMLEXISTS predicate.

Distinct type comparisons

A value with a distinct type can only be compared to another value with exactly the same type because distinct types have strong typing, which means that a distinct type is compatible only with its own type.

To compare a distinct type to a value with a different data type, the distinct type value must be cast to the data type of the comparison value or the comparison value must be cast to the distinct type. For example, because constants are built-in data types, a constant can be compared to a distinct type value only if it is first cast to the distinct type or vice versa.

The following table shows examples of valid and invalid comparisons, assuming the following SQL statements were used to define two distinct types AGE_TYPE and CAMP_DATE and table CAMP_ROSTER table.

```
CREATE TYPE AGE_TYPE AS INTEGER;
CREATE TYPE CAMP_DATE AS DATE;
CREATE TABLE CAMP_ROSTER
( NAME                VARCHAR(20),
  ATTENDEE_NUMBER     INTEGER NOT NULL,
```


AGE	AGE_TYPE,
FIRST_CAMP_DATE	CAMP_DATE,
LAST_CAMP_DATE	CAMP_DATE,
BIRTHDATE	DATE);

Table 26. Examples of valid and invalid comparisons involving distinct types

SQL statement	Valid	Reason
Distinct types with distinct types		
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE < LAST_CAMP_DATE;	Yes	Both values are the same distinct type.
Distinct types with columns of the same source data type		
SELECT * FROM CAMP_ROSTER WHERE AGE > ATTENDEE_NUMBER;	No	A distinct type cannot be compared to integer.
SELECT * FROM CAMP_ROSTER WHERE INTEGER(AGE) > ATTENDEE_NUMBER;	Yes	The distinct type is cast to an integer, making the comparison of two integers.
SELECT * FROM CAMP_ROSTER WHERE CAST(AGE AS INTEGER) > ATTENDEE_NUMBER;		
SELECT * FROM CAMP_ROSTER WHERE AGE > AGE_TYPE(ATTENDEE_NUMBER);	Yes	Integer ATTENDEE_NUMBER is cast to the distinct type AGE_TYPE, making both values the same distinct type.
SELECT * FROM CAMP_ROSTER WHERE AGE > CAST(ATTENDEE_NUMBER as AGE_TYPE);		
Distinct types with constants		
SELECT * FROM CAMP_ROSTER WHERE AGE IN (15,16,17);	No	A distinct type cannot be compared to a constant.
SELECT * FROM CAMP_ROSTER WHERE INTEGER(AGE) IN (15,16,17);	Yes	The distinct type is cast to the data type of constants, making all the values in the comparison integers.
SELECT * FROM CAMP_ROSTER WHERE AGE IN (AGE_TYPE(15),AGE_TYPE(16),AGE_TYPE(17));	Yes	Constants are cast to distinct type AGE_TYPE, making all the values in the comparison the same distinct type.
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > '06/12/99';	No	A distinct type cannot be compared to a constant.
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > CAST('06/12/99' AS CAMP_DATE);	No	The string constant '06/12/99', a VARCHAR data type, cannot be cast directly to distinct type CAMP_DATE, which is based on a DATE data type. As illustrated in the next row, the constant must be cast to a DATE data type and then to the distinct type.
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > CAST(DATE('06/12/1999') AS CAMP_DATE);	Yes	The string constant '06/12/99' is cast to the distinct type CAMP_DATE, making both values the same distinct type. To cast a string constant to a distinct type that is based on a DATE, TIME, or TIMESTAMP data type, the string constant must first be cast to a DATE, TIME, or TIMESTAMP data type.
Distinct types with host variables		
SELECT * FROM CAMP_ROSTER WHERE AGE BETWEEN :HV_INTEGER AND :HV_INTEGER2;	No	The host variables have integer data types. A distinct type cannot be compared to an integer.
SELECT * FROM CAMP_ROSTER WHERE AGE BETWEEN CAST(:HV_INTEGER AS AGE_TYPE) AND AGE_TYPE(:HV_INTEGER2);	Yes	The host variables are cast to distinct type AGE_TYPE, making all the values the same distinct type.

Table 26. Examples of valid and invalid comparisons involving distinct types (continued)

SQL statement	Valid	Reason
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > :HV_VARCHAR;	No	The host variable has a VARCHAR data type. A distinct type cannot be compared to a VARCHAR.
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > CAST(DATE(:HV_VARCHAR) AS CAMP_DATE);	Yes	The host variable is cast to the distinct type CAMP_DATE, making both values the same distinct type. To cast a VARCHAR host variable to a distinct type that is based on a DATE, TIME, or TIMESTAMP data type, the host variable must first be cast to a DATE, TIME, or TIMESTAMP data type.

Rules for result data types

Rules that are applied to the operands of an operation determine the data type of the result. Certain rules apply in certain situations and apply depending on the data type of operands.

The rules apply to:

- Corresponding columns in set operations (UNION, INTERSECT, or EXCEPT)
- Result expressions of a CASE expression
- Arguments of the scalar functions COALESCE and IFNULL
- Expression values of the IN list of an IN predicate

For the result data type of expressions that involve the operators '/', '*', '+' and '-', see "With arithmetic operators" on page 182.

For the result data type of expressions that involve the CONCAT operator, see "With the concatenation operator" on page 188.

Evaluation of the operands of an operation determines the data type of the result. If an operation has more than one pair of operands, DB2 determines the result type of the first pair, uses this result type with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type of the operation.

With the exception of the COALESCE function, the result of an operation can be null unless the operands do not allow nulls.

Numeric operands

Numeric types are compatible only with other numeric types.

Table 27. Result data types with numeric operands

One operand	Other operand	Data type of the result
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
BIGINT	SMALLINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	BIGINT	BIGINT

Table 27. Result data types with numeric operands (continued)

One operand	Other operand	Data type of the result
DECIMAL(<i>w</i> , <i>x</i>)	SMALLINT	DECIMAL(<i>p</i> , <i>x</i>) where $p = x + \max(w - x, 5)^1$
DECIMAL(<i>w</i> , <i>x</i>)	INTEGER	DECIMAL(<i>p</i> , <i>x</i>) where $p = x + \max(w - x, 11)^1$
DECIMAL(<i>w</i> , <i>x</i>)	BIGINT	DECIMAL(<i>p</i> , <i>x</i>) where $p = x + \max(w - x, 19)^1$
DECIMAL(<i>w</i> , <i>x</i>)	DECIMAL(<i>y</i> , <i>z</i>)	DECIMAL(<i>p</i> , <i>s</i>) where $p = \max(x, z) + \max(w - x, y - z)^1$ $s = \max(x, z)$
REAL	REAL	REAL
REAL	DECIMAL, BIGINT, INTEGER, or SMALLINT	DOUBLE
REAL	BIGINT	DOUBLE
DOUBLE	DOUBLE, REAL, DECIMAL, BIGINT, INTEGER, or SMALLINT	DOUBLE
DECFLOAT(<i>n</i>)	SMALLINT	DECFLOAT(<i>n</i>)
DECFLOAT(<i>n</i>)	INTEGER	DECFLOAT(<i>n</i>)
DECFLOAT(<i>n</i>)	BIGINT	DECFLOAT(34)
DECFLOAT(<i>n</i>)	DECIMAL(≤ 16 , <i>s</i>)	DECFLOAT(<i>n</i>)
DECFLOAT(<i>n</i>)	DECIMAL(> 16 , <i>s</i>)	DECFLOAT (34)
DECFLOAT(<i>n</i>)	REAL	DECFLOAT(<i>n</i>)
DECFLOAT(<i>n</i>)	DOUBLE	DECFLOAT(<i>n</i>)
DECFLOAT(<i>n</i>)	DECFLOAT(<i>m</i>)	DECFLOAT(max(<i>n</i> , <i>m</i>))

Notes:

1. Precision cannot exceed 31.

Character and graphic string operands

Character and graphic strings are compatible with other character and graphic strings as long as there is a conversion between their corresponding CCSIDs.

Table 28. Result data types with string operands

One operand	Other operand	Data type of the result
CHAR(<i>x</i>)	CHAR(<i>y</i>)	CHAR(<i>z</i>) where $z = \max(x, y)$
GRAPHIC(<i>x</i>)	CHAR(<i>y</i>)	VARGRAPHIC(<i>y</i>) where <i>y</i> > maximum length of a graphic
GRAPHIC(<i>x</i>)	CHAR(<i>y</i>)	GRAPHIC(<i>z</i>) where $z = \max(x, y)$
VARCHAR(<i>x</i>)	VARCHAR(<i>y</i>) or CHAR(<i>y</i>)	VARCHAR(<i>z</i>) where $z = \max(x, y)$
VARCHAR(<i>x</i>)	GRAPHIC(<i>y</i>)	VARGRAPHIC(<i>z</i>) where $z = \max(x, y)$

Table 28. Result data types with string operands (continued)

One operand	Other operand	Data type of the result
VARGRAPHIC(<i>x</i>)	VARGRAPHIC(<i>y</i>), GRAPHIC(<i>y</i>), VARCHAR(<i>y</i>), or CHAR(<i>y</i>)	VARGRAPHIC(<i>z</i>) where $z = \max(x, y)$
CLOB(<i>x</i>)	CLOB(<i>y</i>), VARCHAR(<i>y</i>), or CHAR(<i>y</i>)	CLOB(<i>z</i>) where $z = \max(x, y)$
CLOB(<i>x</i>)	GRAPHIC(<i>y</i>) or VARGRAPHIC(<i>y</i>)	DBCLOB(<i>z</i>) where $z = \max(x, y)$
DBCLOB(<i>x</i>)	CHAR(<i>y</i>), VARCHAR(<i>y</i>), CLOB(<i>y</i>), GRAPHIC(<i>y</i>), VARGRAPHIC(<i>y</i>), or DBCLOB(<i>y</i>)	DBCLOB(<i>z</i>) where $z = \max(x, y)$

Character string subtypes are determined as indicated in the following table:

Table 29. Result data types with character string operands

One operand	Other operand	Data type of the result
Bit data	Mixed, SBCS, or bit data	Bit data
Mixed data	Mixed or SBCS data	Mixed data
SBCS data	SBCS data	SBCS data

Binary string operands

Binary strings are compatible with other binary strings. Binary strings include BINARY, VARBINARY, and BLOB.

Table 30. Result data types with binary string operands

One operand	Other operand	Data type of the result
BINARY(<i>x</i>)	BINARY(<i>y</i>)	BINARY(<i>z</i>) where $z = \max(x, y)$
VARBINARY(<i>x</i>)	BINARY(<i>y</i>) or VARBINARY(<i>y</i>)	VARBINARY(<i>z</i>) where $z = \max(x, y)$
BLOB(<i>x</i>)	BINARY(<i>y</i>), VARBINARY(<i>y</i>), or BLOB(<i>y</i>)	BLOB(<i>z</i>) where $z = \max(x, y)$

Datetime operands

A date, time, or timestamp value is compatible with another value of the same type or any string expression that contains a valid string representation of the same type.

A DATE type is compatible with another DATE type or any string expression that contains a valid string representation of a date. A string representation is a value that is a built-in character string data type or graphic string data type. A string representation must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is DATE.

A TIME type is compatible with another TIME type or any string expression that contains a valid string representation of a time. A string representation is a value that is a built-in character string data type or graphic string data type. A string representation must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is TIME.

A TIMESTAMP type is compatible with another TIMESTAMP type or any string expression that contains a valid string representation of a timestamp. A string representation is a value that is a built-in character string data type or graphic string data type. A string representation must not be a CLOB or DBCLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is TIMESTAMP.

If both operands are in the same encoding scheme, the result is in that encoding scheme. Otherwise the result is in the application encoding scheme.

Row ID operands

A row ID data type is compatible only with itself. The result has a row ID data type.

XML operands

XML data is compatible only with other XML data. The data type of the result is XML.

Other data types can be treated as an XML data type by using the CAST specification or XMLPARSE functions to cast character, graphic, or binary data to XML data.

Distinct type operands

A distinct type is compatible only with itself. The data type of the result is the distinct type.

Conversion rules for operations that combine strings

When two strings are compared, one of the strings is first converted, if necessary, to the coded character set of the other string. Conversion is necessary only if certain rules apply.

Conversion is necessary only if all of the following are true:

- The CCSIDs of the two strings are different.
- Neither CCSID is X'FFFF' (neither string is defined as a binary string).
- The string selected for conversion is neither null nor empty.
- The following conversion tables (Table 32 on page 124 or Table 33 on page 125 indicate when conversion is necessary.

These same rules apply to determine the result CCSID in a fullselect.

The string selected for conversion depends on the type of the operands. For the purpose of CCSID determination, string expressions in a statement are divided into 6 types, as described in the following table.

Table 31. Operand types

Type of operand	CCSID of the operand type
Columns	CCSID from the containing table

Table 31. Operand types (continued)

Type of operand	CCSID of the operand type
String constants	CCSID associated with the application encoding scheme. For dynamic statements, this is the CURRENT APPLICATION ENCODING SCHEME special register. For static statements, this is the ENCODING bind option or the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures..
Special registers	CCSID associated with the application encoding scheme. For dynamic statements, this is the CURRENT APPLICATION ENCODING SCHEME special register. For static statements, this is the ENCODING bind option or the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures.
Host variables	CCSID specified in the DECLARE VARIABLE statement, associated with the application encoding scheme, or specified in SQLDAID or SQLDA
Derived value based on a column	<p>CCSID derived from the source of the derived value. A derived value based on a column is an expression whose source is directly or indirectly based on columns. The CCSID of such an expression is the CCSID derived from its source.</p> <p>For example:</p> <ul style="list-style-type: none"> • The CCSID of SUBSTR(column_1, 5, length(column_2)) is the CCSID of column_1. Note that the CCSID of column_2 has no influence on the CCSID of SUBSTR. • The CCSID of column_1 'ABC' is the CCSID of column_1, derived from the rules described in Table 32 on page 124. • The CCSID of column_1 GX'42C1' is the DBCS CCSID from the CCSID set of column_1, derived from the rules described in Table 32 on page 124 and Table 33 on page 125. • The CCSID of COALESCE(EBCDIC_column_1, ASCII_column_1) is the UNICODE CCSID, derived from the rules described in Table 32 on page 124. • The CCSID of CAST(string_column_1 AS GRAPHIC(10)) is the DBCS CCSID from the CCSID set of string_column_1. • The CCSID of CAST(EBCDIC_string_column_1 AS VARCHAR(10) CCSID UNICODE) is the UNICODE CCSID derived from the rules described in Table 32 on page 124. • The CCSID of CASE WHEN(1=1) THEN '1' ELSE ASCII_column_1 END is the CCSID of ASCII_column_1, derived from the rules described in Table 32 on page 124. • The CCSID of CASE WHEN(1=1) THEN EBCDIC_column_1 ELSE ASCII_column_1 END is the UNICODE CCSID derived from the rules described in Table 32 on page 124. • The CCSID of a scalar fullselect (SELECT column_1 FROM table_1) is the CCSID of column_1.

Table 31. Operand types (continued)

Type of operand	CCSID of the operand type
Derived value not based on a column	<p>CCSID derived from the source of the derived value. A derived value not based on a column is an expression whose source is not directly or indirectly based on any column. The CCSID of such an expression is the CCSID derived from its source.</p> <ul style="list-style-type: none"> For example, the CCSID of SUBSTR('ABDC', 1, length('AB')) is the CCSID of the string constant 'ABCD'. Note that the CCSID of column_1 has no influence on the CCSID of SUBSTR. the CCSID of user_defined_function1(column1) is the output CCSID defined by user_defined_function1. the CCSID of the cast function of distinct type, shape, is the CCSID of distinct type, shape. the CCSID of CURRENT SQLID UX'0041' is the UNICODE DBCS CCSID, derived from the rules described in Table 32 and Table 33 on page 125. the CCSID of CAST('abc' as CHAR(10) CCSID UNICODE) is the UNICODE CCSID.

The following table shows which operand supplies the target CCSID set when the comparison is part of an SQL statement involving multiple tables with different CCSID sets.

Table 32. Operand that supplies the CCSID for character conversion

First operand	Second operand				Derived value based on a column	Derived value not based on a column
	Column value	String constant	Special register	Host variable		
Column value	1	first operand	first operand	first operand	1	first operand
String constant	second operand	1	1	1	second operand	1
Special register	second operand	1	1	1	second operand	1
Host variable	second operand	1	1	1	second operand	1
Derived value based on a column	1	first operand	first operand	first operand	1	first operand
Derived value not based on a column	second operand	1	1	1	second operand	1
Note: 1. If the CCSID sets are different, both operands are converted, if necessary, to Unicode. SBCS and Mixed are converted to UTF-8. DBCS is converted to UTF-16. See the next table to determine which operand supplies the CCSID for character conversion.						

The following table shows which operand is selected for conversion when both operands are based on a column or are not based on a column as represented in the previous table.

Table 33. Operand that supplies the CCSID for character conversion when both operands are based or not based on a column

First operand	Second operand		
	SBCS data	Mixed data	DBCS data
SBCS data		second operand ¹	second operand
Mixed data	first operand ¹		second operand
DBCS data	first operand	first operand	

Note: 1. For ASCII and EBCDIC data, the conversion depends on the value of the field MIXED DATA on installation panel DSNTIPF at the DB2 that does the comparison. If MIXED DATA = YES, the SBCS operand is converted to MIXED. If MIXED DATA = NO, the MIXED operand is converted to SBCS.

For example, assume a comparison of the form:

string-constant-SBCS =derived-value-not-based-on-column-DBCS

Assume that the operands have different encoding schemes. First look at Table 32 on page 124. The relevant table entry is in the third row and second column. The value for this entry shows that if the CCSID sets are different, the operands are converted to Unicode. The first operand (string-constant-SBCS) is converted to UTF-8 (Mixed) if it is not already Unicode. In addition, the second operand (derived-value-not-based-on-column-DBCS) is converted to UTF-16 (Unicode DBCS) if necessary. After the operands have been converted to Unicode, Table 33 is used to determine which operand supplies the specific CCSID for the conversion. The relevant table entry is in the second row and third column. It indicates that the second operand (derived-value-not-based-on-column-DBCS) determines the CCSID because DBCS data takes precedence over Mixed data.

An error occurs if a character of the string cannot be converted, the SYSSTRINGS table is used but contains no information about the pair of CCSIDs of the operands being compared, or DB2 cannot do the conversion through z/OS support for Unicode. A warning occurs if a character of the string is converted to a substitution character.

A derived value based on a column is an expression that includes columns that affects the result CCSID of the expression. For example, in the expression COL1 || 'abc', COL1 determines the result CCSID. Therefore, the expression COL1 || 'abc' is considered to be a derived value based on a column that continues to give the column precedence in any further comparisons. The expression CASE WHEN COL1 > 1 THEN 'abc' ELSE 'def' END contains a column that does not affect the result CCSID of the expression and is therefore not considered to be a derived value based on a column.

The following table defines which expressions are considered to be a derived value based on a column.

Table 34. Derived values based on a column

Expression	Condition
<i>expression1</i> <i>expression2</i>	<i>expression1</i> or <i>expression2</i> is a column or a derived value based on a column
CASE <i>when-clause</i> THEN <i>result-expression</i> ELSE <i>result-expression</i> END	any <i>result-expression</i> is a <i>string-expression</i> that is a column or derived value based on a column

Table 34. Derived values based on a column (continued)

Expression	Condition
CAST(<i>expression</i> as <i>data-type</i>)	<i>expression</i> is a <i>string-expression</i> that is a column or a derived value based on a column and <i>data-type</i> is a string data type
Scalar-fullselect: (SELECT <i>expression</i> FROM <i>table</i>)	<i>expression</i> is a <i>string-expression</i> that is a column or a derived value based on a column and <i>data-type</i> is a string data type

When a statement contains multiple CCSID sets, if the length of one of the strings changes after CCSID conversion, the string becomes a varying-length string. That is, the data type becomes VARCHAR, CLOB, VARGRAPHIC, or DBCLOB. The following table shows the worse case resulting lengths of CCSID conversion, where X is length in bytes.

Table 35. Worst case result length of CCSID conversion, where X represents LENGTH(string in bytes)

From CCSID		To CCSID								
		EBCDIC			ASCII			Unicode		
		SBCS	Mixed	DBCS	SBCS	Mixed	DBCS	SBCS	UTF-8	UTF-16
EBCDIC	SBCS	X	X	X*2 ¹	X	X	X*2 ¹	X ¹	X*3	X*2
	Mixed	X	X	X*2 ¹	X	X	X*2 ¹	X ¹	X*3	X*2
	DBCS	X*0.5 ¹	X+2	X	X*0.5 ¹	X	X	X*0.5	X*1.5	X
ASCII	SBCS	X	X	X*2 ¹	X	X	X*2 ¹	X ¹	X*3	X*2
	Mixed	X	X*1.8	X*2 ¹	X	X	X*2 ¹	X ¹	X*3	X*2
	DBCS	X*0.5 ¹	X+2	X	X*0.5 ¹	X	X	X*0.5	X*1.5	X
Unicode	SBCS	X	X	X*2	X	X	X*2	X	X	X*2
	UTF-8	X	X*1.25	X	X	X	X	X	X	X*2
	UTF-16	X*0.5	X+2	X	X*0.5	X	X	X*0.5	X*1.5	X

Note:

1. Because of the high probability of data loss, IBM does not provide conversion tables for this combination of two CCSIDs and data subtypes.

Constants

A *constant* (also called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, decimal, or decimal floating-point. String constants are classified as character, graphic, or binary.

All constants have the attribute NOT NULL. A negative sign in a numeric constant with a value of zero is ignored, except for a decimal floating-point constant.

Constants have a built-in data type. Therefore, an operation that involves a constant and a distinct type requires that the distinct type be cast to the built-in data type of the constant or the constant be cast to the distinct type. For example, see Table 26 on page 118, which contains an example of casting data types to compare a constant to a distinct type.

Integer constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point.

The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of a large integer, but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Examples:

64 -15 +100 32767 720176

In syntax diagrams, the term *integer* is used for a large integer constant that must not include a sign.

Floating-point constants

A *floating-point constant* specifies a double-precision floating-point number as two numbers separated by an E.

The first number can include a sign and a decimal point. The second number can include a sign but not a decimal point. The value of the constant is the product of the first number and the power of 10 specified by the second number. It must be within the range of floating-point numbers. The number of characters in the constant must not exceed 30. Excluding leading zeros, the number of digits in the first number must not exceed 17 and the number of digits in the second must not exceed 2.

Examples: The following floating-point constants represent the numbers '150', '200000', -0.22, and '500':

15E1 2.E5 -2.2E-1 +5.E+2

Decimal constants

A *decimal constant* is a signed or unsigned number of no more than 31 digits and either includes a decimal point or is not within the range of binary integers.

The precision is the total number of digits, including those, if any, to the right of the decimal point. The total includes all leading and trailing zeros. The scale is the number of digits to the right of the decimal point, including trailing zeros.

Examples: The following decimal constants have, respectively, precisions and scales of 5 and 2; 4 and 0; 2 and 0; and 23 and 2:

025.50 1000. -15. +37589333333333333333.33

Decimal floating-point constants

A *decimal floating-point constant* specifies a decimal floating-point number as two numbers separated by an E. The first number can include a sign and a decimal point. The second number can include a sign but not a decimal point.

The value of the constant is the product of the first number and the power of 10 specified by the second number. The value must be within the range of DECFLOAT(34). The number of characters in the constant must not exceed 42.

Excluding leading zeros, the number of digits in the first number must not exceed 34 and the number of digits in the second number must not exceed 4.

A constant that is specified as two numbers separated by an E is a decimal-floating point constant only if the value is outside the range of a floating-point constant. A constant that is specified as a number that does not contain an E, and has more than 31 digits, is also a decimal-floating point constant.

In addition to numeric constants, the following reserved keywords can be used to specify decimal-floating point special values:

- INF or INFINITY - represents infinity
- NAN - represents quiet not-a-number
- SNAN - represents signaling not-a-number

The keywords can be any combination of uppercase or lowercase letters and can be preceded by an operational sign (+ or -).

SNAN results in a warning or exception when it is used in a numerical operation; NAN does not. SNAN can be used in non-numerical operations without causing a warning or exception. For example, SNAN can be used in the VALUES list of an insert operation or as a constant used in a comparison in a predicate.

When the keywords are used in a predicate, the following order of precedence applies:

`-NAN < -SNAN < -INFINITY < -0 < 0 < INFINITY < SNAN < NAN`

Examples: The following decimal floating-point constants represent the numbers 123456789012345678, sNaN, and negative infinity:

123456789012345678E0 SNAN -INFINITY

Character string constants

A *character string constant* specifies a varying-length character string. There are two forms of character string constant.

- A sequence of characters that starts and ends with a string delimiter, which is either an apostrophe (') or a quotation mark ("). For the factors that determine which is applicable, see "Apostrophes and quotation marks as string delimiters" on page 253. This form of string constant specifies the character string contained between the string delimiters. The number of bytes between the delimiters must not be greater than 32704. The limit of 32704 refers to the length (in bytes) of the UTF-8 representation of the string. If you produced the string in a CCSID other than UTF-8 (for example, an EBCDIC CCSID), the length of the UTF-8 representation might differ from the length of the string's representation in the source CCSID. Two consecutive string delimiters are used to represent one string delimiter within the character string.
- An X followed by a sequence of characters that starts and ends with a string delimiter. This form of a character string constant is also called a *hexadecimal constant*. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 32704. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. A hexadecimal constant allows you to specify characters that do not have a keyboard representation.

Examples:

'12/14/1985' '32' 'DON''T CHANGE' X'FFFF' ''

The right most string in the example (") represents an empty character string constant, which is a string of zero length.

A character string constant is classified as mixed data if it includes a DBCS substring. In all other cases, a character string constant is classified as SBCS data. For information about the CCSID that is assigned to the constant, see "Determining the encoding scheme and CCSID of a string" on page 43. A mixed string constant can be continued from one line to the next only if the break occurs between single byte characters. A Unicode string is always considered mixed regardless of the content of the string.

For Unicode, character constants can be assigned to UTF-8 and UTF-16. The form of the constant does not matter. Typically, character string constants are used only with character strings, but they also can be used with graphic UTF-16 data. However, hexadecimal constants are just character data. Thus, hexadecimal constants being used to insert data into UTF-16 data strings should be in UTF-8 format, not UTF-16 format. For example, if you wanted to insert the number 1 into a UTF-16 column, you would use X'31', not X'0031'. Even though X'0031' is a UTF-16 value, DB2 treats it as two separate UTF-8 code points. Thus, X'0031' would become X'00000031'.

Binary string constants

A binary-string constant specifies a varying-length binary string.

A binary-string constant is formed by specifying a BX followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 32704.

A hexadecimal digit is a digit or any of the letters A through F (upper case or lower case). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents one byte. Note that this representation is similar to the representation of the character-constant that uses the X" form; however binary-string constant and character-string constant are not compatible and the X" form can not be used to specify a binary-string constant, just as the BX" form cannot be used to specify a character-string constant.

Examples of binary-string constants:

BX'0000' BX'C141C242' BX'FF00FF01FF'

Datetime constants

A *datetime constant* is a character string constant of a particular format.

Character-string constants are described under "Character string constants" on page 128.

For information about the valid string formats, see "String representations of datetime values" on page 89.

Graphic string constants

A *graphic string constant* specifies a varying-length graphic string.

In EBCDIC environments, the forms of graphic string constants are shown in the following figure. (Shift-in and shift-out characters for EBCDIC data are discussed in “Character strings” on page 73.)⁶

Context	Graphic String Constant	Empty String	Example
PL/I	$\text{\textcircled{S}}_0 \text{\textcircled{!}} \text{dbcs-string} \text{\textcircled{!}} \text{G} \text{\textcircled{S}}_1$	$\text{\textcircled{S}}_0 \text{\textcircled{!}} \text{\textcircled{!}} \text{G} \text{\textcircled{S}}_1$	$\text{\textcircled{S}}_0 \text{\textcircled{!}} \text{元氣} \text{\textcircled{!}} \text{G} \text{\textcircled{S}}_1$
	$\text{\textcircled{S}}_0 \text{\textcircled{!}} \text{dbcs-string} \text{\textcircled{!}} \text{\textcircled{S}}_1 \text{G}$	$\text{\textcircled{S}}_0 \text{\textcircled{!}} \text{\textcircled{S}}_1 \text{G}$	
<div><div>G represents a DBCS G (X'42C7')</div><div>! represents a DBCS apostrophe (X'427D')</div></div>			
All other contexts	$\text{G} \text{\textcircled{S}}_0 \text{dbcs-string} \text{\textcircled{S}}_1 \text{'}$	$\text{G} \text{\textcircled{S}}_0 \text{\textcircled{S}}_1 \text{'}$	$\text{G} \text{\textcircled{S}}_0 \text{元氣} \text{\textcircled{S}}_1 \text{'}$
		G''	
		$\text{g} \text{\textcircled{S}}_0 \text{\textcircled{S}}_1 \text{'}$	
		g''	
	$\text{N} \text{\textcircled{S}}_0 \text{dbcs-string} \text{\textcircled{S}}_1 \text{'}$	$\text{N} \text{\textcircled{S}}_0 \text{\textcircled{S}}_1 \text{'}$	
		N''	
		$\text{n} \text{\textcircled{S}}_0 \text{\textcircled{S}}_1 \text{'}$	
		n''	

Figure 17. Graphic string constants in EBCDIC

In SQL statements and in host language statements in a source program, graphic string constants cannot be continued from one line to the next. A graphic string constant must be short enough so that its UTF-8 representation requires no more than 32704 bytes.

DB2 supports two types of hexadecimal graphic string constants.

- UX'xxxx' represents a string of graphic Unicode UTF-16 characters, where *x* is a hexadecimal digit. The number of digits must be a multiple of 4 and must not exceed 32704. Each group of 4 digits represents a single UTF-16 graphic character. For example, the UX constant for 'ABC' is UX'004100420043'.
- GX'xxxx' represents a string of graphic characters, where *x* is a hexadecimal digit. The number of digits must be a multiple of 4. Each group of 4 digits represents a single double-byte graphic character. The hexadecimal shift-in and shift-out ('OE'X and 'OF'X), which apply to EBCDIC only, are not included in the string.

If the MIXED DATA install option is set to NO, a GX constant cannot be used. Instead, a UX constant should be used. A GX constant cannot be used when the encoding scheme is UNICODE.

For information about the CCSID that is assigned to a graphic string constant, including UX'xxxx' and GX'xxxx' string constants, see “Determining the encoding scheme and CCSID of a string” on page 43.

6. The PL/I form of graphic string constants is supported only in static SQL statements.

Special registers

| A special register is a storage area that is defined for an application process by
| DB2 and is used to store information that can be referenced in SQL statements. A
| reference to a special register is a reference to a value provided by the current
| server. If the value is a string, its CCSID is a default CCSID of the current server.

The special registers can be referenced as follows:

special registers

CURRENT APPLICATION ENCODING SCHEME
CURRENT CLIENT_ACCTNG
CURRENT CLIENT_APPLNAME
CURRENT CLIENT_USERID
CURRENT CLIENT_WRKSTNNAME
CURRENT DATE
(1)
CURRENT_DATE
CURRENT DEBUG MODE
CURRENT DECFLOAT ROUNDING MODE
CURRENT DEGREE
LOCALE
CURRENT LC_CTYPE
CURRENT LC_CTYPE
TABLE
CURRENT MAINTAINED TYPES FOR OPTIMIZATION
CURRENT MEMBER
CURRENT OPTIMIZATION HINT
CURRENT PACKAGE PATH
CURRENT PACKAGESET
CURRENT PATH
CURRENT_PATH
CURRENT PRECISION
CURRENT REFRESH AGE
CURRENT ROUTINE VERSION
CURRENT RULES
CURRENT SCHEMA
(1)
CURRENT_SCHEMA
CURRENT SERVER
CURRENT SQLID
CURRENT TIME
(1)
CURRENT_TIME
CURRENT TIMESTAMP
(1)
CURRENT_TIMESTAMP
CURRENT TIMEZONE
(2)
ENCRYPTION PASSWORD
SESSION_USER
USER

Notes:

- 1 The SQL standard uses the form with the underline.
- 2 The ENCRYPTION PASSWORD special register can only be explicitly referenced in the SET ENCRYPTION PASSWORD statement. It is used implicitly used by the encryption and decryption functions.

General rules for special registers

A set of general rules applies for special registers. Specific rules for each special register are described with that special register.

Following these general rules for special registers, each special register is described individually.

Changing register values: A commit operation might cause special registers to be re-initialized. Whether a special register is affected by a commit depends on whether the special register has been explicitly set within the application process. For example, assume that the PATH special register has not been explicitly set with a SET PATH statement in the application process. After a commit, the value of PATH is re-initialized. For information on the initialization of PATH, which can take the current value of CURRENT SQLID into consideration, see “CURRENT SQLID” on page 148.

A rollback operation has no effect on the values of special registers. Nor does any SQL statement, with the following exceptions:

- SQL SET statements can change the values of the following special registers:
 - CURRENT APPLICATION ENCODING SCHEME
 - CURRENT DEBUG MODE
 - CURRENT DECFLOAT ROUNDING MODE
 - CURRENT DEGREE
 - CURRENT LOCALE LC_CTYPE
 - CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
 - CURRENT OPTIMIZATION HINT
 - CURRENT PACKAGE PATH
 - CURRENT PACKAGESET
 - CURRENT PATH
 - CURRENT PRECISION
 - CURRENT REFRESH AGE
 - CURRENT ROUTINE VERSION
 - CURRENT RULES
 - CURRENT SCHEMA
 - CURRENT SQLID⁷
 - ENCRYPTION PASSWORD
- SQL CONNECT statements can change the value of CURRENT SERVER.

You can use various statements to determine the value of a special register. For instance, a SELECT statement, a SET statement, the VALUES statement (if the statement is within a trigger action) will provide the value of a special register. The following examples find the value of the CURRENT PRECISION special register:

```
SELECT CURRENT PRECISION FROM SYSIBM.SYSDUMMY1;
SET :hv = CURRENT PRECISION
VALUES(CURRENT PRECISION)
```

CCSIDS for register values: Special registers that contain character strings have an associated CCSID. The particular CCSID depends on the context in which the special register is referenced. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Datetime special registers: The datetime registers are named CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP. Datetime special registers are

7. If the SET CURRENT SQLID statement is executed in a stored procedure or user-defined function package that has a dynamic SQL behavior other than run behavior, the SET CURRENT SQLID statement does not affect the authorization ID that is used for dynamic SQL statements in the package. The dynamic SQL behavior determines the authorization ID. For more information, see the discussion of DYNAMICRULES in *DB2 Command Reference*.

stored in an internal format. When two or more of these registers are implicitly or explicitly specified in a single SQL statement, they represent the same point in time. A datetime special register is implicitly specified when it is used to provide the default value of a datetime column.

If the SQL statement in which a datetime special register is used is in a user-defined function or stored procedure that is within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement to determine the special register value.

The values of these special registers are based on:

- The time-of-day clock of the processor for the server executing the SQL statement
- The TIMEZONE parameter for this processor. The TIMEZONE parameter is in SYS1.PARMLIB(CLOCKXX).

To evaluate the references when the statement is being executed, a single reading from the time-of-day clock is incremented by the number of hours, minutes, and seconds specified by the TIMEZONE parameter. The values derived from this are assumed to be the local date, time, or timestamp, where local means local to the DB2 that executes the statement. This assumption is correct if the clock is set to local time and the TIMEZONE parameter is zero or the clock is set to GMT and the TIMEZONE parameter gives the difference from GMT. Universal time, coordinated (UTC) is another name for Greenwich Mean Time (GMT).

Since the datetime special registers and the CURRENT TIMEZONE special register depend on the parameter PARMTZ(SYS1.PARMLIB(CLOCKXX)), their values are affected if the local time at the server is changed by the z/OS system command SET CLOCK. The values of the CURRENT DATE and CURRENT TIMESTAMP special registers might be affected if the local date at the server is changed by the system command SET DATE⁸.

Where special registers are processed: In distributed applications, CURRENT APPLICATION ENCODING SCHEME, CURRENT SERVER, and CURRENT PACKAGESET are processed locally. All other special registers are processed at the server.

CURRENT APPLICATION ENCODING SCHEME

CURRENT APPLICATION ENCODING SCHEME specifies which encoding scheme is to be used for dynamic statements. It allows an application to indicate the encoding scheme that is used to process data. This register is not supported in REXX applications or in stored procedures written in REXX.

The initial value of CURRENT APPLICATION ENCODING SCHEME is determined by the value of the ENCODING bind option or the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures if the option is specified. If the option was not specified, the initial value is the value of field DEFAULT APPLICATION ENCODING SCHEME on installation panel DSNTIPF. You can change the value of the register by executing the statement SET CURRENT APPLICATION ENCODING SCHEME. For a description of the statement, see "SET CURRENT APPLICATION ENCODING SCHEME" on page 1477.

8. Whether the SET DATE command affects these special registers depends on the system level and the program temporary fix (PTF) level of the system.

The value contained in the special register is a character representation of a CCSID. Although you can use the values ASCII, EBCDIC, or UNICODE to set the special register, what is stored in the special register is a character representation of the numeric CCSID that corresponds to the value used in the SET CURRENT APPLICATION ENCODING SCHEME statement. The value ASCII, EBCDIC, or UNICODE is not stored. The CCSID_ENCODING scalar function can be used to get a value of ASCII, EBCDIC, or UNICODE from a numeric CCSID value.

The data type is CHAR(8). If necessary, the value is padded on the right with blanks so that its length is 8 bytes.

For stored procedures and user-defined functions, the initial value of the CURRENT APPLICATION ENCODING SCHEME special register is determined by the value of the ENCODING bind option for the package that is associated with the procedure or function or by the APPLICATION ENCODING SCHEME option of the CREATE PROCEDURE or ALTER PROCEDURE statement for a native SQL procedure. If the option was not specified, the initial value is the value of the field DEFAULT APPLICATION ENCODING SCHEME field on installation panel DSNTIPF.

For triggers, the initial value of the CURRENT APPLICATION ENCODING SCHEME special register is the value of field DEFAULT APPLICATION ENCODING SCHEME on installation panel DSNTIPF.

Example: The CURRENT APPLICATION ENCODING SCHEME special register can be used like any other special register:

```
EXEC SQL VALUES(CURRENT APPLICATION ENCODING SCHEME) INTO :HV1;
EXEC SQL INSERT INTO T1 VALUES (CURRENT APPLICATION ENCODING SCHEME);
EXEC SQL SET :HV1 = CURRENT APPLICATION ENCODING SCHEME;
EXEC SQL SELECT C1 FROM T1 WHERE C1 = CURRENT APPLICATION ENCODING SCHEME;
```

CURRENT CLIENT_ACCTNG

CURRENT CLIENT_ACCTNG contains the value of the accounting string from the client information that is specified for the connection.

The data type is VARCHAR(255).

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- Set Client Information (sqleseti)
- DB2Connection.setDB2ClientAccountingInformation(String info)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

The value for the accounting string will be obtained first from the accounting string that is set by the SET_CLIENT_ID function, AUTH SIGNON function, or the Set Client Information (sqleseti) API, or alternatively from the accounting token set by RRSAF if accounting string has not been set.

In a distributed environment, if the value set by the API exceeds 200 bytes, it is truncated to 200 bytes. If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

Example: Get the current value of the accounting string for this connection.

```
SET :ACCT_STRING = CURRENT CLIENT_ACCTNG
```

CURRENT CLIENT_APPLNAME

CURRENT CLIENT_APPLNAME contains the value of the application name from the client information that is specified for the connection.

The data type is VARCHAR(255).

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- Set Client Information (sqleseti)
- DB2Connection.setDB2ClientApplicationInformation(String info)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

In a distributed environment, if the value set by the API exceeds 32 bytes, it is truncated to 32 bytes. If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

Example: Select the departments that are allowed to use the application that is being used in this connection.

```
SELECT DEPT
FROM DEPT_APPL_MAP
WHERE APPL_NAME = CURRENT CLIENT_APPLNAME
```

CURRENT CLIENT_USERID

CURRENT CLIENT_USERID contains the value of the client user ID from the client information that is specified for the connection.

The data type is VARCHAR(255).

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- Set Client Information (sqleseti)
- DB2Connection.setDB2ClientUser(String info)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

In a distributed environment, if the value set by the API exceeds 16 bytes, it is truncated to 16 bytes. If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

Example: Find out in which department the current client user ID works.

```
SELECT DEPT
FROM DEPT_USERID_MAP
WHERE USER_ID = CURRENT CLIENT_USERID
```

CURRENT CLIENT_WRKSTNNAME

CURRENT CLIENT_WRKSTNNAME contains the value of the workstation name from the client information that is specified for the connection.

The data type is VARCHAR(255).

The value of the special register can be changed by using one of the following application programming interfaces (APIs):

- Set Client Information (sqleseti)
- DB2Connection.setDB2ClientWorkstation(String info)
- The RRS DSNRLI SIGNON, AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID function
- The WLM_SET_CLIENT_INFO stored procedure

In a distributed environment, if the value set by the API exceeds 18 bytes, it is truncated to 18 bytes. If one of these APIs is not used to set the value of the special register, an empty string is returned when the special register is referenced.

Example: Get the name of the workstation that is being used in this connection.

```
SET :WS_NAME = CURRENT CLIENT_WRKSTNNAME
```

CURRENT DATE

The CURRENT DATE special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than one time within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.⁹

The value of CURRENT DATE in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151. For other applications, the date is derived by the DB2 that executes the SQL statement that refers to the special register. For a description of how the date is derived, see Datetime special registers.

Specifying CURRENT_DATE is equivalent to specifying CURRENT DATE.

Example: Display the average age of employees.

```
SELECT AVG(YEAR(CURRENT DATE - BIRTHDATE))  
FROM DSN8910.EMP;
```

CURRENT DEBUG MODE

CURRENT DEBUG MODE specifies the default value for the DEBUG MODE option when certain routines are created. The DEBUG MODE option specifies whether the routine should be built with the ability to run in debugging mode.

CURRENT DEBUG MODE specifies the default value for the DEBUG MODE option of the following statements:

- ALTER PROCEDURE for a new version of a native SQL procedure
- CREATE PROCEDURE for a Java procedure
- CREATE PROCEDURE for a native SQL procedure

The data type is VARCHAR(8). The following values are valid:

- ALLOW — Specifies that the routine can be run in debugging mode.

9. Except for the case of a non-atomic multiple row INSERT or MERGE statement.

- **DISALLOW** — Specifies that the routine cannot be run in debugging mode. A subsequent ALTER statement can change the DEBUG MODE option to allow the routine to run in debugging mode.
- **DISABLE** — Specifies that the routine can never be run in debugging mode. When DISABLE is in effect, the routine cannot be changed to run in debugging mode. A subsequent ALTER statement cannot change the DEBUG MODE option to allow or disallow the routine to run in debugging mode.

The value of CURRENT DEBUG MODE in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151. In other contexts the initial value of CURRENT DEBUG MODE is DISALLOW.

You can change the value of the CURRENT DEBUG MODE special register by running the SET CURRENT DEBUG MODE statement. For details about this statement, see “SET CURRENT DEBUG MODE” on page 1478.

Example: Set the host variable DEBUG_MODE_OPT to the value of the CURRENT DEBUG MODE special register:

```
VALUES CURRENT DEBUG MODE INTO :DEBUG_MODE_OPT;
```

CURRENT DECFLOAT ROUNDING MODE

CURRENT DECFLOAT ROUNDING MODE specifies the default rounding mode that is used for DECFLOAT values.

The data type is VARCHAR(128). The following rounding modes are supported:

- **ROUND_CEILING** — rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by 1.
- **ROUND_DOWN** — rounds the value towards 0 (truncation). The discarded digits are ignored.
- **ROUND_FLOOR** — rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1.
- **ROUND_HALF_DOWN** — rounds the value to the nearest value; if the values are equidistant, rounds the value towards zero. If the discarded digits represent greater than half (0.5) of the value of a one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored. This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.
- **ROUND_HALF_EVEN** — rounds the value to the nearest value; if the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represents greater than half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 if its rightmost digit is odd (to make an even digit).
- **ROUND_HALF_UP** — rounds the value to the nearest value; if the values are equidistant, rounds the value away from zero. If the discarded digits represent greater than or equal to half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored.

- **ROUND_UP** — rounds the value away from 0. If all of the discarded digits are zero the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by 1. This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.

The initial value of **CURRENT DECFLOAT ROUNDING MODE** is the value of the **ROUNDING** bind option or the native SQL procedure option. If the **ROUNDING** option is not specified, the initial value is the value of the **DEF DECFLOAT ROUND MODE** field on installation panel **DSNTIPF**.

The value of **CURRENT DECFLOAT ROUNDING MODE** in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the **CURRENT DECFLOAT ROUNDING MODE** by executing the statement **SET CURRENT DECFLOAT ROUNDING MODE**. For details about this statement, see “**SET CURRENT DECFLOAT ROUNDING MODE**” on page 1480.

Example: Set the **DECFLOAT** rounding mode to **ROUND_CEILING**:

```
SET CURRENT DECFLOAT ROUNDING MODE = 'ROUND_CEILING';
```

CURRENT DEGREE

CURRENT DEGREE specifies the degree of parallelism for the execution of queries that are dynamically prepared by the application process.

The data type of the register is **CHAR(3)** and the only valid values are 1 (padded on the right with two blanks) and **ANY**.

If the value of **CURRENT DEGREE** is 1 when a query is dynamically prepared, the execution of that query will not use parallelism. If the value of **CURRENT DEGREE** is **ANY** when a query is dynamically prepared, the execution of that query can involve parallelism. See *DB2 Performance Monitoring and Tuning Guide* for a description of query parallelism.

The initial value of **CURRENT DEGREE** is determined by the value of field **CURRENT DEGREE** on installation panel **DSNTIP8**. The default for the initial value of that field is 1 unless your installation has changed it to be **ANY** by modifying the value in that field. The initial value of **CURRENT DEGREE** in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the register by executing the statement **SET CURRENT DEGREE**. For details about this statement, see “**SET CURRENT DEGREE**” on page 1482.

CURRENT DEGREE is a register at the database server. Its value applies to queries that are dynamically prepared at that server and to queries that are dynamically prepared at another DB2 subsystem as a result of the use of a DB2 private connection between that server and that DB2 subsystem.

Example: The following statement inhibits parallelism:

```
SET CURRENT DEGREE = '1';
```

CURRENT LOCALE LC_CTYPE

CURRENT LOCALE LC_CTYPE specifies the LC_CTYPE locale that will be used to execute SQL statements that use a built-in function that references a locale. Functions LCASE, UCASE, and TRANSLATE (with a single argument) refer to the locale when they are executed.

The data type is CHAR(50). If necessary, the value is padded on the right with blanks so that its length is 50 bytes.

The initial value of CURRENT LOCALE LC_CTYPE is determined by the value of field LOCALE LC_CTYPE on installation panel DSNTIPF. The default for the initial value of that field is blank unless your installation has changed the value of that field. The initial value of CURRENT LOCALE LC_CTYPE in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the register by executing the statement SET CURRENT LOCALE LC_CTYPE. For details about this statement, see “SET CURRENT LOCALE LC_CTYPE” on page 1483.

Example: Save the value of current register CURRENT LOCALE LC_CTYPE in host variable HV1, which is defined as VARCHAR(50).

```
EXEC SQL VALUES(CURRENT LOCALE LC_CTYPE) INTO :HV1;
```

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION specifies a value that identifies the types of objects that can be considered to optimize the processing of dynamic SQL queries. This register contains a keyword representing table types.

The data type is VARCHAR(255).

The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is determined by the value of field CURRENT MAINT TYPES on installation panel DSNTIP8. The default for the initial value of that field is SYSTEM unless your installation has changed the value of that field. The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the register by executing the SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement. For details about this statement, see “SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION” on page 1485. The object types controlled by this special register are never considered by static embedded SQL queries.

Example: Set the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register so that all materialized query tables are considered.

```
SET CURRENT MAINTAINED TABLE TYPES ALL;
```

CURRENT MEMBER

CURRENT MEMBER specifies the member name of a current DB2 data sharing member on which a statement is executing. The value of CURRENT MEMBER is a character string.

The data type is CHAR(8). If necessary, the member name is padded to the right with blanks so that its length is 8 bytes.

The value of a CURRENT MEMBER is a string of blanks when the application process is connected to a DB2 subsystem that is not a member of a data sharing group.

The SQL SET statement cannot change the value of CURRENT MEMBER.

Example: Use one of the following statements to set the host variable MEM to the name of the current DB2 member.

```
EXEC SQL SET :MEM = CURRENT MEMBER;  
EXEC VALUES (CURRENT MEMBER) into :MEM;
```

CURRENT OPTIMIZATION HINT

CURRENT OPTIMIZATION HINT specifies the user-defined optimization hint that DB2 should use to generate the access path for dynamic statements.

The data type is VARCHAR(128).

The value of the register identifies the rows in *owner*.PLAN_TABLE that DB2 uses to generate the access path. DB2 uses information in the rows in *owner*.PLAN_TABLE for which the value of the OPTHINT column matches the value of the CURRENT OPTIMIZATION special register. If the value of the register is an empty string or all blanks, DB2 uses normal optimization and ignores optimization hints. If the value of the register includes any non-blank characters and DB2 was installed without optimization hints enabled (field OPTIMIZATION HINTS on installation panel DSNTIP8), a warning occurs.

The initial value of CURRENT OPTIMIZATION HINT is the value of the OPTHINT bind option or of the native SQL procedure option. The initial value of CURRENT OPTIMIZATION HINT in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151. You can change the value of the special register by executing the statement SET CURRENT OPTIMIZATION HINT. For details about this statement, see “SET CURRENT OPTIMIZATION HINT” on page 1487.

Example: Set the CURRENT OPTIMIZATION HINT special register so that DB2 uses the optimization plan hint that is identified by host variable NOHYB when generating the access path for dynamic statements.

```
SET CURRENT OPTIMIZATION HINT = :NOHYB
```

For more information about telling DB2 how to generate access paths, see *DB2 Application Programming and SQL Guide*.

CURRENT PACKAGE PATH

CURRENT PACKAGE PATH specifies a value that identifies the path used to resolve references to packages that are used to execute SQL statements. This special register applies to both static and dynamic statements.

The data type is VARCHAR(4096). The value can be an empty or blank string, or a list of one or more collection IDs, where the collection IDs are enclosed in double quotation marks and separated by commas. Any quotation marks within the string

are repeated as they are in any delimited identifier. The delimiters and commas are included in the length of the special register.

The initial value of CURRENT PACKAGE PATH is an empty string. The value is a list of collections only if the application process has explicitly specified a list of collections by means of the SET CURRENT PACKAGE PATH statement. For details about this statement, see “SET CURRENT PACKAGE PATH” on page 1488. The initial value of CURRENT PACKAGE PATH in a user-defined function or procedure is inherited according to the rules in Table 37 on page 151.

When CURRENT PACKAGE PATH or CURRENT PACKAGESET is set, DB2 uses the values in these registers to resolve the collection for a package. The value of CURRENT PACKAGE PATH takes priority over CURRENT PACKAGESET. In a distributed environment, the value of CURRENT PACKAGE PATH at the remote server takes precedence of the value of CURRENT PACKAGE PATH at the local server (the requester). For more information on package resolution, see *DB2 Application Programming and SQL Guide*.

Example: In an application that is using SQLJ packages (in collection SQLJ1 and SQLJ2) and a JDBC package in DB2JAVA, set the CURRENT PACKAGE PATH special register to check SQLJ1 first, followed by SQLJ2, and DB2JAVA:

```
SET CURRENT PACKAGE PATH = SQLJ1, SQLJ2, DB2JAVA;
```

The following statement sets the host variable to the value of the resulting list:

```
SET :HVPKLIST = CURRENT PACKAGE PATH;
```

The value of the host variable would be "SQLJ1", "SQLJ2", "DB2JAVA".

CURRENT PACKAGESET

CURRENT PACKAGESET specifies an empty string, a string of blanks, or the collection ID of the package that will be used to execute SQL statements.

The data type is VARCHAR(128).

The initial value of CURRENT PACKAGESET is an empty string. The value is a collection ID only if the application process has explicitly specified a collection ID by means of the SET CURRENT PACKAGESET statement. For details about this statement, see “SET CURRENT PACKAGESET” on page 1492. The initial value of CURRENT PACKAGESET in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

Example: Before passing control to another program, identify the collection ID for its package as ALPHA.

```
EXEC SQL SET CURRENT PACKAGESET = 'ALPHA';
```

CURRENT PATH

CURRENT PATH specifies the SQL path used to resolve unqualified data type names and function names in dynamically prepared SQL statements. It is also used to resolve unqualified procedure names that are specified as host variables in SQL CALL statements (CALL *host-variable*).

The data type is VARCHAR(2048).

The CURRENT PATH special register contains a list of one or more schema names, where each schema name is enclosed in delimiters and separated from the following schema by a comma (any delimiters within the string are repeated as they are in any delimited identifier). The delimiters and commas are included in the 2048 character length.

For information on when the SQL path is used to resolve unqualified names in both dynamic and static SQL statements and the effect of its value, see “SQL path” on page 56.

The initial value of the CURRENT PATH special register is either:

- The value of the PATH bind option
- The SQL PATH option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures
- "SYSIBM", "SYSFUN", "SYSPROC", "*value of CURRENT SQLID special register*" if the PATH bind option or SQL PATH option was not specified
- "SYSIBM", "SYSFUN", "SYSPROC", "*value of the role name that is associated with the user in the trusted context*" if the PATH bind option or SQL PATH option was not specified and if the connection is trusted with the role as object owner and qualifier options are in effect.

If the value of the CURRENT SQLID special register changes after the initial value of PATH special register is established, the value of the PATH special register is unaffected when the CURRENT SQLID is updated. However, if a commit later occurs and a SET PATH statement has not been processed, the value of PATH special register is reinitialized taking into consideration the current value of the CURRENT SQLID special register.

The initial value of CURRENT PATH in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the register by executing the statement SET PATH. For details about this statement, see “SET PATH” on page 1507. For portability across the platforms, it is recommended that a SET PATH statement be issued at the beginning of an application.

Example: Set the special register so that schema SMITH is searched before schemas SYSIBM, SYSFUN, and SYSPROC.

```
SET PATH = SMITH, SYSIBM, SYSFUN, SYSPROC;
```

CURRENT PRECISION

CURRENT PRECISION specifies the rules to be used when both operands in a decimal operation have precisions of 15 or less.

The data type of the register is CHAR(5).

Valid values for the CURRENT PRECISION special register include 'DEC15', 'DEC31', or 'Dpp.s' where 'pp' is either 15 or 31 and 's' is a number between 1 and 9. DEC15 specifies the rules that do not allow a precision greater than 15 digits, and DEC31 specifies the rules that allow a precision of up to 31 digits. The rules for DEC31 are always used if either operand has a precision greater than 15. If the form 'Dpp.s' is used, 'pp' represents the precision that will be used as the rules where DEC15 and DEC31 rules are used, and 's' represents the minimum divide

scale to use for division operations. The separator used in the form 'Dpp.s' can be either the '.' or the ',' character, regardless of the setting of the default decimal point.

The initial value of CURRENT PRECISION is determined by the value of field DECIMAL ARITHMETIC on installation panel DSNTIP4. The default for the initial value is DEC15 unless your installation has changed it to be DEC31 by modifying the value in that field. The initial value of CURRENT PRECISION in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the register by executing the statement SET CURRENT PRECISION. For details about this statement, see “SET CURRENT PRECISION” on page 1494.

CURRENT PRECISION only affects dynamic SQL. When an SQL statement is dynamically prepared and the value of CURRENT PRECISION is DEC15 or D15.s, where 's' is a number between 1 and 9, DEC15 rules will apply. When an SQL statement is dynamically prepared and the value of CURRENT PRECISION is DEC31 or D31.s, where 's' is a number between 1 and 9, DEC31 rules will apply. Preparation of a statement with DEC31 instead of DEC15 is more likely to result in an error, especially for division operations. Specification of CURRENT PRECISION in the form 'Dpp.s' where 'pp' is either 15 or 31 and 's' represents the minimum divide scale, will in some cases make division errors less likely when 'pp' is set to 31. For more information, see “Arithmetic with two decimal operands” on page 183.

Example 1: Set CURRENT PRECISION so that subsequent statements that are prepared use DEC31 rules for decimal arithmetic:

```
SET CURRENT PRECISION = 'DEC31';
```

Example 2: Set CURRENT PRECISION so that subsequent statements that are prepared use DEC31 rules for decimal arithmetic with a minimum divide scale of 3:

```
SET CURRENT PRECISION = 'D31.3';
```

CURRENT REFRESH AGE

CURRENT REFRESH AGE specifies a timestamp duration value. This duration is the maximum duration since a REFRESH TABLE statement has been processed on a system-maintained REFRESH DEFERRED materialized query table such that the materialized query table can be used to optimize the processing of a query. This special register affects dynamic statement cache matching.

The data type of the register is DECIMAL(20,6). For a description of durations, see “Datetime operands and durations” on page 192.

If CURRENT REFRESH AGE has a value of 9999999999999999 (ANY), REFRESH DEFERRED materialized query tables are considered to optimize the processing of a dynamic SQL query. This value represents 9999 years, 99 months, 99 days, 99 hours, and 99 seconds.

The initial value of CURRENT REFRESH AGE is determined by the value of field CURRENT REFRESH AGE on installation panel DSNTIP8. The default for the initial value of that field is 0 unless your installation has changed it to ANY by

modifying the value of that field. The initial value of CURRENT REFRESH AGE in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the register by executing the SET CURRENT REFRESH AGE statement. For details about this statement, see “SET CURRENT REFRESH AGE” on page 1495.

Example : The following example retrieves the current value of the CURRENT REFRESH AGE special register into the host variable, *CURMAXAGE*:

```
EXEC SQL VALUES (CURRENT REFRESH AGE) INTO :CURMAXAGE;
```

The value would be '99999999999999.000000'.

CURRENT ROUTINE VERSION

CURRENT ROUTINE VERSION specifies the version identifier that is to be used when invoking a native SQL procedure. CURRENT ROUTINE VERSION is used for CALL statements that use a host variable to specify the procedure name.

The data type of CURRENT ROUTINE VERSION is VARCHAR(64).

The initial value of CURRENT ROUTINE VERSION in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151. In other contexts the initial value of CURRENT ROUTINE VERSION is an empty string. An empty string indicates that a version identifier is not in effect for the SQL routine. When an SQL routine that does not have a version identifier in effect is invoked, the currently active version (as indicated in the catalog) of that routine is used.

You can change the value of the CURRENT ROUTINE VERSION by executing the statement SET CURRENT ROUTINE VERSION. For details about this statement, see “SET CURRENT ROUTINE VERSION” on page 1497.

Setting the CURRENT ROUTINE VERSION special register to a version identifier might affect native SQL procedures that are invoked until the value of CURRENT ROUTINE VERSION is changed. If a version of an SQL procedure has a version identifier that matches the version identifier in the special register, that version of the SQL procedure is used when the SQL procedure is invoked. If an SQL procedure does not have a version identifier that matches the version identifier in the special register, the currently active version of the SQL procedure (as defined in the catalog) is used when the SQL procedure is invoked.

Example: Set the host variable *ROUTINE_VER* to the value of the CURRENT ROUTINE VERSION special register:

```
VALUES CURRENT ROUTINE VERSION INTO :ROUTINE_VER;
```

CURRENT RULES

CURRENT RULES specifies whether certain SQL statements are executed in accordance with DB2 rules or the rules of the SQL standard.

The data type of the register is CHAR(3), and the only valid values are 'DB2' and 'STD'.

CURRENT RULES is a register at the database server. If the server is not the local DB2, the initial value of the register is 'DB2'. Otherwise, the initial value is the

same as the value of the SQLRULES bind option. The initial value of CURRENT RULES in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

You can change the value of the register by executing the statement SET CURRENT RULES. For details about this statement, see “SET CURRENT RULES” on page 1499.

CURRENT RULES affects the statements listed in the following table. The table summarizes when the statements are affected and shows where to find detailed information. CURRENT RULES also affects whether DB2 issues an existence error (SQLCODE -204) or an authorization error (SQLCODE -551) when an object does not exist. For CURRENT RULES 'STD', DB2 issues an authorization error (SQLCODE -551) when an object does not exist instead of the existence error (SQLCODE -204).

Table 36. Summary of statements affected by CURRENT RULES

Statement	What is affected	Details in topic
ALTER TABLE	Enforcement of check constraints added. Default value of the delete rule for referential constraints. Whether DB2 creates LOB table spaces, auxiliary tables, and indexes on auxiliary tables for added LOB columns. Whether DB2 creates an index for an added ROWID column that is defined with GENERATED BY DEFAULT.	“ALTER TABLE” on page 789
CREATE TABLE	Default value of the delete rule for referential constraints. Whether DB2 creates LOB table spaces, auxiliary tables, and indexes on auxiliary tables for LOB columns if the table is explicitly created. Whether DB2 creates an index for a ROWID column that is defined with GENERATED BY DEFAULT if the table is explicitly created.	“CREATE TABLE” on page 1079
DELETE	Authorization requirements for searched DELETE.	“DELETE” on page 1224
GRANT	Granting privileges to yourself.	“GRANT” on page 1333
REVOKE	Revoking privileges from authorization IDs	“REVOKE” on page 1432
UPDATE	Authorization requirements for searched UPDATE.	“UPDATE” on page 1521

Example: Set CURRENT RULES so that a later ALTER TABLE statement is executed in accordance with the rules of the SQL standard:

```
SET CURRENT RULES = 'STD';
```

CURRENT SCHEMA

The CURRENT SCHEMA special register specifies the schema name used to qualify unqualified database object references in dynamically prepared SQL statements.

The data type is VARCHAR(128).

For information on when the CURRENT SCHEMA is used to resolve unqualified names in dynamic SQL statements and the effect of its value, see “Qualification of unqualified object names” on page 57.

The CURRENT SCHEMA special register contains a value that is a single identifier without delimiters.

The initial value of the special register is the value of CURRENT SQLID at the time the connection is established. If the connection is established as a trusted connection with a role as the object owner and qualifier, the initial value of the special register is the value of the role name that is associated with the user in the trusted context. The initial value of the special register in a user-defined function or procedure is inherited according to the rules in Table 37 on page 151.

The value of the special register can be changed by executing the SET SCHEMA statement. The value of CURRENT SCHEMA is the same as the value of CURRENT SQLID unless a SET SCHEMA statement has been issued specifying a different value. After a SET SCHEMA statement has been issued in an application, the values of CURRENT SCHEMA and CURRENT SQLID are separate. Therefore, if the value of CURRENT SCHEMA needs to be changed, a SET SCHEMA statement must be issued.

Specifying CURRENT_SCHEMA is equivalent to specifying CURRENT SCHEMA.

Example: Set the schema for object qualification to 'D123'.

```
SET SCHEMA = 'D123'
```

CURRENT SERVER

CURRENT SERVER specifies the location name of the current server.

The data type is CHAR(16). If necessary, the location name is padded on the right with blanks so that its length is 16 bytes.

The initial value of CURRENT SERVER depends on the CURRENTSERVER bind option. If CURRENTSERVER X is specified on the bind subcommand, the initial value is X. If the option is not specified, the initial value is the location name of the local DB2. The initial value of CURRENT SERVER in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151. The value of CURRENT SERVER is changed by the successful execution of a CONNECT statement.

The value of CURRENT SERVER is a string of blanks when either of the following conditions apply:

- The application process is in the unconnected state

- The application process is connected to a local DB2 subsystem that does not have a location name.

Example: Set the host variable CS to the location name of the current server.

```
EXEC SQL SET :CS = CURRENT SERVER;
```

CURRENT SQLID

CURRENT SQLID specifies the SQL authorization ID of the process.

The data type is VARCHAR(128).

The SQL authorization ID is:

- The authorization ID used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym created by a dynamically issued CREATE statement.

The initial value of CURRENT SQLID can be provided by the connection or sign-on exit routine. If not, the initial value is the primary authorization ID of the process. The value remains in effect until one of the following events occurs:

- The SQL authorization ID is changed by the execution of a SET CURRENT SQLID statement.
- A SIGNON or re-SIGNON request is received from a CICS transaction subtask or an IMS independent region.
- The DB2 connection is ended.

The initial value of CURRENT SQLID in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

CURRENT SQLID can only be referred to in an SQL statement that is executed by the current server.

CURRENT SQLID cannot be a role.

Example: Set the SQL authorization ID to 'GROUP34' (one of the authorization IDs of the process).

```
SET CURRENT SQLID = 'GROUP34';
```

CURRENT TIME

The CURRENT TIME special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than one time within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.¹⁰

The value of CURRENT TIME in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151. For other applications, the time is derived by the DB2 that executes the SQL statement that refers to the special register. For a description of how the date is derived, see Datetime special registers.

10. Except for the case of a non-atomic multiple row INSERT or MERGE statement.

Specifying CURRENT_TIME is equivalent to specifying CURRENT TIME.

Example: Display information about all project activities and include the current date and time in each row of the result.

```
SELECT DSN8910.PROJACT.*, CURRENT DATE, CURRENT TIME
FROM DSN8910.PROJACT;
```

CURRENT TIMESTAMP

The CURRENT_TIMESTAMP special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

If this special register is used more than one time within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading.¹¹

The value of CURRENT_TIMESTAMP in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151. For other applications, the timestamp is derived by the DB2 that executes the SQL statement that refers to the special register. For a description of how the date is derived, see Datetime special registers.

Specifying CURRENT_TIMESTAMP is equivalent to specifying CURRENT TIMESTAMP.

Example 1: Display information about the full image copies that were taken in the last week.

```
SELECT * FROM SYSIBM.SYSCOPY
WHERE TIMESTAMP > CURRENT TIMESTAMP - 7 DAYS;
```

CURRENT TIMEZONE

CURRENT_TIMEZONE specifies the TIMEZONE parameter in the form of a time duration.

The data type is DECIMAL(6,0).

The time duration is derived by the DB2 that executes the SQL statement that refers to the special register. The seconds part of the time duration is always zero. An error occurs if the hours portion of the TIMEZONE parameter is not between -24 and 24. The value of CURRENT_TIMEZONE in a user-defined function or stored procedure is inherited according to the rules in Table 37 on page 151.

Example: Display information from SYSCOPY, but with the TIMESTAMP converted to GMT. This example is based on the assumption that the installation sets the clock to GMT and the TIMEZONE parameter to the difference from GMT.

```
SELECT DBNAME, TSNAME, DSNUM, ICTYPE, TIMESTAMP - CURRENT TIMEZONE
FROM SYSIBM.SYSCOPY;
```

ENCRYPTION PASSWORD

The ENCRYPTION_PASSWORD special register specifies the encryption password and the password hint (if one exists) that are used by the encryption and decryption built-in functions.

11. Except for the case of a non-atomic multiple row INSERT or MERGE statement.

This special register can only be set, by using the SET ENCRYPTION PASSWORD statement, and cannot be referenced directly. The ENCRYPTION PASSWORD special register contains the value of the password that is used by the ENCRYPTION and DECRYPTION built-in functions to encrypt and decrypt data when a password is not explicitly specified as a function argument. The ENCRYPTION PASSWORD special register can also contain a password hint which is associated with the values that are encrypted using the encryption password. The password hint is a character string that is used to help in remembering the password. The GETHINT function is used to return the password hint for an encrypted value.

The initial value of the ENCRYPTION PASSWORD special register is the empty string (' ').

The initial value of the ENCRYPTION PASSWORD special register in a user-defined function or procedure is inherited from the invoking application. In other contexts, the initial value of the special register is the empty string.

The password is not related to DB2 authentication and is used only for data encryption. For more information, see “SET ENCRYPTION PASSWORD” on page 1502.

SESSION_USER

SESSION_USER specifies the primary authorization ID of the process.

The data type is VARCHAR(128).

If SESSION_USER is referred to in an SQL statement that is executed at a remote DB2 and the primary authorization ID has been translated to a different authorization ID, SESSION_USER specifies the translated authorization ID. For an explanation of authorization ID translation, see *DB2 Administration Guide*. The value of SESSION_USER in a user-defined function or stored procedure is determined according to the rules in Table 37 on page 151.

USER can be specified as a synonym for SESSION_USER.

Example: Display information about tables, views, and aliases that are owned by the primary authorization ID of the process.

```
SELECT * FROM SYSIBM.SYSTABLES WHERE CREATOR = SESSION_USER;
```

USER

USER specifies the primary authorization ID of the process. The data type is VARCHAR(128). SESSION_USER is the preferred spelling.

If USER is referred to in an SQL statement that is executed at a remote DB2 and the primary authorization ID has been translated to a different authorization ID, USER specifies the translated authorization ID. For an explanation of authorization ID translation, see *DB2 Administration Guide*. The value of USER in a user-defined function or stored procedure is determined according to the rules in Table 37 on page 151.

Example: Display information about tables, views, and aliases that are owned by the primary authorization ID of the process.

```
SELECT * FROM SYSIBM.SYSTABLES WHERE CREATOR = USER;
```

Using special registers in a user-defined function or a stored procedure

You can use all special registers in a user-defined function or a stored procedure. However, you can modify only some of those special registers.

After a user-defined function or a stored procedure completes, DB2 restores all special registers to the values they had before invocation.

The following table shows information you need when you use special registers in a user-defined function or stored procedure.

Table 37. Characteristics of special registers in a user-defined function or a stored procedure

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT APPLICATION ENCODING SCHEME	The value of bind option ENCODING for the user-defined function or stored procedure package	The value of bind option ENCODING for the user-defined function or stored procedure package	Yes
CURRENT CLIENT_ACCTNG	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT CLIENT_APPLNAME	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT CLIENT_USERID	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT CLIENT_WRKSTNNAME	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT DATE	New value for each SQL statement in the user-defined function or stored procedure package ¹	New value for each SQL statement in the user-defined function or stored procedure package ¹	Not applicable ⁵
CURRENT DEBUG MODE	Inherited from the invoking application	DISALLOW	Yes
CURRENT DECFLOAT ROUNDING MODE	Inherited from the invoking application	The value of bind option ROUNDING for the user-defined function or stored procedure package	Yes
CURRENT DEGREE	CURRENT DEGREE ²	The value of field CURRENT DEGREE on installation panel DSNTIP8	Yes
CURRENT LOCALE LC_CTYPE	Inherited from the invoking application	The value of field CURRENT LC_CTYPE on installation panel DSNTIPF	Yes
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Inherited from the invoking application	System default value	Yes
CURRENT MEMBER	New value for each SET <i>host-variable</i> =CURRENT MEMBER statement	New value for each SET <i>host-variable</i> =CURRENT MEMBER statement	Not applicable ⁵

Table 37. Characteristics of special registers in a user-defined function or a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT OPTIMIZATION HINT	The value of bind option OPTHINT for the user-defined function or stored procedure package or inherited from the invoking application ⁶	The value of bind option OPTHINT for the user-defined function or stored procedure package	Yes
CURRENT PACKAGE PATH	An empty string if the routine was defined with a COLLID value; otherwise, inherited from the invoking application ⁴	An empty string, regardless of whether a COLLID value was specified for the routine ⁴	Yes
CURRENT PACKAGESET	Inherited from the invoking application ³	Inherited from the invoking application ³	Yes
CURRENT PATH	The value of bind option PATH for the user-defined function or stored procedure package or inherited from the invoking application ⁶	The value of bind option PATH for the user-defined function or stored procedure package	Yes
CURRENT PRECISION	Inherited from the invoking application	The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4	Yes
CURRENT REFRESH AGE	Inherited from the invoking application	System default value	Yes
CURRENT ROUTINE VERSION	Inherited from the invoking application	The empty string	Yes
CURRENT RULES	Inherited from the invoking application	The value of bind option SQLRULES for the plan that invokes a user-defined function or stored procedure	Yes
CURRENT SCHEMA	Inherited from the invoking application	The value of CURRENT SCHEMA when the routine is entered	Yes
CURRENT SERVER	Inherited from the invoking application	Inherited from the invoking application	Yes
CURRENT SQLID	The primary authorization ID of the application process or inherited from the invoking application ⁷	The primary authorization ID of the application process	Yes ⁸
CURRENT TIME	New value for each SQL statement in the user-defined function or stored procedure package ¹	New value for each SQL statement in the user-defined function or stored procedure package ¹	Not applicable ⁵
CURRENT TIMESTAMP	New value for each SQL statement in the user-defined function or stored procedure package ¹	New value for each SQL statement in the user-defined function or stored procedure package ¹	Not applicable ⁵
CURRENT TIMEZONE	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
ENCRYPTION PASSWORD	Inherited from the invoking application	Inherited from the invoking application	Yes

Table 37. Characteristics of special registers in a user-defined function or a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
SESSION_USER or USER	Primary authorization ID of the application process	Primary authorization ID of the application process	Not applicable ⁵

Notes:

1. If the user-defined function or stored procedure is invoked within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement as the timestamp for all SQL statements in the package.
2. DB2 allows parallelism at only one level of a nested SQL statement. If you set the value of the CURRENT DEGREE special register to ANY, and parallelism is disabled, DB2 ignores the CURRENT DEGREE value.
3. If the routine definition includes a specification for COLLID, DB2 sets CURRENT PACKAGESET to the value of COLLID. If both CURRENT PACKAGE PATH and COLLID are specified, the CURRENT PACKAGE PATH value takes precedence and COLLID is ignored.
4. If the function definition includes a specification for PACKAGE PATH, DB2 sets CURRENT PACKAGE PATH to the value of PACKAGE PATH.
5. Not applicable because no SET statement exists for the special register.
6. If a program within the scope of the invoking program issues a SET statement for the special register before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, the special register contains the value that is set by the bind option for the user-defined function or stored procedure package.
7. If a program within the scope of the invoking program issues a SET CURRENT SQLID statement before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, CURRENT SQLID contains the authorization ID of the application process.
8. If the user-defined function or stored procedure package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed but does not affect the authorization ID that is used for the dynamic SQL statements in the package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements. For more information, see the discussion of DYNAMICRULES in *DB2 Command Reference*.

Column names

The meaning of a column name depends on its context.

A column name can be used to:

- Declare the name of a column, as in a CREATE TABLE statement.
- Specify the name of a column, as in a CREATE FUNCTION statement to name a column of the result table of a table function.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
 - In an *aggregate function*, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. (Groups and intermediate result tables are explained in Chapter 4, “Queries,” on page 629.) For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
 - In a *GROUP BY* or *ORDER BY* clause, a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.
 - In an *expression*, a *search condition*, or a *scalar function*, a column name specifies a value for each row or group to which the construct is applied. For example,

when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.

- Provide a column name for an expression to temporarily rename a column, as in the *correlation-clause* of a *table-reference* in a FROM clause or as in the AS clause in the *select-clause*.

Qualified column names

A qualifier for a column name can be a table name, a view name, an alias name, a synonym, or a correlation name. Whether a column name can be qualified depends, like its meaning, on its context.

- In some forms of the COMMENT and LABEL statements, a column name must be qualified. This is shown in the syntax diagrams.
- Where the column name specifies values of the column, a column name can be qualified at the user's option.
- In the column list of an INSERT statement, a column name can be qualified.
- In the *assignment-clause* of an UPDATE or a MERGE statement, a column name can be qualified.
- In all other contexts, a column name must not be qualified. This rule will be mentioned in the discussion of each statement to which it applies.

Where a qualifier is optional, it can serve two purposes. See “Column name qualifiers to avoid ambiguity” on page 155 and “Column name qualifiers in correlated references” on page 157 for details.

Correlation names

A *correlation name* can be defined in the FROM clause of a query and after the name of the target table or view in an UPDATE, MERGE, or DELETE statement.

For example, the following clause establishes Z as a correlation name for X.MYTABLE:

```
FROM X.MYTABLE Z
```

With Z defined as a correlation name for table X.MYTABLE, only Z should be used to qualify a reference to a column of X.MYTABLE in that SELECT statement.

A correlation name is associated with a table, view, nested table expression or table function only within the context in which it is defined. Hence, the same correlation name can be defined for different purposes in different statements. In a nested table expression or table function, a correlation name is required.

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. It can also be used merely as a shorter name for a table or view. In the example, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

Names that are specified in a FROM clause are either *exposed* or *non-exposed*. A correlation name is always an exposed name. A table name or view name is said to be exposed in that FROM clause if a correlation name is not specified. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE, but not for DEPARTMENT; therefore, DEPARTMENT is an exposed name, and EMPLOYEE is not an exposed name:

```
FROM EMPLOYEE E, DEPARTMENT
```

The use of a correlation name in the FROM clause also allows the option of specifying a list of column names to be associated with the columns of the result table. As with a correlation name, the listed column names should be the names that are used to reference the columns in that SELECT statement. For example, assume that the name of the first column in the DEPT table is DEPTNO. Given this FROM clause in a SELECT statement:

```
FROM DEPT D (NUM,NAME,MGR,ANUM,LOC)
```

You should use D.NUM instead of D.DEPTNO to reference the first column of the table.

If a list of columns is specified, it must consist of as many names as there are columns in the *table-reference*. Each column must be unique and unqualified.

Column name qualifiers to avoid ambiguity

In the context of a function, a GROUP BY clause, an ORDER BY clause, an expression, or a search condition, a column name refers to values of a column in some table or view in a DELETE or UPDATE statement or *table-reference* in a FROM clause.

The tables, views, and *table-references*¹² that might contain the column are called the *object tables* of the context. Two or more object tables might contain columns with the same name. One reason for qualifying a column name is to designate the object from which the column comes. For information on avoiding ambiguity between SQL parameters and variables and column names, see “References to SQL parameters and SQL variables” on page 1611.

A nested table expression which is preceded by a TABLE keyword will consider *table-references* that precede it in the FROM clause as object tables. The *table-references* that follow it are not considered as object tables.

Table designators: A qualifier that designates a specific object table is called a *table designator*. The clause that identifies the object tables also establishes the table designators for them. For example, the object tables of an expression in a SELECT statement are named in the FROM clause that follows it, as in the following statement:

```
SELECT DISTINCT Z.EMPNO, EMPTIME, PHONENO
FROM DSN8910.EMP Z, DSN8910.EMPPROJACT
WHERE WORKDEPT = 'D11'
AND EMPTIME > 0.5
AND Z.EMPNO = DSN8910.EMPPROJACT.EMPNO;
```

Table designators in the FROM clause are established as follows:

- A name that follows a table or view name is both a correlation name and a table designator. Thus, Z is a table designator and qualifies the first column name in the select list.
- An exposed table or view name is a table designator. Thus, the qualified table name, DSN8910.EMPPROJACT is a table designator and qualifies the second column name in the select list.

Two or more object tables can be instances of the same table. In this case, distinct correlation names must be used to unambiguously designate the particular

12. In the case of a *joined-table*, each *table-reference* within the *joined-table* is an object table.

instance of the table. In the following example, the X and Y in the FROM clause are defined to refer, respectively, to the first and second instances of the DSN8910.EMP table:

```
SELECT *
FROM DSN8910.EMP X, DSN8910.EMP Y;
```

Avoiding undefined or ambiguous references in DB2 SQL: When a column name refers to values of a column, the following situations result in errors:

- No object table contains a column with the specified name. The reference is undefined.
- The column name is qualified by a table designator, but the table named does not include a column with the specified name. Again, the reference is undefined.
- The name is unqualified and more than one object table includes a column with that name. The reference is ambiguous.

Avoid ambiguous references by qualifying a column name with a uniquely defined table designator. If the column is contained in several object tables with different names, the table names can be used as designators. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

Two or more object tables can be instances of the same table. A FROM clause that includes n references to the same table should include at least $n - 1$ unique correlation names.

For example, in the following FROM clause X and Y are defined to refer, respectively, to the first and second instances of the table EMP.

```
SELECT X.LASTNAME, Y.LASTNAME
FROM DSN8910.EMP X, DSN8910.EMP Y
WHERE Y.JOB = 'MANAGER'
AND X.WORKDEPT = Y.WORKDEPT
AND X.JOB <> 'MANAGER';
```

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name can be used. However, the qualifier used and the table used must be the same after fully qualifying the table name or view name and the table designator.

Example 1: If the authorization ID of the statement is CORPDATA, the following statement is valid:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
FROM EMPLOYEE;
```

Example 2: If the authorization ID of the statement is REGION, the following statement is invalid because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT           -- Incorrect
FROM EMPLOYEE;
```

Example 3: If the authorization ID of the statement is REGION, the following statement is invalid because EMPLOYEE in the select list represents the table REGION.EMPLOYEE, but the explicitly qualified table name in the FROM clause represents a different table, CORPDATA.EMPLOYEE.


```
SELECT EMPLOYEE.WORKDEPT
FROM CORPDATA.EMPLOYEE;
```

-- Incorrect

Column name qualifiers in correlated references

A reference to a column of a table identified at a higher level is called a *correlated reference*. Because the same table or view can be identified at many levels, unique correlation names are recommended as table designators. It is good practice to use these unique correlation names to qualify column names.

A *subselect* is a form of a query that can be used as a component of various SQL statements. Refer to Chapter 4, “Queries,” on page 629 for more information on subselects. A *subquery* is a form of a fullselect that is enclosed within parenthesis. For example, a subquery can be used in a search condition. A fullselect that is used to retrieve a single value as an expression within a statement is called a *scalar fullselect* or a *scalar subquery*. A fullselect that is used in the FROM clause of a query is called a *nested table expression*.

A subquery can include search conditions of its own, and these search conditions can, in turn, include subqueries. Thus, an SQL statement can contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are said to be at a higher level than the subqueries they contain.

Every element of the hierarchy has a clause that establishes one or more table designators. This is the FROM clause, except in the highest level of an UPDATE, where it is the table or view being updated. A search condition of a subquery can reference not only columns of the tables identified by the FROM clause of its own element of the hierarchy, but also columns of tables identified at any level along the path from its own element to the highest level of the hierarchy. A reference to a column of a table identified at a higher level is called a *correlated reference*.

A correlated reference to column C of table T can be of the form C, T.C, or Q.C, if Q is a correlation name defined for T. However, **a correlated reference in the form of an unqualified column name is not good practice**. The following explanation is based on the assumption that a correlated reference is always in the form of a qualified column name and that the qualifier is a correlation name.

A qualified column name, Q.C, is a correlated reference only if these three conditions are met:

- Q.C is used in a search condition or in a select list of a subquery.
- Q does not name a table used in the FROM clause of that subquery.
- Q does name a table used at some higher level.

Q.C refers to column C of the table or view at the level where Q is used as the table designator of that table or view. Because the same table or view can be identified at many levels, unique correlation names are recommended as table designators. If Q is used to name a table at more than one level, Q.C refers to the lowest level that contains the subquery that includes Q.C.

For example, in the following statement, the correlated reference X.WORKDEPT (in the last line) refers to the value of WORKDEPT in table DSN8910.EMP at the level of the first FROM clause (which establishes X as a correlation name for DSN8910.EMP). The statement lists employees who make less than the average salary for their department.

```

SELECT EMPNO, LASTNAME, WORKDEPT
FROM DSN8910.EMP X
WHERE SALARY < (SELECT AVG(SALARY)
                FROM DSN8910.EMP
                WHERE WORKDEPT = X.WORKDEPT);

```

The following example shows a correlated reference in the select list of the subquery.

```

SELECT T1.KEY1
FROM BP1TBL T1
GROUP BY T1.KEY1
HAVING MAX(T1.KEY1) = (SELECT MIN(T1.KEY1) + MIN(T2.KEY1)
                      FROM BP2TBL T2);

```

Resolution of column name qualifiers and column names

The rules for resolving column name qualifiers apply to every SQL statement that includes a subselect and are applied before synonyms and aliases are resolved.

Names in a FROM clause are either *exposed* or *non-exposed*. A correlation name for a table name, view name, nested table expression, or reference to a table function is always exposed. A table name or a view name that is not followed by a correlation name is also exposed.

Although DB2 for z/OS does not enforce this rule strictly, in IBM SQL and ANSI/ISO SQL, the exposed names in a FROM clause must be unique, and the qualifier of a column name must be an exposed name. Therefore, for good programming practices, ensure that all exposed names are unique and that all qualified column names are qualified with the appropriate exposed name.

The rules for finding the referent of a column name qualifier are as follows:

1. Let Q be a one-, two-, or three-part name, and let Q.C denote a column name in subselect S. Q must designate a table or view identified in the statement that includes S and that table or view must have a column named C. An additional requirement differs for two cases:
 - If Q.C is not in a *search-condition* or S is not a subquery, Q must designate a table or view identified in the FROM clause of S. For example, if Q.C is in a SELECT clause, Q refers to a table or view in the following FROM clause.
 - If Q.C is in a *search-condition* and S is a subquery, Q must designate a table or view identified either in the FROM clause of S or in a FROM clause of a subselect that directly or indirectly includes S. For example, if Q.C is in a WHERE clause and S is the only subquery in the statement, the table or view that Q refers to is either in the FROM clause of S or the FROM clause of the subselect that includes S.
2. The same table or view can be identified more than once in the same statement. The particular occurrence of the table or view that Q refers to is determined by a procedure equivalent to the following steps:
 - a. The one- and two-part names in every FROM clause and the one- and two-part qualifiers of column names are expanded into a fully-qualified form.

For example, if a dynamic SQL statement uses FROM Q and DYNAMICRULES run behavior (RUN) is in effect, Q is expanded to S.A.Q, where S is the value of CURRENT SERVER and A is the value of CURRENT SCHEMA. (If DYNAMICRULES bind behavior is in effect instead, A is the plan or package qualifier as determined during the bind process or the qualifier for the native SQL procedure as determined when

the procedure was defined.) This step is later referred to as “name completion”. An error occurs if the first part of every name (the location) is not the same.

- b. Q, now a three-part name, is compared with every name in the FROM clause of S. If Q.C is in a *search-condition* and S is a subquery, Q is next compared with every name in the FROM clause of the subselect that contains S. If that subselect is a subquery, Q is then compared with every name in the FROM clause of the subselect containing that subquery, and so on. If a FROM clause includes multiple names, the comparisons in that clause are made in order from left to right.
 - c. The referent of Q is selected by these rules:
 - If Q matches exactly one name, that name is selected.
 - If Q matches more than one name, but only one exposed name, that exposed name is selected.
 - If Q matches more than one exposed name, the first of those names is selected.
 - If Q matches more than one name, none of which are exposed names, the first of those names is selected.If Q does not match any name, or if the table or view designated by Q does not include a column named C, an error occurs.
 - d. Otherwise, Q.C is resolved to column C of the occurrence of the table or view identified by the selected name.
3. A warning occurs for any of these cases:
- The selected name is not an exposed name.
 - The selected name is an exposed name that has an unexposed duplicate that appears before the selected name in the ordered list of names to which Q is compared.
 - The selected name is an exposed name that has an exposed duplicate in the same FROM clause.
 - Another name would have been selected had the matching been performed before name completion.

The rules for resolving column name qualifiers apply to every SQL statement that includes a subselect and are applied before synonyms and aliases are resolved. In the case of a searched UPDATE or DELETE statement, the first clause of the statement identifies the table or view to be updated or deleted. That clause can include a correlation name and, with regard to name resolution, is equivalent to the first FROM clause of a SELECT statement. For example, a subquery in the search condition of an UPDATE statement can include a correlated reference to a column of the updated rows.

The rules for column names in the ORDER BY clause are the same as other clauses.

References to variables

A *variable* in an SQL statement specifies a value that can be changed when the SQL statement is executed. There are several types of variables used in SQL statements.

host variable

Host variables are defined by statements of a host language. For more information about how to refer to host variables, see “References to host variables” on page 160.

transition variable

Transition variables are defined in a trigger and refer to either the old or new values of columns of the subject table of a trigger. For more information about how to refer to transition variables, see “CREATE TRIGGER” on page 1151.

SQL variable

SQL variables are defined by an SQL compound statement in an SQL procedure. For more information about SQL variables, see “References to SQL parameters and SQL variables” on page 1542.

SQL parameter

SQL parameters are defined in an CREATE FUNCTION (SQL Scalar) or CREATE PROCEDURE (SQL) statement. For more information about SQL parameters, see “References to SQL parameters and SQL variables” on page 1542.

parameter marker

Parameter markers are specified in an SQL statement that is dynamically prepared instead of host variables. For more information about parameter markers, see Parameter markers. in the PREPARE statement.

Unless otherwise noted, the term *host variable* in syntax diagrams is used to describe where a host variable, transition variable, SQL variable, SQL parameter, or parameter marker can be used.

References to host variables

Host variables are defined directly by statements of the host language or indirectly by SQL extensions. A *host-variable* in an SQL statement must identify a host variable that is described in the program according to the rules for declaring host variables. Host variables cannot be referenced in dynamic SQL statements; parameter markers must be used instead.

A *host variable* is either of these items that is referred to in an SQL statement:

- A variable in a host language such as a PL/I variable, C variable, Fortran variable, REXX variable, Java variable, COBOL data item, or Assembler language storage area
- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions

Host variables are defined directly by statements of the host language or indirectly by SQL extensions as described in *DB2 Application Programming and SQL Guide*. Host variables cannot be referenced in dynamic SQL statements; parameter markers must be used instead. For more information about parameter markers, see “Host variables in dynamic SQL” on page 162.

A host-variable in an SQL statement must identify a host variable that is described in the program according to the rules for declaring host variables.

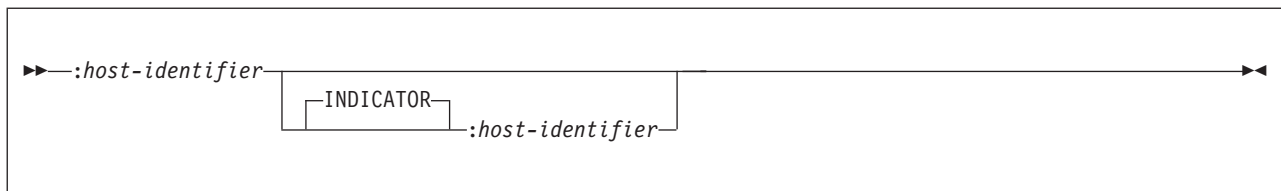
In PL/I, C, and COBOL, host variables can be referred to in ways that do not apply to Fortran and Assembler language. This is explained in “Host structures in PL/I, C, and COBOL” on page 171. The following applies to all host languages.

The term *host-variable*, as used in the syntax diagrams, shows a reference to a host variable. In a SET *host variable* statement and the INTO clause of a FETCH, SELECT INTO, or VALUES INTO statement, a host variable is an output variable

to which a value is assigned by DB2. In a CALL statement, a host variable can be an output argument that is assigned a value after execution of the procedure, an input argument that provides an input value for the procedure, or both an input and output argument. In all other contexts, a host variable is an input variable which provides a value to DB2.

Non-Java variable references

The general form of a host variable reference in all languages other than Java is:



Each host identifier must be declared in the source program, except in a program written in REXX. The first host identifier designates the main variable; the second host identifier designates its indicator variable. The variable designated by the second host identifier must be a small integer. The purposes of the indicator variable are to:

- Specify the null value. A negative value of the indicator variable specifies the null value. A -1 value indicates that the value that was selected was the null value. A -2 value indicates that a numeric conversion or arithmetic expression error occurred in the SELECT list of an outer SELECT statement. A -3 value indicates that values were not returned for the row because a hole was detected on a multiple row FETCH. The value of -3 is used only for multiple-row FETCH statements; otherwise, the only indication of a hole is a warning.
- Record the original length of a truncated string.
- Indicate that a character could not be converted.
- Record the seconds portion of a time if the time is truncated on assignment to a host variable.

For example, if `:V1:V2` is used to specify an insert or update value, and if `V2` is negative, the value specified is the null value. If `V2` is not negative, the value specified is the value of `V1`.

Similarly, if `:V1:V2` is specified in certain statements (like a FETCH or SELECT INTO statement), and if the value returned is null, `V1` is not changed and `V2` is set to -1 or -2. It is set to -1 if the value selected was actually null. It is set to -2 if the null value was returned because of numeric conversion errors or arithmetic expression errors in the SELECT list of an outer SELECT statement. It is also set to -2 as the result of a character conversion error.

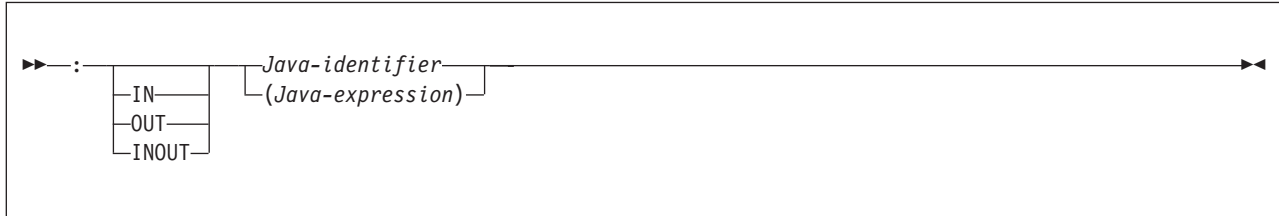
If the value returned is not null, that value is assigned to `V1`, and `V2` is set to zero (unless the assignment to `V1` requires string truncation, in which case `V2` is set to the original length of the string). If an assignment requires truncation of the seconds part of a time, `V2` is set to the number of seconds.

If the second host identifier is omitted, the host variable does not have an indicator variable: the value specified by the host variable `:V1` is always the value of `V1` and null values cannot be assigned to the variable. Thus, this form should not be used in an INTO clause unless the corresponding

result column cannot contain null values. If this form is used for an output host variable and the value returned is null, DB2 will generate an error at run time.

Java variable references

The general form of a host variable reference in Java is:



In Java, indicator variables are not used. Instead, instances of a Java class can be set to a null value. Variables defined as Java primitive types can not be set to a null value.

If IN, OUT, or INOUT is not specified, the default depends on the context in which the variable is used. If the Java variable is used in an INTO clause, OUT is the default. Otherwise, IN is the default.

An SQL statement that refers to host variables must be within the scope of the declaration of those host variables. For host variables referred to in the SELECT statement of a cursor, the OPEN statement, and the DECLARE CURSOR statement have to be in the same scope.

| All references to host variables must be preceded by a colon. If an SQL statement
 | references a host variable without a preceding colon, the precompiler issues an
 | error for the missing colon or interprets the host variable as an unqualified column
 | name, which might lead to unintended results. The interpretation of a host variable
 | without a colon as a column name occurs when the host variable is referenced in a
 | context in which a column name can also be referenced.

Host variables in dynamic SQL

In dynamic SQL statements, parameter markers are used instead of host variables.

A *parameter marker* is a question mark (?) that represents a position in a dynamic SQL statement where the application will provide a value; that is, where a host variable would be found if the statement string were a static SQL statement. The following examples show a static SQL statement that uses host variables and a dynamic statement that uses parameter markers:

```

  INSERT INTO DEPT VALUES (:HV_DEPTNO, :HV_DEPTNAME, :HV_MGRNO, :HV_ADMRDEPT)
  INSERT INTO DEPT VALUES (?, ?, ?, ?)
  
```

For more information on parameter markers, see Parameter markers under the PREPARE statement.

References to LOB host variables

| Regular LOB variables (CLOB, DBCLOB, and BLOB), LOB locator variables and
 | LOB file reference variables can be defined in all host languages with a few
 | exceptions.

Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these variables are not native data types in host programming languages, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

- REXX supports LOB locators and LOB file reference variables.
- Java supports LOBs and LOB file references, but not LOB locators.

When it is possible to define a host variable that is large enough to hold an entire LOB value and the performance benefit of delaying the transfer of data from the server is not required, a LOB locator or LOB file reference is not needed. However, it is often not acceptable to store an entire LOB value in temporary storage due to host language restrictions, storage restrictions, or performance requirements. When storing an entire LOB value at one time is not acceptable, a LOB value can be referenced using a LOB locator and portions of the LOB value can be accessed or the entire LOB value can be stored in a file and a LOB file reference can be used to access the LOB data.

References to LOB locator variables

A LOB locator variable is a host variable that contains the locator representing a LOB value on the database server.

A locator variable in an SQL statement must identify a LOB locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement. For example, in C:

```
static volatile SQL TYPE IS CLOB_LOCATOR *loc1;
```

The term *locator-variable*, as used in the syntax diagrams, shows a reference to a LOB locator variable. The meta-variable *locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

Like all other host variables, a LOB locator variable can have an associated indicator variable. Indicator variables for LOB locator variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the locator host variable is unchanged. This means a locator can never represent a null value. However, when the indicator variable associated with a LOB locator is null, the value of the referenced LOB value is null.

If a locator variable does not currently represent any value, an error occurs when the locator variable is referenced.

When a transaction commits, LOB locators that were acquired by the transaction are released unless a HOLD LOCATOR statement was issued for the LOB locator. When the transaction ends, all LOB locators are released.

It is the application programmer's responsibility to guarantee that any LOB locator is used only in SQL statements that are executed at the same server that originally generated the LOB locator. For example, assume that a LOB locator is returned from one server and assigned to a LOB locator variable. If that LOB locator variable is subsequently used in an SQL statement that is executed at a different server unpredictable results will occur.

References to XML host variables

XML host variables can be declared as another host variable type. This allows XML data to be declared in host languages that do not support XML data as a native data type.

XML host variables can be declared as the following host variable types:

- XML AS CLOB(*n*)
Declares a CLOB host variable that contains XML data that is encoded in the CCSID for the host variable.
- XML AS DBCLOB(*n*)
Declares a DBCLOB host variable that contains XML data that is encoded in the graphic CCSID for the host variable.
- XML AS BLOB(*n*)
Declares a BLOB host variable that contains XML data that is encoded as specified within the data according to the XML 1.0 specification for determining encoding.
- XML AS CLOB_FILE
Declares a CLOB file reference variable that contains XML data that is encoded in the CCSID for the file reference variable.
- XML AS DBCLOB_FILE
Declares a DBCLOB file reference variable that contains XML data that is encoded in the CCSID for the file reference variable.
- XML AS BLOB_FILE
Declares a BLOB file reference variable that contains XML data that is encoded in the CCSID for the file reference variable.

See “References to file reference variables” on page 165 for additional information about file reference variables.

Although the application XML host variable declaration includes a LOB type specification, the host variable declarations all map to the XML data type, not the LOB type that is used in the application declaration. The application might also use non-XML host variables in place of XML host variables. For example, when a prepared statement is executed, the application might use a character host variable to replace an XML parameter marker in the statement.

Although the XML data type is incompatible with all other data types, both XML and non-XML data types can be used for input to and output from XML data. Applications can use either XML host variables, character host variables, or binary string host variables to input to and output from XML data by using SQL statements.

The following table summarizes the conversions built-in data types (including XML) to and from the supported host variable data types within embedded applications. The built-in data types are specified in the rows. A Y indicates that the built-in data type can be assigned to or assigned from the host variable type.

Table 38. Application host variable compatibility with the built-in data types for applications that contain embedded SQL

built-in data type	application host variable data type					
	CHAR, VARCHAR, CLOB, CLOB_FILE	GRAPHIC, VARGRAPHIC, DBCLOB, DBCLOB_FILE	BINARY, VARBINARY, BLOB, BLOB_FILE	XML AS CLOB, XML AS CLOB_FILE	XML AS DBCLOB, XML AS DBCLOB_FILE	XML AS BLOB, XML AS BLOB_FILE
CHAR	Y	Y				
VARCHAR	Y	Y				
CLOB	Y	Y				
GRAPHIC	Y	Y				
VARGRAPHIC	Y	Y				
DBCLOB	Y	Y				
BINARY			Y			
VARBINARY			Y			
BLOB			Y			
XML	Y	Y	Y	Y	Y	Y

References to file reference variables

File reference variables and arrays are used for direct file input and output for LOB and XML values (when the XML value is declared using XML AS *variable-type*), and can be defined in all host languages.

Since these are not native data types, SQL extensions are used and the DB2 precompiler or coprocessor generates the host language constructs necessary to represent each variable or array. In the case of REXX, LOB values are mapped to strings. See “References to XML host variables” on page 164 for more information about XML host variables.

A file reference variable represents (rather than contains) the file, just as a LOB locator represents, rather than contains, the LOB data. Database queries, updates, and inserts can use file reference variables to store or to retrieve single column values. As with all other host variables, file reference variables can have an associated indicator variable.

A file reference variable has the following properties:

Data type

BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared using BLOB_FILE, CLOB_FILE, or DBCLOB_FILE.

Direction

This must be specified by the application program at run time (it is implicitly specified as part of the File options value). The direction can be either of the following:

- Input
Input is used as a source of data on an EXECUTE statement, an OPEN statement, an update operation, an insert operation, or a delete operation.
- Output
Output is used as the target of data. For example, on a FETCH statement or a SELECT INTO statement.

File name

This must be specified by the application program at run time. It must be the complete path name of the file. Within an application, a file should only be referenced one time in a file reference variable.

File name length

This must be specified by the application program at run time. It is the length of the file name in bytes.

Data length

Sets the data length to the length of the new data that is written to the file. The length is in bytes. Data length is unused on input.

File options

Options are set by an INTEGER value in a field in the file reference variable structure. One of the following values must be specified in an application for each file reference variable before that file reference variable can be used in the application:

SQL_FILE_READ

This is a regular file that can be opened, read and closed. (The option is SQL-FILE-READ in COBOL, `sql_file_read` in FORTRAN, and READ in REXX.) SQL_FILE_READ is an input (from client to server) file option.

SQL_FILE_CREATE

Create a new file. If the file already exists, an error is returned. (The option is SQL-FILE-CREATE in COBOL, `sql_file_create` in FORTRAN, and CREATE in REXX.) SQL_FILE_CREATE is an output (from server to client) file option.

SQL_FILE_OVERWRITE

If an existing file with the specified name exists, it is overwritten; otherwise a new file is created. (The option is SQL-FILE-OVERWRITE in COBOL, `sql_file_overwrite` in FORTRAN, and OVERWRITE in REXX.) SQL_FILE_OVERWRITE is an output (from server to client) file option.

SQL_FILE_APPEND

If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created. (The option is SQL-FILE-APPEND in COBOL, `sql_file_append` in FORTRAN, and APPEND in REXX.) SQL_FILE_APPEND is an output (from server to client) file option.

The encoding scheme CCSID of the file name is based on the encoding scheme of the application. The CCSID of the LOB or XML data (the contents of the file) can be set by the application by using a DECLARE *host-variable* CCSID statement if the CCSID of the LOB or XML data is different from the CCSID of the application. DB2 will perform any character conversion that is required prior to the LOB or XML data being inserted into a table or written to a file.

References to stored procedure result sets

An application, written in a programming language other than Java, can access a result set that is returned from a stored procedure. A result set locator is used by the invoking application to access the result set.

A result set locator value for a result set can be obtained from an ASSOCIATE LOCATOR statement or with the DESCRIBE PROCEDURE statement. For more information, see “ASSOCIATE LOCATORS” on page 868 and “DESCRIBE PROCEDURE” on page 1250.

The result set locator value is specified on an ALLOCATE CURSOR statement to define a cursor in the invoking application and to associate it with a stored procedure result set. For more information, see “ALLOCATE CURSOR” on page 696.

A DESCRIBE CURSOR statement can be used in the invoking application to obtain information on the characteristics of the columns of a stored procedure result set. For more information, see “DESCRIBE CURSOR” on page 1238.

The application can then access the rows of the result set using FETCH statements with the allocated cursor.

References to result set locator variables

A *result set locator variable* is a variable that contains the locator that identifies a stored procedure result set. A result set locator variable in an SQL statement must identify a result set locator variable described in the program according to the rules for declaring result set locator variables. This is always indirectly through an SQL statement.

For example, in C:

```
static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1;
```

A result set locator variable in an SQL procedure is defined with the RESULT_SET_LOCATOR VARYING in a compound statement. See “compound-statement” on page 1554 for additional information.

The term *rs-locator-variable*, as used in the syntax diagrams, shows a reference to a result set locator variable. The meta-variable *rs-locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

When the indicator variable associated with a result set locator is null, the referenced result set is not defined.

If a result set locator variable does not currently represent any stored procedure result set, an error occurs when the locator variable is referenced.

A commit operation destroys all open cursors that were declared in the stored procedure without the WITH HOLD option and the result set locators that are associated with those cursors. Otherwise, a cursor and its associated result set locator persist past the commit.

References to built-in session variables

DB2 provides several built-in session variables that contain information about the server and application process. The value of a built-in session variable can be obtained by invoking the GETVARIABLE function with the name of the built-in session variable.

DB2 provides the following built-in session variables:

SYSIBM.APPLICATION_ENCODING_SCHEME

Contains a string that corresponds to the value that is specified for the APPLICATION ENCODING field on the DSNTIPF installation panel. The value will be EBCDIC, ASCII, UNICODE, or 1-65533, and this session variable can never be null.

SYSIBM.COBOL_STRING_DELIMITER

Contains a string that corresponds to the value that is specified for the STRING DELIMITER field on the DSNTIPF installation panel. The value will be DEFAULT, ", or ', and this session variable can never be null.

SYSIBM.DATA_SHARING_GROUP_NAME

Contains a string that corresponds to the name of the data sharing group for this DB2 subsystem. If the subsystem is not part of data sharing group, the null value is returned.

SYSIBM.DATE_FORMAT

Contains a string that corresponds to the value that is specified for the DATE FORMAT field on the DSNTIP4 installation panel. The value will be ISO, JIS, USA, EUR, or LOCAL, and this session variable can never be null.

SYSIBM.DATE_LENGTH

Contains a string that corresponds to the value that is specified for the LOCAL DATE LENGTH field on the DSNTIP4 installation panel. The value will be 10-254, or 0 for no exit, and this session variable can never be null.

SYSIBM.DECIMAL_ARITHMETIC

Contains a string that corresponds to the value that is specified for the DECIMAL ARITHMETIC field on the DSNTIP4 installation panel. The value will be DEC15, DEC31, 15, or 31, and this session variable can never be null.

SYSIBM.DECIMAL_POINT

Contains a string that corresponds to the value that is specified for the DECIMAL POINT IS field on the DSNTIPF installation panel. The value will be '.' or ',' and this session variable can never be null.

SYSIBM.DEFAULT_DECFLOAT_ROUND_MODE

Contains a string that corresponds to the value that is specified for the DECFLOAT ROUNDING MODE field on the DSNTIPF installation panel. This session variable can never be null.

SYSIBM.DEFAULT_DEFAULT_SSID

Contains a string that corresponds to the value that is specified for the GROUP ATTACH field on the DSNTIPK installation panel or the SUBSYSTEM NAME field on the DSNTIPM installation panel. This session variable can never be null.

SYSIBM.DEFAULT_LANGUAGE

Contains a string that corresponds to the value that is specified for the

LANGUAGE DEFAULT field on the DSNTIPF installation panel. The value will be ASM, C, CPP, IBMCOB, FORTRAN, or PL/I, and this session variable can never be null.

SYSIBM.DEFAULT_LOCALE_LC_CTYPE

Contains a string that corresponds to the value that is specified for the LOCALE LC_CTYPE field on the DSNTIPF installation panel. This session variable can never be null.

SYSIBM.DISTRIBUTED_SQL_STRING_DELIMITER

Contains a string that corresponds to the value that is specified for the DIST SQL STR DELIMTR field on the DSNTIPF installation panel. The value will be ", or ', and this session variable can never be null.

SYSIBM.DSNHDECP_NAME

Contains a string that corresponds to the fully qualified data set name of the data set from which the application defaults module DSNHDECP was loaded. For instance, 'DSN910.SDSNEXIT(DSNHDECP)'. This session variable can never be null.

SYSIBM.DYNAMIC_RULES

Contains a string that corresponds to the value that is specified for the USE FOR DYNAMICRULES field on the DSNTIP4 installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.ENCODING_SCHEME

Contains a string that corresponds to the value that is specified for the DEF ENCODING SCHEME field on the DSNTIPF installation panel. The value will be EBCDIC, ASCII, or UNICODE, and this session variable can never be null.

SYSIBM.MIXED_DATA

Contains a string that corresponds to the value that is specified for the MIXED DATA field on the DSNTIPF installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.NEWFUN

Contains a string that corresponds to the value that is specified for the INSTALL TYPE field on the DSNTIPA1 installation panel. The value will be INSTALL, UPDATE, MIGRATE, or ENFM, and this session variable can never be null. The value reflects the setting of the DSNHDECP variable NEWFUN.

SYSIBM.PACKAGE_NAME

Contains a string that corresponds to the name of the package that is currently being executed. If a package is not currently being executed, the null value is returned. (This situation can occur when the plan that is being executed bound one or more DBRMs directly).

SYSIBM.PACKAGE_SCHEMA

Contains a string that corresponds to the collection id of the package that is currently being executed. If a package is not currently being executed, the null value is returned.

SYSIBM.PACKAGE_VERSION

Contains a string that corresponds to the version of the package that is currently being executed. If a package is not currently being executed, the null value is returned.

SYSIBM.PAD_NUL_TERMINATED

Contains a string that corresponds to the value that is specified for the

PAD NUL-TERMINATED field on the DSNTIP4 installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.PLAN_NAME

Contains a string that corresponds to the name to the plan that is currently being executed. This session variable can never be null.

SYSIBM.SECLABEL

Contains a string that corresponds to the RACF SECLABEL value, if any, that has been defined for the current userid. If a value has not been defined, the null value is returned.

SYSIBM.SQL_STRING_DELIMITER

Contains a string that corresponds to the value that is specified for the SQL STRING DELIMITER field on the DSNTIPF installation panel. The value will be DEFAULT, ", or ', and this session variable can never be null.

SYSIBM.SSID

Contains a string that corresponds to the actual DB2 subsystem identifier for this DB2 subsystem. This session variable can never be null.

SYSIBM.STANDARD_SQL

Contains a string that corresponds to the value that is specified for the STD SQL LANGUAGE field on the DSNTIP4 installation panel. The value will be YES or NO, and this session variable can never be null.

SYSIBM.SYSTEM_NAME

Contains a string that corresponds to the name of the DB2 for z/OS subsystem, as defined in field SUBSYSTEM NAME on installation panel DSNTIPM. This session variable can never be null.

SYSIBM.SYSTEM_ASCII_CCSID

Contains a value that represents the ASCII CCSIDs that are in use on this system. The information is returned as a comma-delimited string that corresponds to the ASCII CCSID that was specified on installation panel DSNTIPF. The three values that are returned correspond to the SBCS, MIXED, and graphic CCSID that are in use for ASCII data on this system. A value of 65534 for the MIXED or graphic CCSID indicates that this system does not support storing data in that CCSID. This session variable can never be null.

SYSIBM.SYSTEM_EBCDIC_CCSID

Contains a value that represents the EBCDIC CCSIDs that are in use on this system. The information is returned as a comma-delimited string that corresponds to the EBCDIC CCSID that was specified on installation panel DSNTIPF. The three values that are returned correspond to the SBCS, MIXED, and graphic CCSID that are in use for EBCDIC data on this system. A value of 65534 for the MIXED or graphic CCSID indicates that this system does not support storing data in that CCSID. This session variable can never be null.

SYSIBM.SYSTEM_UNICODE_CCSID

Contains a value that represents the Unicode CCSIDs that are in use on this system. The information is returned as a comma-delimited string that corresponds to the UNICODE CCSID that was specified on installation panel DSNTIPF. The three values that are returned correspond to the SBCS, MIXED, and graphic CCSID that are in use for Unicode data on this system. This session variable can never be null.

SYSIBM.TIME_FORMAT

Contains a string that corresponds to the value that is specified for the

TIME FORMAT field on the DSNTIP4 installation panel. The value will be ISO, JIS, USA, EUR, or LOCAL, and this session variable can never be null.

SYSIBM.TIME_LENGTH

Contains a string that corresponds to the value that is specified for the LOCAL TIME LENGTH field on the DSNTIP4 installation panel. The value will be 8-254 or 0 for no exit, and this session variable can never be null.

SYSIBM.VERSION

Contains a string that represents the version of DB2. This string has the form *pppvvrrm* where:

- *ppp* is a product string that is set to the value 'DSN'
- *vv* is a two-digit version identifier such as '09'
- *rr* is a two-digit release identifier such as '01'
- *m* is a one-digit maintenance level identifier such as '5' (Values 0, 1, 2, 3, and 4 are reserved for conversion mode and enabling-new-function mode, and values 5, 6, 7, 8, and 9 are reserved for new function mode.)

This session variable can never be null.

For example, the following statement sets the value of host variable *hv1* to the name of the plan that is currently being executed:

```
SET :hv1 = GETVARIABLE(SYSIBM.PLAN_NAME);
```

For more information about the GETVARIABLE function, see “GETVARIABLE” on page 374.

Host structures in PL/I, C, and COBOL

A *host structure* is a PL/I structure, C structure, or COBOL group that is referred to in an SQL statement.

In Java and REXX, there is no equivalent to a host structure. Host structures are defined by statements of the host language, as explained in *DB2 Application Programming and SQL Guide*. As used here, the term *host structure* does not include an SQLCA or SQLDA.

The form of a host structure reference is identical to the form of a host variable reference. The reference :S1:S2 is a host structure reference if S1 names a host structure. If S1 designates a host structure, S2 must be a small integer variable or an array of small integer variables. S1 is the host structure and S2 is its indicator array.

A host structure can be referred to in any context where a list of host variables can be referenced. A host structure reference is equivalent to a reference to each of the host variables contained within the structure in the order which they are defined in the host language structure declaration. The *n*th variable of the indicator array is the indicator variable for the *n*th variable of the host structure.

In PL/I, for example, if *V1*, *V2*, and *V3* are declared as the variables within the structure *S1*, the following two statements are equivalent:

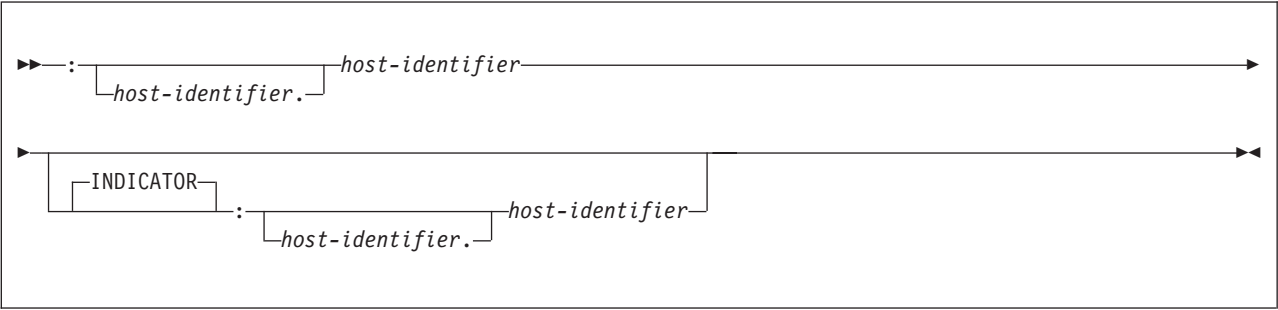
```
EXEC SQL FETCH CURSOR1 INTO :S1;  
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3;
```

If the host structure has *m* more variables than the indicator array, the last *m* variables of the host structure do not have indicator variables. If the host structure has *m* fewer variables than the indicator array, the last *m* variables of the indicator

array are ignored. These rules also apply if a reference to a host structure includes an indicator variable or a reference to a host variable includes an indicator array. If an indicator array or variable is not specified, no variable of the host structure has an indicator variable.

In addition to structure references, individual host variables or indicator variables in PL/I, C, and COBOL can be referred to by qualified names. The qualified form is a host identifier followed by a period and another host identifier. The first host identifier must name a structure, and the second host identifier must name a host variable at the next level within that structure.

In PL/I, C, and COBOL, the syntax of *host-variable* is:



In general, a *host-variable* in an expression must identify a host variable (not a structure) described in the program according to the rules for declaring host variables. However, there are a few SQL statements that allow a host variable in an expression to identify a structure, as specifically noted in the descriptions of the statements.

The following examples show references to host variables and host structures:

:V1 :S1.V1 :S1.V1:V2 :S1.V2:S2.V4

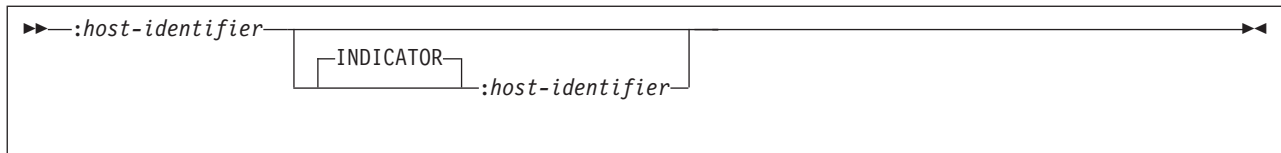
Host-variable-arrays in PL/I, C, C++, and COBOL

A *host-variable-array* is an array in which each element of the array contains a value for the same column. The first element in the array corresponds to the first value, the second element in the array corresponds to the second value, and so on. A host-variable-array can only be referenced in a FETCH statement for a multiple row fetch, in an INSERT statement with a multiple row insert, or in a multiple row MERGE statement.

Host-variable-arrays are defined by statements of the host language as explained in *DB2 Application Programming and SQL Guide*.

The form of a host structure reference is similar to the form of a host variable reference. The reference :COL1:COL1IND is a host-variable-array reference if COL1 designates an array. If COL1 designates an array, COL1IND must be a one dimensional array of small integer host variables. The dimension of the host-variable-array must be less than or equal to the dimension of the indicator array. If an indicator array is not specified, no variable of the main host-variable-array has an indicator variable.

In PL/I, C, C++, and COBOL, the syntax of *host-variable-array* is:



In the following example, *COL1* is the main host-variable-array and *COL1IND* is its indicator array. If *COL1* has 10 elements for fetching a single column of data for multiple rows of data, *COL1IND* must also have 10 entries.

```
EXEC SQL FETCH CURSOR FOR 5 ROWS INTO :COL1 :COL1IND;
```

Functions

A *function* is an operation denoted by a function name followed by zero or more operands that are enclosed in parentheses.

It represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*. For example, a function can be passed with two input arguments that have date and time data types and return a value with a timestamp data type as the result.

Types of functions

There are several ways to classify functions. One way to classify functions is as built-in functions, user-defined functions, or cast functions that are generated for distinct types.

Built-in functions

Built-in functions are functions that come with DB2 for z/OS. These functions provide a single-value result.

Built-in functions include operator functions such as "+", aggregate functions such as AVG, and scalar functions such as SUBSTR. For a list of the built-in aggregate and scalar functions and information on these functions, see Chapter 3, "Functions," on page 259.

The built-in functions are in schema SYSIBM.

The RANK, DENSE_RANK, and ROW_NUMBER specifications are sometimes referred to as built-in functions. Refer to "OLAP specification" on page 212 for more information on these specifications.

User-defined functions

User-defined functions are functions that are created using the CREATE FUNCTION statement and registered to the DB2 in the catalog. These functions allow users to extend the function of DB2 by adding their own or third party vendor function definitions.

A user-defined function is an SQL, external, or sourced function. An SQL function is defined to the database using only an SQL RETURN statement. An external function is defined to the database with a reference to an external program that is executed when the function is invoked. A sourced function is defined to the database with a reference to a built-in function or another user-defined function. Sourced functions can be used to extend built-in aggregate and scalar functions for use on distinct types.

A user-defined function resides in the schema in which it was registered. The schema cannot be SYSIBM.

To help you define and implement user-defined functions, sample user-defined functions are supplied with DB2. You can also use these sample user-defined functions in your application program just as you would any other user-defined function if the appropriate installation job has been run. For a list of the sample user-defined functions, see “Sample user-defined functions” on page 2007. For more information on creating and using user-defined functions, see *DB2 Application Programming and SQL Guide*.

Cast functions

Cast functions are functions that DB2 automatically generates when a distinct type is created using the CREATE TYPE statement.

Cast functions support casting from the distinct type to the source type and from the source type to the distinct type. The ability to cast between the data types is important because a distinct type is compatible only with itself.

The generated cast functions reside in the same schema as the distinct type for which they were created. The schema cannot be SYSIBM. For more information on the functions that are generated for a distinct type, see “CREATE TYPE” on page 1177.

Additional way to classify functions

Another way to classify functions is as aggregate, scalar, or table functions, depending on the input data values and result values.

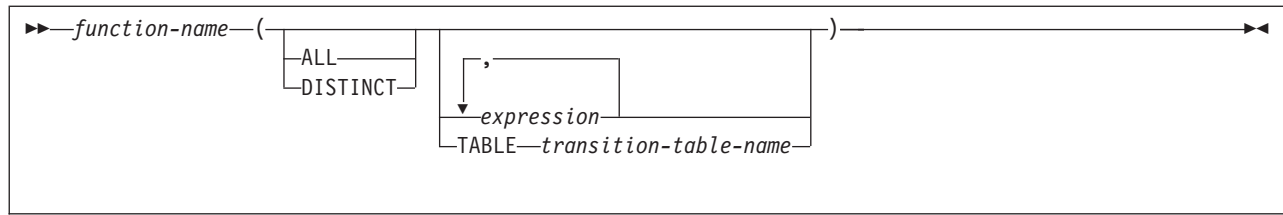
- An *aggregate function* receives a set of values for each argument (such as the values of a column) and returns a single-value result for the set of input values. Aggregate functions are sometimes called *column functions*. Built-in functions and user-defined sourced functions can be aggregate functions. Aggregate functions cannot be external user-defined function or SQL functions.
- A *scalar function* receives a single value for each argument and returns a single-value result. Built-in functions and user-defined functions, external, sourced, and SQL, can be scalar functions. The functions that are created for distinct types are also scalar functions.
- A *table function* returns a table for the set of arguments it receives. Each argument is a single value. A table function can only be referenced in the FROM clause of a subselect. A table function can be defined as an external function, but a table function cannot be a sourced function, or an SQL function.

Table functions can be used to apply SQL language processing power to data that is not DB2 data or to convert such data into a DB2 table. For example, a table function can take a file and convert it to a table or can get data from the Internet and tabularize it.

For a list of the aggregate, scalar, and table functions and information on these functions, see Chapter 3, “Functions,” on page 259.

Function invocation

Each reference to a scalar or aggregate function (either built-in or user-defined) conforms to the following syntax:



In the above syntax, *expression* cannot include an aggregate function. See “Expressions” on page 180 for other rules for *expression*.

The **ALL** or **DISTINCT** keyword can only be specified for an aggregate function or a user-defined function that is sourced on an aggregate function.

When a function is invoked within a trigger body, the **TABLE** keyword can be specified to indicate that an argument is a trigger transition table. In this case, the corresponding parameter of the function must have been defined with the **TABLE LIKE** clause.

Table functions can be referenced only in the FROM clause of a subselect. For more information on referencing a table function, see the description of the “from-clause” on page 638.

When the function is invoked, the value of each of its parameters is assigned using storage assignment, to the corresponding parameter of the function. Control is passed to external functions according to the calling conventions of the host language. When execution of a user-defined aggregate or scalar function is complete, the result of the function is assigned, using storage assignment, to the result data type. For information about assignment rules, see “Assignment and comparison” on page 102.

Additionally, a character FOR BIT DATA argument cannot be passed as input for a parameter that is not defined as character FOR BIT DATA. Likewise, a character argument that is not FOR BIT DATA cannot be passed as input for a parameter defined as character FOR BIT DATA.

Function resolution

After a function is invoked, DB2 must determine which function to execute. This process is called *function resolution* and it applies to both built-in and user-defined function.

A function is invoked by its function name, which is implicitly or explicitly qualified with a schema name, followed by parentheses that enclose the arguments to the function. Within the database, each function is uniquely identified by its *function signature*, which is its schema name, function name, the number of parameters, and the data types of the parameters. Thus, a schema can contain several functions that have the same name but each of which have a different number of parameters or parameters with different data types. Also, a function with the same name, number of parameters, and types of parameters can exist in multiple schemas.

Function resolution is similar for functions that are invoked with a qualified or unqualified function name with the exception that for an unqualified name, DB2 needs to search more than one schema.

Qualified function resolution: When a function is invoked with a schema name and a function name, DB2 only searches the specified schema to resolve which function to execute.

DB2 finds the appropriate function instance when all of the following conditions are true:

- The name of the function instance matches the name in the function invocation.
- The number of input parameters in the function instance matches the number of function arguments in the function invocation.
- The authorization ID of the statement has the EXECUTE privilege to the function instance.
- The data type of each input argument of the function invocation matches or is *promotable* to the data type of the corresponding parameter of the function instance.

For a function invocation that passes a transition table, the data type, length, precision, and scale of each column in the transition table must match exactly the data type, length, precision, and scale of each column of the table that is named in the function instance definition.

If the function invocation contains no untyped parameter markers, the comparison of data types results in one best fit, which is the choice for execution (see “Determining the best fit” on page 177). For information on the promotion of data types, see “Promotion of data types” on page 95.

For a function invocation that contains untyped parameter markers, the data types of those parameter markers are considered to match or be promotable to the data types of the parameters in the function instance.

- The create timestamp for the function must be older than the bind timestamp for the package or plan in which the function is invoked.

If a function invoked from a trigger body receives a transition table, and the invocation occurs during an automatic rebind, the form of the invoked function used for function selection includes only the columns of the table that existed at the time of the original BIND or REBIND package or plan for the invoking program.

If the function invocation contains untyped parameter markers, the comparison can result in more than one best fit. In that case, DB2 returns an error.

If DB2 authorization checking is in effect, and DB2 performs an automatic rebind on a plan or package that contains a user-defined function invocation, any user-defined functions that were created after the original BIND or REBIND of the invoking plan or package are not candidates for execution.

If you use an access control authorization exit routine, some user-defined functions that were not candidates for execution before the original BIND or REBIND of the invoking plan or package might become candidates for execution during the automatic rebind of the invoking plan or package. See *DB2 Administration Guide* for information about function resolution with access control authorization exit routines.

If no function in the schema meets these criteria, an error occurs. If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns an integer data type where a character data type is required or returns a table where a table function is not allowed, an error occurs.

Unqualified function resolution: When a function is invoked with only a function name and no schema name, DB2 needs to search more than one schema to resolve the function instance to execute. DB2 uses these steps to choose the function:

1. The *SQL path* contains the list of schemas to search. For each schema in the path, DB2 selects a candidate function based on the same criteria described immediately above for qualified function resolution. However, if no function in the schema meets the criteria, an error does not occur, and a candidate function is not selected for that schema.
2. After identifying the candidate functions for the schemas in the path, DB2 selects the candidate with the best fit as the function to execute. If more than one schema contains the function instance with the best fit (the function signatures are identical except for the schema name), DB2 selects the function whose schema is first in the SQL path. If no function in any schema in the SQL path meets the criteria, an error occurs.

The create timestamp of a user-defined function must be older than the timestamp resulting from an explicit bind for the plan or package containing the function invocation. During autobind, built-in functions introduced in a later DB2 release than the DB2 release that was used to explicitly bind the package or plan are not considered for function resolution.

In a CREATE VIEW statement, function resolution occurs at the time the view is created. If another function with the same name is subsequently created, the view is not affected, even if the new function is a better fit than the one chosen at the time the view was created.

For more information on user-defined functions, such as how you can simplify function resolution or use the DSN_FUNCTION_TABLE to see how DB2 resolves a function, see *DB2 Application Programming and SQL Guide*.

Determining the best fit

More than one function with the same name might exist that is a candidate for execution. In that case, DB2 determines which function is the best fit for the invocation by comparing the data types of the parameters of each function in the set of candidate functions to determine which function satisfies the best fit requirements.

DB2 determines the function, or set of functions, that meet the best fit requirements for the invocation by comparing the argument and parameter data types. The data type of the result of the function or the type of function (aggregate, scalar, or table) under consideration does not enter into the determination of best fit.

When determining whether the data types of the parameters are the same as the arguments:

- Synonyms of data types match. For example, DOUBLE and FLOAT are considered to be the same.
- Attributes of a data type (such as length, precision, scale, CCSID) are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2) and DECIMAL(4,3).
- The character and graphic types are considered to be the same. For example, the following data types are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, and CLOB and DBCLOB. CHAR(13) and GRAPHIC(8) are considered to be the same type.

- For this argument, if one function has a data type that fits the function invocation better than the data types in the other functions, that function is the best fit. The precedence list for the promotion of data types in shows the data types that fit each data type, in best-to-worst order.
- If the data types of the first parameter for all the candidate functions fit the function invocation equally well, DB2 repeats this process for the next argument of the function invocation. DB2 continues this process for each argument until a best fit is found.

A subset of the candidate functions is obtained by considering only those functions for which the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance. The precedence list for the promotion of data types in Table 14 on page 95 shows the data types that fit (considering promotion) for each data type in best-to-worst order. If this subset is not empty, the best fit is determined using the promotable process on this subset of candidate functions. If this subset is empty, and the original set of candidate functions consisted of a single function, the best fit is determined using the castable process on the original candidate function. Otherwise, an error is returned.

Best-fit consideration:

After determining the function that is the best fit, use of the function still might not be permitted. Each function is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the function is invoked, an error occurs.

For example, assume functions named STEP are defined with different data types:

```
STEP(SMALLINT)returns CHAR(5)
STEP(DOUBLE)returns INTEGER
```

Assume also that the function is invoked with the following function reference (where S is a SMALLINT column):

```
SELECT ... 3+STEP(S) ...
```

Because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

In cases where the arguments of the function invocation are not an exact match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. See “Assignment and comparison” on page 102. Problems with conversions can also occur when precision, scale, length, or the encoding scheme differs between the argument and the parameter. Conversion might occur for a character string argument when the corresponding parameter of the function has a different encoding scheme or CCSID. For example, an error occurs on function invocation when mixed data that actually contains DBCS characters is specified as an argument and the corresponding parameter of the function is declared with an SBCS subtype.

Additionally, a character FOR BIT DATA argument cannot be passed as input for a parameter that is not defined as character FOR BIT DATA. Likewise, a character argument that is not FOR BIT DATA cannot be passed as input for a parameter that is defined as character FOR BIT DATA.

An error also occurs in the following examples:

- The function is referenced in a FROM clause, but the function selected by the function resolution step is a scalar or aggregate function.
- The function calls for a scalar or aggregate function, but the function selected by the resolution step is a table function.

SQL path considerations for built-in functions

Function resolution applies to all functions, including built-in functions and other functions provided by DB2. If a function is invoked without its schema name, the SQL path is searched.

With the exception of the DB2 MQSeries® functions, which are in schemas DB2MQ1C and DB2MQ2C, the other built-in functions are in schema SYSIBM. If SYSIBM is not first in the path, it is possible that DB2 will select another function instead of the intended built-in function. Likewise, if DB2MQ1C and DB2MQ2C are not in the path, it is possible that DB2 will select a function other than one of the provided DB2 MQSeries functions. If schema SYSIBM, SYSFUN, or SYSPROC is not explicitly specified in the SQL path, the schema is implicitly assumed at the front of the path. DB2 adds implicitly assumed schemas in the order of SYSIBM, SYSFUN, and SYSPROC. See “SQL path” on page 56 for information on how to specify the path so that the intended function is selected when it is invoked with an unqualified name.

Examples of function resolution

The following examples illustrate function resolution.

Example 1: Assume that MYSCHEMA contains two functions, both named FUNA, that were registered with these partial CREATE FUNCTION statements.

1. CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), INT, DOUBLE) ...
2. CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), REAL, DOUBLE) ...

Also assume that a function with three arguments of data types VARCHAR(10), SMALLINT, and DECIMAL is invoked with a qualified name:

```
MYSCHEMA.FUNA(VARCHARCOL, SMALLINTCOL, DECIMALCOL)
```

Both MYSCHEMA.FUNA functions are candidates for this function invocation because they meet the criteria specified in “Function resolution” on page 175. The data types of the first parameter for the two function instances in the schema, which are both VARCHAR, fit the data type of the first argument of the function invocation, which is VARCHAR, equally well. However, for the second parameter, the data type of the first function (INT) fits the data type of the second argument (SMALLINT) better than the data type of second function (REAL). Therefore, DB2 selects the first MYSCHEMA.FUNA function as the function instance to execute.

Example 2: Assume that these functions were registered with these partial CREATE FUNCTION statements:

1. CREATE FUNCTION SMITH.ADDIT (CHAR(5), INT, DOUBLE) ...
2. CREATE FUNCTION SMITH.ADDIT (INT, INT, DOUBLE) ...
3. CREATE FUNCTION SMITH.ADDIT (INT, INT, DOUBLE, INT) ...
4. CREATE FUNCTION JOHNSON.ADDIT (INT, DOUBLE, DOUBLE) ...
5. CREATE FUNCTION JOHNSON.ADDIT (INT, INT, DOUBLE) ...
6. CREATE FUNCTION TODD.ADDIT (REAL) ...
7. CREATE FUNCTION TAYLOR.SUBIT (INT, INT, DECIMAL) ...

Also assume that the SQL path at the time an application invokes a function is "TAYLOR" "JOHNSON", "SMITH". The function is invoked with three data types (INT, INT, DECIMAL) as follows:

```
SELECT ... ADDIT(INTCOL1, INTCOL2, DECIMALCOL) ...
```

Function 5 is chosen as the function instance to execute based on the following evaluation:

- Function 6 is eliminated as a candidate because schema TODD is not in the SQL path.
- Function 7 in schema TAYLOR is eliminated as a candidate because it does not have the correct function name.
- Function 1 in schema SMITH is eliminated as a candidate because the INT data type is not promotable to the CHAR data type of the first parameter of Function 1.
- Function 3 in schema SMITH is eliminated as a candidate because it has the wrong number of parameters.
- Function 2 is a candidate because the data types of its parameters match or are promotable to the data types of the arguments.
- Both Function 4 and 5 in schema JOHNSON are candidates because the data types of their parameters match or are promotable to the data types of the arguments. However, Function 5 is chosen as the better candidate because although the data types of the first parameter of both functions (INT) match the first argument (INT), the data type of the second parameter of Function 5 (INT) is a better match of the second argument (INT) than Function 4 (DOUBLE).
- Of the remaining candidates, Function 2 and 5, DB2 selects Function 5 because schema JOHNSON comes before schema SMITH in the SQL path.

Expressions

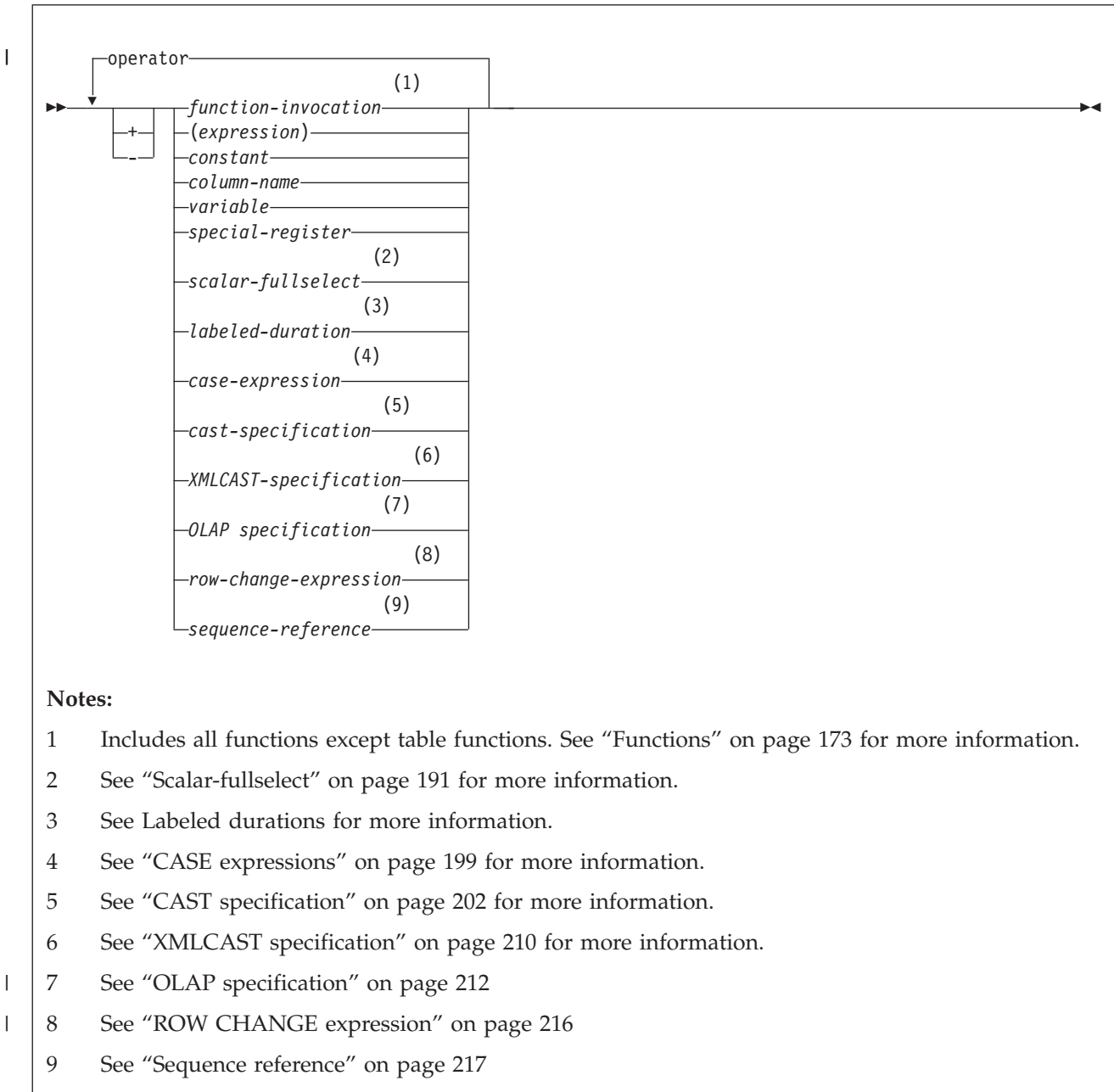
An *expression* specifies a value and can take a number of different forms.

Authorization: The use of some of the expressions, such as a *scalar-fullselect*, *sequence-reference*, or user-defined *function*, requires having the appropriate authorization. For these objects, the privilege set that is defined below must include the following authorization:

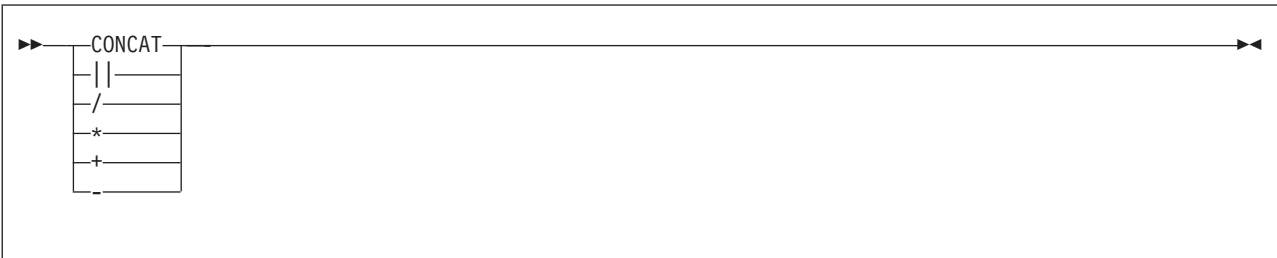
- *scalar-fullselect*. For information about authorization considerations, see "Authorization" on page 630.
- *sequence-reference*. The USAGE privilege on the specified sequence, ownership of the sequence, or SYSADM authority. For example, with a sequence reference, USAGE authorization on the sequence is required.
- *function-invocation*. Authorization to execute the function. For information about how the particular function is chosen and authorization considerations, see "Function resolution" on page 175.
- *cast-specification*. The authorization to reference a user-defined type in a cast specification. For information about authorization considerations, see "CAST specification" on page 202.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

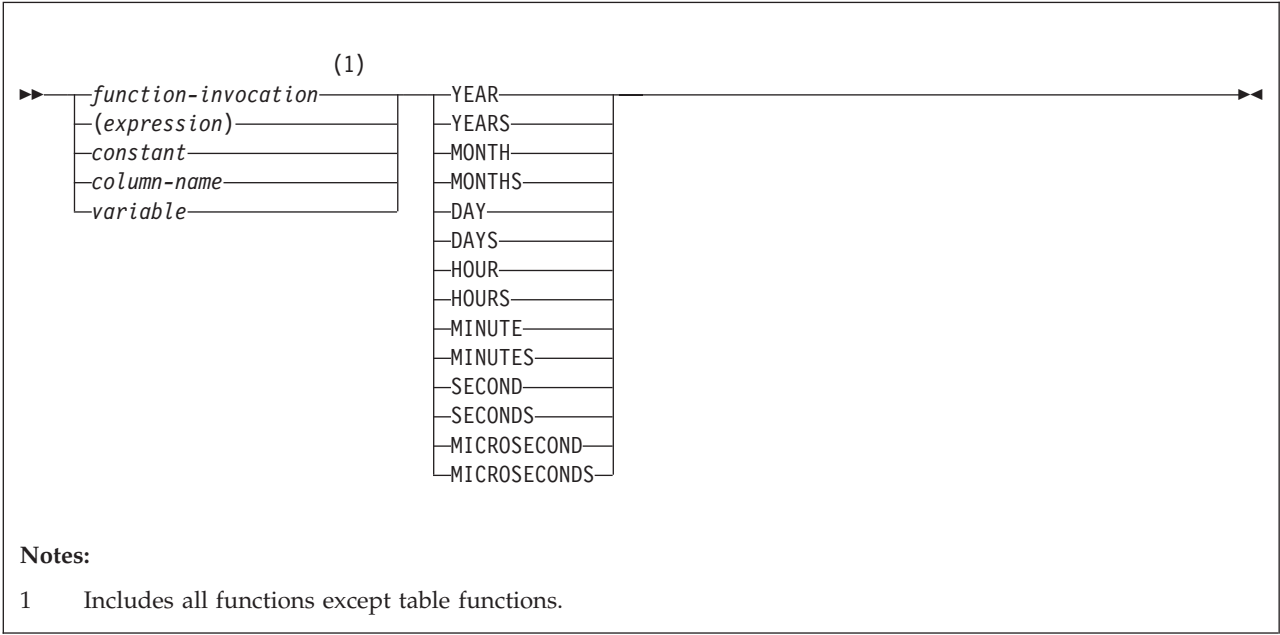
The form of an expression is as follows:



operator:



labeled-duration:



Without operators

If no operators are used, the result of the expression is the specified value.

Examples:

SALARY :SALARY 'SALARY' MAX(SALARY)

With arithmetic operators

If arithmetic operators are used, the result of the expression is a number derived from the application of the operators to the values of the operands.

The result of the expression can be null. If any operand has the null value, the result of the expression is the null value. Arithmetic operators (except unary plus, which is meaningless) must not be applied to strings. For example, USER+2 is invalid. Multiplication and division operators must not be applied to datetime values, which can only be added and subtracted.

The prefix operator + (unary plus) does not change its operand. The prefix operator - (unary minus) reverses the sign of a nonzero operand. If the data type of A is small integer, the data type of -A is large integer. The first character of the token following a prefix operator must not be a plus or minus sign.

The infix operators +, -, *, and / specify addition, subtraction, multiplication, and division, respectively. The value of the second operand of division must not be zero.

Arithmetic with two integer operands

If both operands of an arithmetic operator are integers, the operation is performed in binary. The result is a large integer unless either (or both) operand is a big integer, in which case the result is a big integer.

The result of an integer arithmetic operation (including unary minus) must be within the range of the result type.

Arithmetic with an integer and a decimal operand

If one operand is an integer and the other operand is decimal, the operation is performed in decimal. The arithmetic operation uses a temporary copy of the integer that has been converted to a decimal number.

The temporary copy of the integer that has been converted to a decimal number has a precision p and scale 0. p is 19 for a big integer, 11 for a large integer, and 5 for a small integer. In the case of an integer constant, p depends on the number of digits in the integer constant. p is 5 for an integer constant consisting of 5 digits or fewer. Otherwise, p is the same as the number of digits in the integer constant.

Arithmetic with an integer and a decimal floating-point operand

If one operand is a small integer, large integer, or big integer and the other is a decimal floating-point number, the operation is performed in decimal floating point. The arithmetic operation uses a temporary copy of the integer that has been converted to a decimal floating-point number.

For small integer or large integer, the temporary copy of the integer is converted to DECFLOAT(16). For big integer, the temporary copy of the big integer is converted to DECFLOAT(34). The rules for two decimal floating point operands are then applied.

Arithmetic with two decimal operands

If both operands are decimal, the operation is performed in decimal.

The result of any decimal arithmetic operation is a decimal number with a precision and scale that depend on two factors:

The precision and scale of the operands

In the discussion of operations with two decimal operands, the precision and scale of the first operand are denoted by p and s , that of the second operand by p' and s' . Thus, for a division, the dividend has precision p and scale s , and the divisor has precision p' and scale s' .

Whether DEC31 or DEC15 is in effect for the operation

DEC31 and DEC15 specify the rules to be used when both operands in a decimal operation have precisions of 15 or less. DEC15 specifies the rules which do not allow a precision greater than 15 digits, and DEC31 specifies the rules which allow a precision of up to 31 digits. The rules for DEC31 are always used if either operand has a precision greater than 15.

For static SQL statements, the value of the field DECIMAL ARITHMETIC on installation panel DSNTIP4 or the precompiler option DEC determines whether DEC15 or DEC31 is used.

For dynamic SQL statements, the value of the field DECIMAL ARITHMETIC on installation panel DSNTIP4, the precompiler option DEC, or the special register CURRENT PRECISION determines whether DEC15 or DEC31 is used according to these rules:

- Field DECIMAL ARITHMETIC applies if either of these conditions is true:
 - DYNAMICRULES run behavior applies and the application has not set CURRENT PRECISION.

For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 64.

- DYNAMICRULES bind, define, or invoke behavior applies; the value of installation panel field USE FOR DYNAMICRULES is YES; and the application has not set CURRENT PRECISION.
- Precompiler option DEC applies if DYNAMICRULES bind, define, or invoke behavior is in effect, the value of installation panel field USE FOR DYNAMICRULES is NO, and the application has not set CURRENT PRECISION.
- Special register CURRENT PRECISION applies if the application sets the register.

The value of DECIMAL ARITHMETIC is the default value for the precompiler option and the special register. SQL statements executed using SPUFI use the value in DECIMAL ARITHMETIC.

Decimal addition and subtraction:

For decimal operations, the precision and scale of the result depends on the precision and scale of the operands.

If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands that has been extended with trailing zeros so that its fractional part has the same number of digits as the other operand.

The precision of the result is the minimum of n and the quantity $\text{MAX}(p-s, p'-s')+1$. The scale is $\text{MAX}(s, s')$. n is 31 if DEC31 is in effect or if the precision of at least one operand is greater than 15. Otherwise, n is 15.

In COBOL, blanks must precede and follow a minus sign to avoid any ambiguity with COBOL host variable names (which allow the use of a dash).

Decimal multiplication:

For decimal multiplication, the precision and scale of the result depends on the precision and scale of the operands.

For multiplication, the precision of the result is $\text{MIN}(n, p+p')$, and the scale is $\text{MIN}(n, s+s')$. n is 31 if DEC31 is in effect or if the precision of at least one operand is greater than 15. Otherwise, n is 15.

If both operands have a precision greater than 15, the operation is performed using a temporary copy of the operand with the smaller precision. If the operands have the same precision, the second operand is selected. If more than 15 significant digits are needed for the integral part of the copy, the statement's execution is ended and an error occurs. Otherwise, the copy is converted to a number with precision 15, by truncating the copy on the right. The truncated copy has a scale of $\text{MAX}(0, S-(P-15))$, where P and S are the original precision and scale. If, in the process of truncation, one or more nonzero digits are removed, SQLWARN7 in SQLCA is set to W, indicating loss of precision.

When both operands have a precision greater than 15, the foregoing formulas for the precision and scale of the result still apply, with one change: for the operand selected as the copy, use the precision and scale of the truncated copy; that is, use 15 as the precision and $\text{MAX}(0, S-(P-15))$ for the scale.

Let n denote the value of the operand with the greater precision or the first operand in the case of operands with the same precision. The number of leading zeros in a 31-digit representation of n must be greater than the precision of the other operand. This is always the case if the precision of the operand is 15 or less. With greater precisions, overflow can occur even if the precision of the result is less than 31. For example, the expression:

100000000000000000000000000000000. * 1

will cause overflow because the number of leading zeros in the 31-digit representation of the large number and the precision of the small number are both 5 (see "Arithmetic with an integer and a decimal operand" on page 183).

Decimal division:

The rules for a specific decimal division depend on whether the DEC31 option is in effect for the operation, whether p is greater than 15, and whether p' is greater than 15

The following table shows how the precision and scale of the result depend on these factors. In that table, the occurrence of "N/A" in a row implies that the indicated factor is not relevant to the case represented by the row.

Table 39. Precision (p) and scale (s) of the result of a decimal division

DEC31	p	p'	P	S
Not in effect	≤ 15	≤ 15	15	$15 - (p - s + s')$
In effect	≤ 15	≤ 15	31	$N - (p - s + s')$, where N is $30 - p'$ if p' is odd. N is $29 - p'$ if p' is even.
N/A	> 15	≤ 15	31	$N - (p - s + s')$, where N is $30 - p'$ if p' is odd. N is $29 - p'$ if p' is even.
N/A	N/A	> 15	31	$15 - (p - s + x)$, where x is $\text{MAX}(0, s' - (p' - 15))$ (See Note 2 below)

Notes on decimal division:

1. If the calculated value of ' s ' is negative, an error occurs. If a minimum divide result scale is specified, this error does not occur. A minimum divide result scale of 3 can be specified using the MINIMUM DIVIDE SCALE field on the installation panel DSNTIP4. A minimum divide scale result between 1 and 9 can be specified using the DECIMAL ARITHMETIC OPTION of the form 'Dpp.s' where ' pp ' is 15 or 31 and represents the precision and ' s ' represents the minimum divide scale, as a number between 1 and 9. Such a specification overrides the MINIMUM DIVIDE SCALE. When a minimum divide result scale is specified, the formula $\text{MAX}(s, s')$, where s represents the scale derived from the above table and s' represents the value specified by the minimum divide result scale, is applied and a new scale is derived. The newly derived scale is the scale of the result and overrides any scale derived using the table above.
2. If p' is greater than 15, the division is performed using a temporary copy of the divisor. If more than 15 significant digits are needed for the integral part of the divisor, the statement's execution is ended, and an error occurs. Otherwise, the copy is converted to a number with precision 15, by truncating the copy on the right. The truncated copy has a scale of $\text{MAX}(0, s' - (p' - 15))$, which is the

formula for x . If, in the process of truncation, one or more nonzero digits are removed, SQLWARN7 in SQLCA is set to W, indicating loss of precision.

Arithmetic with a decimal and a decimal floating-point operand

If one operand is a decimal and the other is a decimal floating point, the operation is performed in decimal floating point. The arithmetic operation uses a temporary copy of the decimal that has been converted to a decimal floating point based on the precision of the decimal number.

If the decimal number has a precision of less than 17, the decimal number is converted to DECFLOAT(16). Otherwise, the decimal number is converted to DECFLOAT(34). The rules for two decimal floating-point operands are then applied.

Arithmetic with floating-point operands

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point. If necessary, the operands are first converted to double-precision floating-point numbers. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer that has been converted to double-precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number that has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can affect the results slightly because floating-point operands are approximate representations of real numbers. Because the order in which operands are processed might be implicitly modified by DB2 (for example, DB2 might decide what degree of parallelism to use and what access plan to use), an application that uses floating-point operands should not depend on the results being precisely the same each time an SQL statement is executed.

Arithmetic with a floating-point and a decimal floating-point operand

If one operand is a floating-point number (real or double) and the other is a decimal floating-point number, the operation is performed in decimal floating-point. The arithmetic operation uses a temporary copy of the floating-point number that has been converted to a decimal floating-point number.

Arithmetic with two decimal floating-point operands

If both operands are decimal floating point, the operation is performed in decimal floating point. If one operand is DECFLOAT(n) and the other is DECFLOAT(m), the operation is performed in DECFLOAT(max(n , m)).

General Arithmetic Operation Rules for DECFLOAT:

Certain general rules apply to all arithmetic operations on the DECFLOAT data type.

The following general rules apply to all arithmetic operations on the DECFLOAT data type:

- Every operation on finite numbers is carried out as though an exact mathematical result is computed, using integer arithmetic on the coefficient where possible.

If the coefficient of the theoretical exact result has no more than the number of digits that reflect its precision (16 or 34), it is used for the result without change (unless there is an underflow or overflow condition). If the coefficient has more than the number of digits that reflect its precision, it is rounded to exactly the number of digits that reflect its precision (16 or 34), and the exponent is increased by the number of digits that are removed.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

If the value of the adjusted exponent of the result is less than E_{\min} , an exception condition is returned. In this case, the calculated coefficient and exponent form the result, unless the value of the exponent is less than E_{tiny} in which case the exponent is set to E_{tiny} , the coefficient is rounded (possibly to zero) to match the adjustment of the exponent, and the sign is unchanged. If this rounding gives an inexact result, an underflow exception condition is returned.

If the value of the adjusted exponent of the result is larger than E_{\max} , an overflow exception condition is returned. In this case, the result is as defined as an overflow exception condition and might be infinite. It will have the same sign as the theoretical result.

- Arithmetic that uses the special value infinity follows the usual rules, where negative infinity is less than every finite number and positive infinity is greater than every finite number.

Under these rules, an infinite result is always exact. Certain uses of infinity return an invalid operation condition. The following list is a list of operations that can cause an invalid operation condition and the result of the operation is NaN when one of the operands is infinity but the other operand is not NaN nor sNaN.

- Add +infinity to -infinity during an addition or subtraction operation
- Multiply 0 by +infinity or -infinity
- Divide either +infinity or -infinity by either +infinity or -infinity
- The dividend for a MOD function is either +infinity or -infinity
- Either argument of the QUANTIZE function is +infinity or -infinity
- The second argument of the POWER[®] function is +infinity or -infinity
- Signaling NaNs when used as an operand to an arithmetic operation

The following arithmetic rules apply to arithmetic operations and the NaN value:

- The result of any arithmetic operation which has an operand which is a NaN (a quiet NaN or signaling NaNs) is NaN. The sign of the result is copied from the first operand which is a signaling NaN, or if neither operand is signaling then the sign is copied from the first operand which is a NaN. Whenever a result is a NaN, the sign of the result depends only on the copied operand.
- The sign of the result of a multiplication or division will be negative only if the operands have different signs and neither is a NaN.
- The sign of the result of an addition or subtraction will be negative only if the result is less than zero and neither operand is a NaN, except for the following cases where the result is a negative 0:

- A result is rounded to zero, and the value, before rounding, had a negative sign
- Subtract 0 from -0
- Addition of operands with opposite signs (or subtraction of operands with the same sign), the result has a coefficient of 0, and the rounding mode is ROUND_FLOOR
- Multiplication or division and the result has a coefficient of 0 and the signs of the operands are different
- The first argument of the POWER function is -0, and the second argument is a positive odd number
- The argument of the CEIL, FLOOR, or SQRT function is -0
- The first argument of the ROUND or TRUNCATE function is -0

Examples involving special DECFLOAT values:

```
INFINITY + 1           = INFINITY
INFINITY + INFINITY    = INFINITY
INFINITY + -INFINITY   = NAN      -- exception
NAN + 1                = NaN
NAN + INFINITY         = NaN
1 - INFINITY           = -INFINITY
INFINITY - INFINITY    = NAN      -- exception
-INFINITY - -INFINITY  = NAN      -- exception
-0.0 - 0.0E1           = -0
-1.0 * 0.0E1           = -0
1.0E1 / 0              = INFINITY
-1.0E5 / 0.0           = -INFINITY
1.0E5 / -0             = -INFINITY
INFINITY / -INFINITY   = NAN      -- exception
INFINITY / 0           = INFINITY
-INFINITY / 0          = -INFINITY
-INFINITY / -0         = INFINITY
```

Arithmetic with distinct type operands

A distinct type cannot be used with arithmetic operators even if its source data type is numeric.

To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE ( INCOME, EXPENSES )
  RETURNS DECIMAL(8,2) SOURCE "-" ( DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a function to subtract the new data types.

```
CREATE FUNCTION "-" ( INCOME, EXPENSES )
  RETURNS DECIMAL(8,2) SOURCE "-" ( DECIMAL, DECIMAL)
```

Alternatively, the distinct type can be cast to a built-in data type and the result used as an operand of an arithmetic operator.

With the concatenation operator

When two strings operands are concatenated, the result of the expression is a string.

The operands of concatenation must be compatible strings. A binary string cannot be concatenated with a character string, including character strings that are defined as FOR BIT DATA (for more information on the compatibility of data types, see the compatibility matrix in Table 20 on page 102). A distinct type that is based on a string type can be concatenated only if an appropriate user-defined function is created.

Both CONCAT and the vertical bars (||) represent the concatenation operator. Vertical bars (or the characters that must be used in place of vertical bars in some countries¹³) can cause parsing errors in statements passed from one DBMS to another. The problem occurs if the statement undergoes character conversion with certain combinations of source and target CCSIDs¹³. Thus, CONCAT is the preferable concatenation operator.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second.

The following table shows how the string operands determine the data type and the length attribute of the result (the order in which the operands are concatenated has no effect on the result).

Table 40. Data type and length of concatenated operands

If one operand column is	And the other operand is	The data type of the result column is ¹
CHAR(x)	CHAR(y) with a combined length attribute that is less than 256	CHAR(x+y) ²
	CHAR(y) with a combined length attribute that is greater than 255	VARCHAR(MIN(x'+y',32704)) ³
	VARCHAR(y)	
VARCHAR(x)	VARCHAR(y)	VARCHAR(MIN(x'+y',32704)) ³
CLOB(x)	CHAR(y)	CLOB(MIN(x'+y',2G))
	VARCHAR(y)	
	CLOB(y)	
	GRAPHIC(y)	DBCLOB(MIN(x+y,1G))
	VARGRAPHIC(y)	
	DBCLOB(y)	
GRAPHIC(x)	CHAR(y)	VARGRAPHIC(MIN(x+y,16352)) ⁴
	VARCHAR(y)	
	VARGRAPHIC(y)	
VARGRAPHIC(x)	CHAR(y)	VARGRAPHIC(MIN(x+y,16352)) ⁴
	VARCHAR(y)	
	GRAPHIC(y)	
	GRAPHIC(y)	

13. In various EBCDIC code pages, DB2 supports code point combinations X'4F4F', X'BBBB', and X'5A5A' to mean concatenation. X'BBBB' and X'5A5A' are interpreted to mean concatenation only on single byte character set DB2 subsystems.

Table 40. Data type and length of concatenated operands (continued)

If one operand column is	And the other operand is	The data type of the result column is ¹
DBCLOB(x)	CHAR(y)	DBCLOB(MIN(x+y,1G))
	VARCHAR(y)	
	CLOB(y)	
	GRAPHIC(y)	
	VARGRAPHIC(y)	
	DBCLOB(y)	
BINARY(x)	BINARY(y) with a combined length attribute that is less than 256	BINARY(x+y)
	BINARY(y) with a combined length attribute that is greater than 255	VARBINARY(MIN(x+y,32704))
VARBINARY(x)	VARBINARY(y)	VARBINARY(MIN(x+y,32704))
	BINARY(y)	
BLOB(x)	BLOB(y)	BLOB(MIN(x+y, 2G))

Notes:

- 2G represents 2,147,483,647 bytes
 - 1G represents 1,073,741,823 double-byte characters
- Neither CHAR(x) nor CHAR(y) can contain mixed data. If either operand contains mixed data, the result is VARCHAR(MIN(x'+y',32704)).
- If conversion of the first operand is required, x' = 3x; otherwise, it remains x. If conversion of the second operand is required, y' = 3y; otherwise, it remains y.
- Both operands are converted to UTF-16, if necessary (that is, the operand is not already UTF-16), and the results are concatenated.

As the previous table shows, the length of the result is the sum of the lengths of the operands. However, the length of the result is two bytes less if redundant shift code characters are eliminated from the result. Redundant shift code characters exist when both character strings are EBCDIC mixed data, and the first string ends with a “shift-in” character (X'0F') and the second operand begins with a “shift-out” character (X'0E'). These two shift code characters are removed from the result.

The CCSID of the result is determined by the rules set forth in “Character conversion in set operations and concatenations” on page 666. Some consequences of those rules are the following:

- If either operand is BIT data, the result is BIT data.
- The conversion that occurs when SBCS data is compared with mixed data depends on the encoding scheme. If the encoding scheme is Unicode, the SBCS operand is converted to MIXED. Otherwise, the conversion depends on the field MIXED DATA on installation panel DSNTIPF for the DB2 that does the comparison:
 - Mixed data if the MIXED DATA option at the server is YES. The result is not necessarily well-formed mixed data.
 - SBCS data if the MIXED DATA option at the server is NO. If the mixed data cannot be converted to pure SBCS data, an error occurs.

If an operand is a string from a column with a field procedure, the operation applies to the decoded form of the value. The result does not inherit the field procedure.

One operand of concatenation can be a parameter marker. When one operand is a parameter marker, its data type and length attributes are considered to be the same as those for the operand that is not a parameter marker except for a string data type. Refer to Table 40 on page 189 for the formula used to calculate data type length for untyped parameter markers in the CONCAT operator when another operand is a string data type. The order of concatenation operations must be considered to determine these attributes in the case of nested concatenation.

No operand of concatenation can be a distinct type even if the distinct type is based on a character data type. To concatenate a distinct type, create a user-defined function that is sourced on the CONCAT operator. For example, if distinct types TITLE and TITLE_DESCRIPTION were both sourced on data type VARCHAR(25), the following user-defined function, named ATTACH, could be used to concatenate the two distinct types:

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternatively, the concatenation operator could be overloaded by using a user-defined function to add the distinct types:

```
CREATE FUNCTION "||" (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Scalar-fullselect

A *scalar-fullselect* as supported in an expression is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If more than one row is to be returned for a scalar fullselect, an error occurs.

►►—(—fullselect—)—————►◄

If a set operator is not specified in the outermost fullselect and the select list element is an expression that is simply a column name, the result column name is based on the name of the column. Otherwise, the result column is unnamed.

A scalar fullselect cannot be used in the following instances:

- A CHECK constraint in CREATE TABLE and ALTER TABLE statements
- A CREATE VIEW statement where the view definition includes the WITH CHECK option
- A CREATE FUNCTION (SQL) statement (subselect already restricted from the expression in the RETURN clause)
- An argument in a CALL statement for an input parameter
- An argument to an aggregate function
- An ORDER BY clause
- A GROUP BY clause
- A join-condition of the ON clause for INNER and OUTER JOINS

If the scalar fullselect is a subselect, it is also referred to as a scalar subselect. See “subselect” on page 632 for more information.

The following examples illustrate the use of *scalar-fullselect*. Assume that four tables (PARTS, PRODUCTS, PARTPRICE, and PARTINVENTORY) contain product data.

Example 1 - scalar-fullselect in a WHERE clause:

Find which products have the prices in the range of at least twice the lowest price of all the products and at most half the price of all the products.

```
SELECT PRODUCT, PRICE FROM PRODUCTS A
WHERE
    PRICE BETWEEN 2 * (SELECT MIN(PRICE) FROM PRODUCTS)
    AND .5 * (SELECT MAX(PRICE) FROM PRODUCTS);
```

Example 2 - scalar-fullselect in a SELECT list:

For each part, find its price and its inventory.

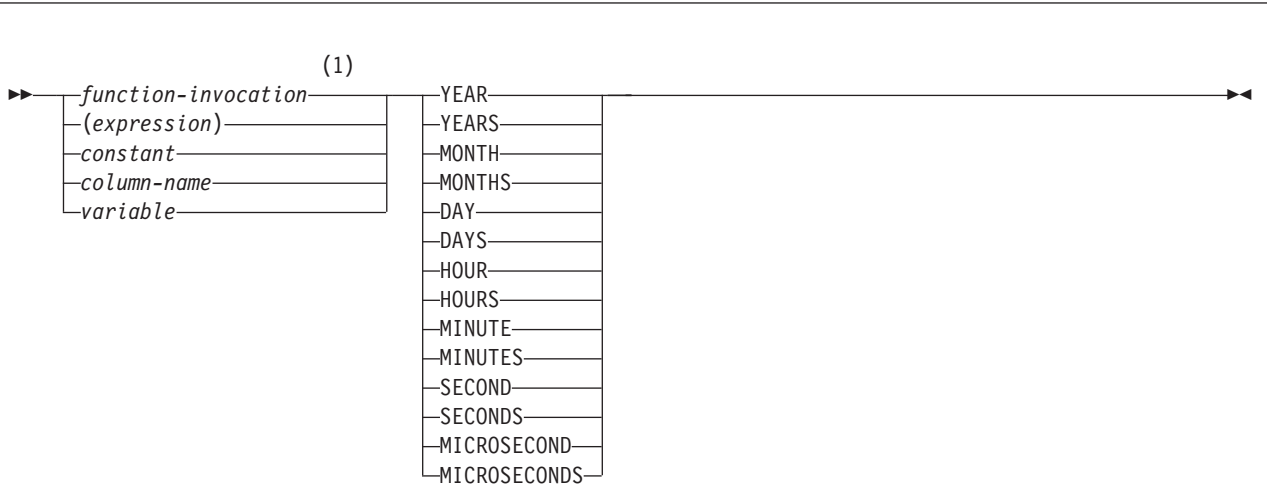
```
SELECT PART,
    (SELECT PRICE FROM PARTPRICE WHERE PART=A.PART),
    (SELECT ONHAND# FROM INVENTORY WHERE PART=A.PART)
FROM PARTS A;
```

Datetime operands and durations

Datetime values can be incremented, decremented, and subtracted. These operations can involve decimal numbers called durations. A *duration* is a positive or negative number representing an interval of time.

Labeled durations

The form a labeled duration is as follows:



Notes:

- 1 Includes all functions except table functions.

A *labeled duration* represents a specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the

seven duration keywords.¹⁴ The number specified is converted as if it were assigned to a DECIMAL(15,0) number.

A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of the data type of date, time, or timestamp. Thus, the expression `HIREDATE + 2 MONTHS + 14 DAYS` is valid, whereas the expression `HIREDATE + (2 MONTHS + 14 DAYS)` is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

Date duration

A *date duration* represents a number of years, months, and days expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. The result of subtracting one DATE value from another, as in the expression `HIREDATE - BIRTHDATE`, is a date duration.

Time duration

A *time duration* represents a number of hours, minutes, and seconds expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format *hhmmss*, where *hh* represents the number of hours, *mm* the number of minutes, and *ss* the number of seconds. The result of subtracting one TIME value from another is a time duration.

Timestamp duration

A *timestamp duration* represents a number of years, months, days, hours, minutes, seconds, and microseconds expressed as a DECIMAL(20,6) number. To be properly interpreted, the number must have the format *yyyyxxddhhmmssnnnnnn*, where *yyyy*, *xx*, *dd*, *hh*, *mm*, *ss*, and *nnnnnn* represent, respectively, the number of years, months, days, hours, minutes, seconds, and microseconds. The result of subtracting one TIMESTAMP value from another is a timestamp duration.

Datetime arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction.

If a datetime value is the operand of addition, the other operand must be a duration. The specific rules governing the use of the addition operator with datetime values follow.

- If one operand is a date, the other operand must be a date duration or labeled duration of years, months, or days.
- If one operand is a time, the other operand must be a time duration or a labeled duration of hours, minutes, or seconds.
- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be a parameter marker. For a discussion of parameter markers, see Parameter markers in “PREPARE” on page 1405.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a

14. The singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of years, months, or days.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.
- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of hours, minutes, or seconds.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.
- Neither operand of the subtraction operator can be a parameter marker.

When an operand in a datetime expression is a string, it might undergo character conversion before it is interpreted and converted to a datetime value. When its CCSID is not that of the default for mixed strings, a mixed string is converted to the default mixed data representation. When its CCSID is not that of the default for SBCS strings, an SBCS string is converted to the default SBCS representation.

Date arithmetic

Date values can be subtracted, incremented, or decremented.

Subtracting dates: The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation $RESULT = DATE1 - DATE2$.

Date subtraction: result = date1 - date2

- If $DAY(DATE2) \leq DAY(DATE1)$ then $DAY(RESULT) = DAY(DATE1) - DAY(DATE2)$
- If $DAY(DATE2) > DAY(DATE1)$ then $DAY(RESULT) = N + DAY(DATE1) - DAY(DATE2)$ where N = the last day of $MONTH(DATE2)$. $MONTH(DATE2)$ is then incremented by 1.
- If $MONTH(DATE2) \leq MONTH(DATE1)$ then $MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2)$
- If $MONTH(DATE2) > MONTH(DATE1)$ then $MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2)$ and $YEAR(DATE2)$ is incremented by 1.
- $YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2)$

For example, the result of $DATE('3/15/2005') - '12/31/2004'$ is 215 (or, a duration of 0 years, 2 months, and 15 days). In this example, notice that the second operand did not need to be converted to a date. According to one of the rules for subtraction, described under “Datetime arithmetic in SQL” on page 193, the second operand can be a string representation of a date if the first operand is a date.

Incrementing and decrementing dates: The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this

operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive. If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. Here the day portion of the result is set to 28, and the SQLWARN6 field of the SQLCA is set to W, indicating that an end-of-month adjustment was made to correct an invalid date. *DB2 Application Programming and SQL Guide* also describes how SQLWARN6 is set.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case the day is set to the last day of the month, and the SQLWARN6 field of the SQLCA is set to W to indicate the adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year. Adding or subtracting a duration of days will not cause an end-of-the-month adjustment.

Date durations, whether positive or negative, can also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and SQLWARN6 is set to W to indicate any necessary end-of-month adjustment.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1+X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS
```

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1-X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS
```

Adding a month to a date gives the same day one month later unless that day does not exist in the later month. In that case, the day in the result is set to the last day of the later month. For example, January 28 plus one month gives February 28; one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29. If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

If one or more months are added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date. In addition, logically equivalent expressions might not produce the same result. For example, the following two expressions do not produce the same result:

```
(DATE('2005 01 31') + 1 MONTH) + 1 MONTH    -- results in 2005-03-28
DATE('2005 01 31') + 2 MONTHS                -- results in 2005-03-31
```

The order in which labeled date durations are added to and subtracted from dates can affect the results. When you add labeled date durations to a date, specify them in the order of YEARS + MONTHS + DAYS. When you subtract labeled date

durations from a date, specify them in the order of DAYS - MONTHS - YEARS. For example, to add one year and one day to a date, specify:

```
DATE1 + 1 YEAR + 1 DAY
```

To subtract one year, one month, and one day from a date, specify:

```
DATE1 - 1 DAY - 1 MONTH - 1 YEAR
```

Time arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting times: The result of subtracting one time (*TIME2*) from another (*TIME1*) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0). If *TIME1* is greater than or equal to *TIME2*, *TIME2* is subtracted from *TIME1*. If *TIME1* is less than *TIME2*, however, *TIME1* is subtracted from *TIME2*, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation $RESULT = TIME1 - TIME2$.

Time subtraction: $result = time1 - time2$

- If $SECOND(TIME2) \leq SECOND(TIME1)$ then $SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2)$.
- If $SECOND(TIME2) > SECOND(TIME1)$ then $SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2)$ and $MINUTE(TIME2)$ is incremented by 1.
- If $MINUTE(TIME2) \leq MINUTE(TIME1)$ then $MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2)$.
- If $MINUTE(TIME2) > MINUTE(TIME1)$ then $MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2)$ and $HOUR(TIME2)$ is incremented by 1.
- $HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2)$.

For example, the result of $TIME('11:02:26') - '00:32:56'$ is '102930' (a duration of 10 hours, 29 minutes, and 30 seconds). In this example, notice that the second operand did not need to be converted to a time. According to one of the rules for subtraction, described under "Datetime arithmetic in SQL" on page 193, the second operand can be a string representation of a time if the first operand is a time.

Incrementing and decrementing times: The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. Adding 24 hours to the time '00:00:00' results in the time '24:00:00'. However, adding 24 hours to any other time results in the same time; for example, adding 24 hours to the time '00:00:59' results in the time '00:00:59'. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds affects the seconds portion of the time and might affect the minutes and hours.

Time durations, whether positive or negative, can also be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. Thus, $TIME1 + X$, where *X* is a positive DECIMAL(6,0) number, is equivalent to the expression

$TIME1 + HOUR(X) \text{ HOURS} + MINUTE(X) \text{ MINUTES} + SECOND(X) \text{ SECONDS}$

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.

Subtracting timestamps: The result of subtracting one timestamp ($TS2$) from another ($TS1$) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps. The data type of the result is `DECIMAL(20,6)`. If $TS1$ is greater than or equal to $TS2$, $TS2$ is subtracted from $TS1$. If $TS1$ is less than $TS2$, however, $TS1$ is subtracted from $TS2$ and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation $RESULT = TS1 - TS2$.

Timestamp subtraction: $result = ts1 - ts2$

- If `MICROSECOND($TS2$) <= MICROSECOND($TS1$)` then `MICROSECOND($RESULT$) = MICROSECOND($TS1$) - MICROSECOND($TS2$)`.
- If `MICROSECOND($TS2$) > MICROSECOND($TS1$)` then `MICROSECOND($RESULT$) = 1000000 + MICROSECOND($TS1$) - MICROSECOND($TS2$)` and `SECOND($TS2$)` is incremented by 1.
- If `HOUR($TS2$) <= HOUR($TS1$)` then `HOUR($RESULT$) = HOUR($TS1$) - HOUR($TS2$)`.
- If `HOUR($TS2$) > HOUR($TS1$)` then `HOUR($RESULT$) = 24 + HOUR($TS1$) - HOUR($TS2$)` and `DAY($TS2$)` is incremented by 1.

The seconds and minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

Incrementing and decrementing timestamps: The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp, is itself a timestamp. Date and time arithmetic is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. When the result of an operation is midnight, the time portion of the result can be '24.00.00' or '00.00.00'; a comparison of those two values does not result in 'equal'. Microseconds overflow into seconds.

Precedence of operations

Expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication, division, and concatenation are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

Example 1: In this example, the first operation is the addition in `(SALARY + BONUS)` because it is within parenthesis. The second operation is multiplication because it is a higher precedence level than the second addition operator and it is to the left of the division operator. The third operation is division because it is at a higher precedence level than the second addition operator. Finally, the remaining addition is performed.

$1.10 * (SALARY + BONUS) + SALARY / :VAR3$
 (2) (1) (4) (3)

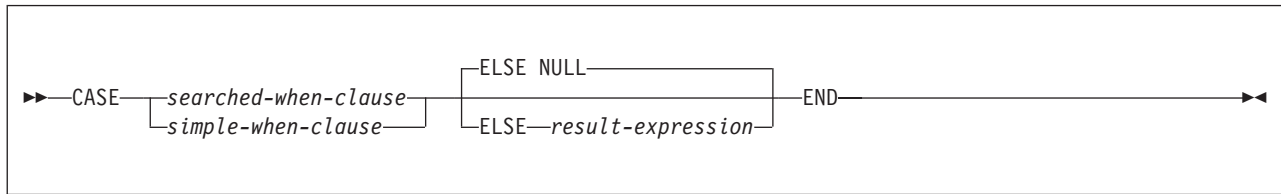
Example 2: In this example, the first operation (`CONCAT`) combines the character strings in the variables `YYYYMM` and `DD` into a string representing a date. The

second operation (-) then subtracts that date from the date being processed in DATECOL. The result is a date duration that indicates the time elapsed between the two dates.

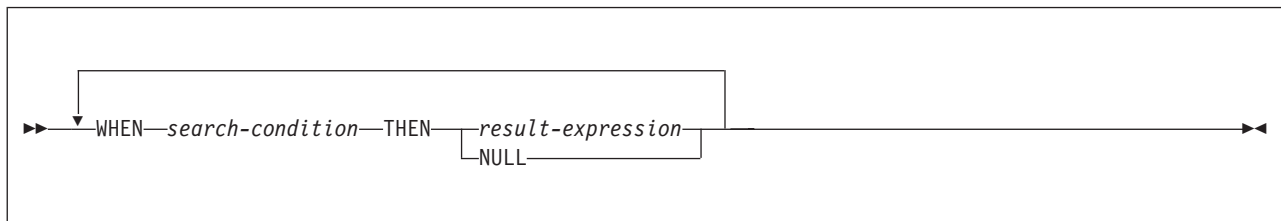
```
DATECOL - :YYYYMM CONCAT :DD  
          (2)          (1)
```


CASE expressions

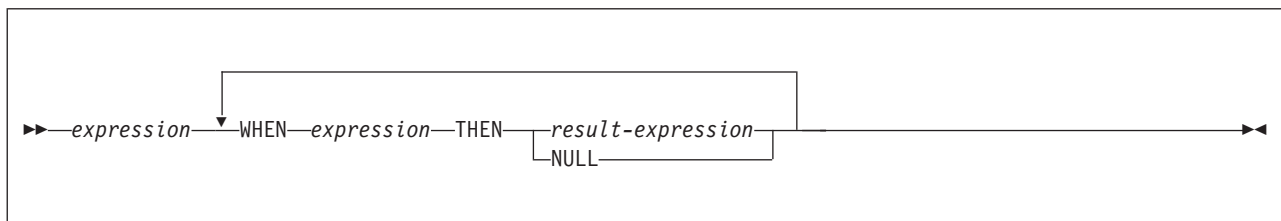
A CASE expression allows an expression to be selected based on the evaluation of one or more conditions.



searched-when-clause:



simple-when-clause:



In general, the value of the *case-expression* is the value of the *result-expression* following the first (leftmost) *when-clause* that evaluates to true. If no case evaluates to true and the **ELSE** keyword is present, the result is the value of the *result-expression* or **NULL**. If no case evaluates to true and the **ELSE** keyword is not present, the result is **NULL**. When a case evaluates to unknown (because of **NULL** values), the case is **NOT** true and hence is treated the same way as a case that evaluates to false.

searched-when-clause

Specifies a *search-condition* that is applied to each row or group of table data presented for evaluation, and the result when that condition is true.

simple-when-clause

Specifies that the value of the *expression* prior to the first **WHEN** keyword is tested for equality with the value of each *expression* that follows the **WHEN** keyword. It also specifies the result for when that condition is true.

The data type of the *expression* prior to the first **WHEN** keyword must be compatible with the data types of the *expression* that follows each **WHEN** keyword. The data type of any of the expressions cannot be a **CLOB**, **DBCLOB** or **BLOB**. In addition, the *expression* prior to the first **WHEN** keyword cannot include a function that is not deterministic or has an external action.

result-expression **or** **NULL**

Specifies the value that follows the THEN and ELSE keywords. It specifies the result of a *searched-when-clause* or a *simple-when-clause* that is true, or the result if no case is true. There must be at least one *result-expression* in the CASE expression with a defined data type. NULL cannot be specified for every case.

All *result-expressions* must have compatible data types. The attributes of the result are determined according to the rules that are described in “Rules for result data types” on page 119. When the result is a string, its attributes include a CCSID. For the rules on how the CCSID is determined, see “Determining the encoding scheme and CCSID of a string” on page 43.

search-condition

Specifies a condition that is true, false, or unknown about a row or group of table data. The *search-condition* can be a predicate, including predicates that contain fullselects (scalar or non-scalar) or row-value expressions.

If the CASE expression is in a select list, a SET clause, a VALUES clause of an INSERT or MERGE statement, or an IN predicate, the *search-condition* cannot be a quantified predicate, an IN predicate, or an EXISTS predicate. However, an IN predicate is allowed within a select list or a VALUES clause if that IN predicate explicitly includes a list of expressions.

END

Ends a *case-expression*.

Two scalar functions, NULLIF and COALESCE, are specialized to handle a subset of the functionality provided by CASE. The following table shows the equivalent expressions using CASE or these functions.

Table 41. Equivalent case expressions

CASE expression	Equivalent expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)

Example 1 (simple-when-clause): Assume that in the EMPLOYEE table the first character of a department number represents the division in the organization. Use a CASE expression to list the full name of the division to which each employee belongs.

```
SELECT EMPNO, LASTNAME,  
       CASE SUBSTR(WORKDEPT,1,1)  
         WHEN 'A' THEN 'Administration'  
         WHEN 'B' THEN 'Human Resources'  
         WHEN 'C' THEN 'Design'  
         WHEN 'D' THEN 'Operations'  
       END  
FROM EMPLOYEE;
```

Example 2 (searched-when-clause): You can also use a CASE expression to avoid “division by zero” errors. From the EMPLOYEE table, find all employees who earn more than 25 percent of their income from commission, but who are not fully paid on commission:

```
SELECT EMPNO, WORKDEPT, SALARY+COMM FROM EMPLOYEE
WHERE (CASE WHEN SALARY=0 THEN 0
          ELSE COMM/(SALARY+COMM)
        END) > 0.25;
```

Example 3 (searched-when-clause): You can use a CASE expression to avoid “division by zero” errors in another way. The following queries show an accumulation or summing operation. In the first query, DB2 performs the division before performing the CASE statement and an error occurs along with the results.

```
SELECT REF_ID,PAYMT_PAST_DUE_CT,
       CASE
         WHEN PAYMT_PAST_DUE_CT=0 THEN 0
         WHEN PAYMT_PAST_DUE_CT>0 THEN
           SUM(BAL_AMT/PAYMT_PAST_DUE_CT)
       END
FROM PAY_TABLE
GROUP BY REF_ID,PAYMT_PAST_DUE_CT;
```

However, if the CASE expression is included in the SUM aggregate function, the CASE expression would prevent the errors. In the following query, the CASE expression screens out the unwanted division because the CASE operation is performed before the division.

```
SELECT REF_ID,PAYMT_PAST_DUE_CT,
       SUM(CASE
         WHEN PAYMT_PAST_DUE_CT=0 THEN 0
         WHEN PAYMT_PAST_DUE_CT>0 THEN
           BAL_AMT/PAYMT_PAST_DUE_CT
       END)
FROM PAY_TABLE
GROUP BY REF_ID,PAYMT_PAST_DUE_CT;
```

Example 4: This example shows how to group the results of a query by a CASE expression without having to re-type the expression. Using the sample employee table, find the maximum, minimum, and average salary. Instead of finding these values for each department, assume that you want to combine some departments into the same group.

```
SELECT CASE_DEPT,MAX(SALARY),MIN(SALARY),AVG(SALARY)
FROM (SELECT SALARY,CASE WHEN WORKDEPT = 'A00' OR WORKDEPT = 'E21'
                        THEN 'A00_E21'
                        WHEN WORKDEPT = 'D11' OR WORKDEPT = 'E11'
                        THEN 'D11_E11'
                        ELSE WORKDEPT
                      END AS CASE_DEPT
FROM DSN8910.EMP) X
GROUP BY CASE_DEPT;
```

CAST specification

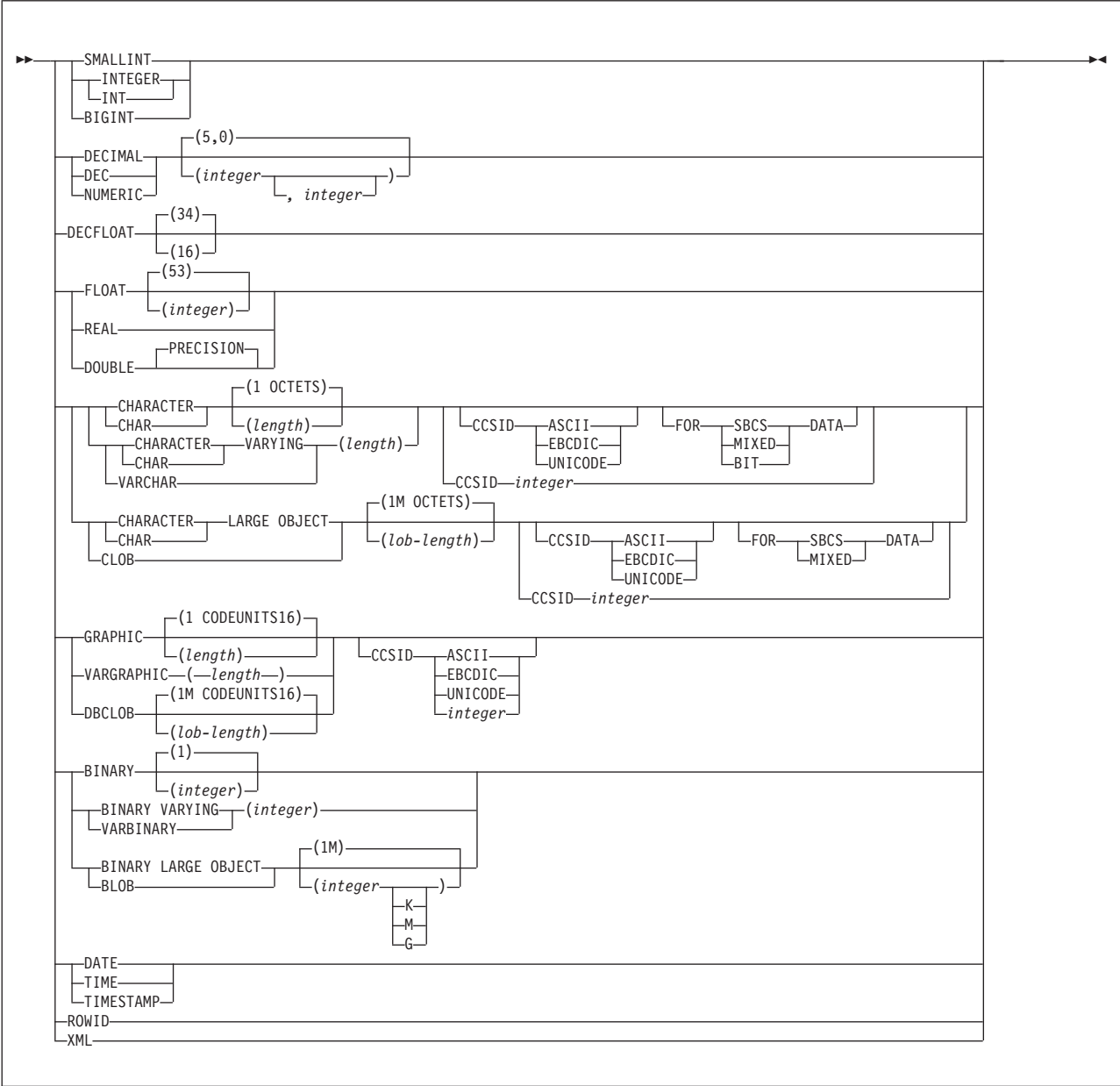
The CAST specification returns the first operand (the cast operand) converted to the data type that is specified by *data-type*.

►►—CAST—(—*expression*—
 |—NULL—
 |—*parameter-marker*—)—AS—*data-type*—)►◄

data-type:

►►—*built-in-type*—
 |—*distinct-type-name*—)►◄

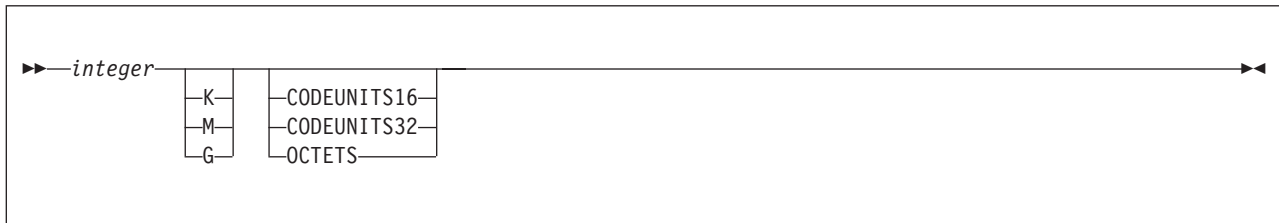
built-in-type:



length:



lob-length:



If the data type of either operand is a distinct type, the privilege set must implicitly include EXECUTE authority on the generated cast functions for the distinct type. The CAST specification allows the second operand to be cast to a particular encoding scheme or CCSID if the second operand represents character data. The CCSID clause can be specified following CHAR, VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, and DBCLOB data types.

expression

Specifies that the cast operand is an expression other than NULL or a parameter marker. The result is the value of the operand value converted to the specified target *data type*.

The supported casts are shown in “Casting between data types” on page 96. If the cast is not supported, an error is returned.

When a character string is cast to a character string with a different length or a graphic string is cast to a graphic string with a different length, a warning occurs if any characters except trailing blanks are truncated. A warning also occurs if any bytes are truncated when a binary string is cast.

NULL

Specifies that the cast operand is null. The result is a null value with the specified target *data type*.

parameter-marker

A parameter marker, which is normally considered an expression, has a special meaning as a cast operand. When the cast operand is a *parameter-marker*, the *data type* that is specified represents the “promise” that the replacement value for the parameter marker will be assignable to the specified data type (using “store assignment” rules). Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers are treated like any other typed value for the purpose of function resolution, a DESCRIBE of a select list, or column assignment.

data-type

Specifies the data type of the result. If the data type is not qualified, the SQL path is used to find the appropriate data type. For more information, see “SQL path” on page 56. For a description of *data-type*, see “CREATE TABLE” on page 1079. (For portability across operating systems, when specifying a floating-point data type, use REAL or DOUBLE instead of FLOAT.)

- If the cast operand is *expression*, see “Casting between data types” on page 96 and use any of the target data types that are supported for the data type of the cast operand.
- If the cast operand is NULL, you can use any data type.
- If the cast operand is a *parameter-marker*, you can use any data type. If the data type is a distinct type, the application that uses the parameter marker will use the source data type of the distinct type.

length

Specifies the length of the result.

You can specify that the length of the result be evaluated in a specific number of string units: CODEUNITS16, CODEUNITS32, or OCTETS. If *expression* is a character string that is defined as bit data, CODEUNITS16, or CODEUNITS32 cannot be specified. If *expression* is a graphic string, OCTETS cannot be specified.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

lob-length

Specifies the length of the result.

You can specify that the length of the result be evaluated in a specific number of string units: CODEUNITS16, CODEUNITS32, or OCTETS. If *expression* is a graphic string, OCTETS cannot be specified.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

CCSID *encoding-scheme*

Specifies the encoding scheme for the target data type. The specific CCSIDs for SBCS, BIT, and MIXED data are determined by the default CCSIDs for the server for the specified encoding scheme. The valid values are ASCII, EBCDIC, and UNICODE.

CCSID *integer*

Specifies that the target data type be encoded using the CCSID *integer*. If the second operand is CHAR, VARCHAR, or CLOB, the CCSID specified must be either a SBCS, or MIXED CCSID, or 65535 for bit data. If the second operand is GRAPHIC, VARGRAPHIC, or DBCLOB, the CCSID specified must be a DBCS CCSID. See Determining the CCSID of the result if neither CCSID *integer* nor CCSID *encoding-scheme* is specified. See Determining the CCSID of the result for special considerations regarding CCSID 367.

Interaction between length and CCSID clauses: If both the *length* and CCSID clauses are specified, the data is first cast to the specified CCSID, and then the *length* is applied. If either CODEUNITS16 or CODEUNITS32 is specified, the specification of length applies to the units specified. That is, the data is converted to an intermediate form (in Unicode), the length is applied, and the data is converted to the specified CCSID.

Resolution of cast functions: DB2 uses the implicit or explicit schema name and the data type name of *data-type*, and function resolution to determine the specific function to use to convert *expression* to *data-type*. See Qualified function resolution for more information.

Result of the CAST: When numeric data is cast to character data, the data type of the result is a fixed-length character string, which is similar to the result that the CHAR function would give. (For more information, see “CHAR” on page 302.) When character data is cast to numeric data, the data type of the result depends on the data type of the specified number. For example, character data that is cast to an integer becomes a large integer, which is similar to the result that the INTEGER function would give. (For more information see “INTEGER or INT” on page 392.)

If the data type of the result is character, the subtype of the result is determined as follows:

- If *expression* is graphic, the subtype of the result is mixed.
- If *expression* is a datetime data type, the subtype of the result is mixed.¹⁵
- If *expression* is a row ID and *data-type* is not CLOB, the result is bit data.
- If *expression* is character, the subtype of the result is the same as *expression*.
- Otherwise, the subtype depends on the encoding scheme of the result. If the encoding scheme of the result is not Unicode and the field MIXED DATA on installation panel DSNTIPF is NO, the subtype of the result is SBCS. Otherwise, the subtype of the result is mixed.

Casting constant values to DECFLOAT: To cast a constant value, where the value is negative zero, or a floating point constant to DECFLOAT, specify the value as a character string constant rather than a numeric constant. For example:

```
DECFLOAT('-0')           -- causes DB2 to retain the negative sign for a
                          -- value of negative zero
DECFLOAT('1.00E20')     -- causes DB2 to preserve the precision of the
                          -- floating point constant
```

Determining the CCSID and encoding scheme of the result: The CCSID of the result depends on whether the CCSID clause was specified and the context in which the CAST specification was specified.

If the CCSID clause was specified, the CCSID clause is used to determine the CCSID of the result as follows:

- If the CCSID clause was specified with EBCDIC, ASCII, or UNICODE, the clause determines the encoding scheme of the result. The CCSID of the result is the appropriate CCSID (from DECP) for that encoding scheme for the data type of the result.
- If the CCSID clause was specified with a numeric value representing bit data (65535), the CCSID of the result depends on the data type of the source. If the source data is not string data, the CCSID of the result is the appropriate CCSID for the application encoding scheme. See Note 1 in Table 4 on page 44. If the source is string data, the encoding scheme of the result is the same as the encoding scheme of *expression*, but the result is considered bit data.
- If the CCSID clause was specified with a numeric value, the value must be one of the CCSID values in DECP, and that number is the CCSID of the result. The encoding scheme of the result is determined from the numeric CCSID. In a CAST specification, CCSID 367 refers to ASCII data. For example, assume that MYDATA is string data to be cast to CHAR(10). The following CAST specification returns ASCII SBCS data:

```
CAST(MYDATA AS CHAR(10) CCSID 367)
```

To explicitly cast the data to Unicode SBCS, use the following syntax:

```
CAST(MYDATA AS CHAR(10) CCSID UNICODE
      FOR SBCS DATA)
```

If the CCSID clause was not specified, the CCSID of the result is 65535 if the result is bit data. Otherwise, if the data type of the result is a character or graphic string data type, the encoding scheme and CCSID of the result are determined as follows:

15. The exception is when the default encoding scheme is EBCDIC and there is no mixed or graphic data on the system for EBCDIC.

- If the *expression* and *data-type* are both character, the encoding scheme of the result is the same as *expression*. For example, assume CHAR_COL is a character column in the following:

```
CAST(CHAR_COL AS VARCHAR(25))
```

The result of the CAST is a varying length string with the same encoding scheme as the input. The CCSID of the result is the appropriate CCSID for the encoding scheme and subtype of the result.

- If the *expression* and *data-type* are both graphic, the encoding scheme and CCSID of the result is the same as *expression*.
- If the result is string and the *expression* is datetime, the result CCSID is the appropriate CCSID of the *expression* encoding scheme and the result subtype is the appropriate subtype of the CCSID.
- If the result is character, the encoding scheme and CCSID of the result depends on the context in which the CAST specification is specified:

- If the statement follows the rules that are described for type 1 statements in “Determining the encoding scheme and CCSID of a string” on page 43, the CCSID is determined as follows:

- If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the result.
- Otherwise, the default EBCDIC encoding scheme is used for the result.

The CCSID of the result is the appropriate CCSID for the encoding scheme and subtype of the result.

- Otherwise, the CCSID of the result is the appropriate CCSID for the application encoding scheme and subtype of the result.

- If the result is graphic, the encoding scheme and the CCSID of the result depends on the context in which the CAST specification is specified:

- If the statement follows the rules that are described for type 1 statements in “Determining the encoding scheme and CCSID of a string” on page 43, the CCSID is determined as follows:

- If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the result.
- Otherwise, the default EBCDIC encoding scheme is used for the result.

The CCSID of the result is the appropriate CCSID for the encoding scheme and data type of the result.

- Otherwise, the CCSID of the result is the appropriate CCSID for the application encoding scheme of the result.

- Otherwise, the CCSID of the result depends on the context in which the CAST specification was specified.

- If the statement follows the rules that are described for type 1 in statements in “Determining the encoding scheme and CCSID of a string” on page 43, the CCSID is determined as follows:

- If the statement references a table or view, the encoding scheme of that table or view determines the encoding scheme for the result.
- Otherwise, the default EBCDIC encoding scheme is used for the result.

The CCSID of the result is the appropriate CCSID for the encoding scheme and data type of the result.

Alternative syntax for casting distinct types: There is alternative syntax for casting a distinct type to its source data type and vice versa. Assume that a distinct type D_MONEY was defined with the following statement and column MONEY was defined with that data type.

```
CREATE TYPE D_MONEY AS DECIMAL(9,2);
```

DECIMAL(MONEY) is equivalent syntax to CAST(MONEY AS DECIMAL(9,2)). Both forms of the syntax use the cast function that DB2 generated when the distinct type D_MONEY was created to convert the distinct type to its source type of DECIMAL(9,2).

However, it is possible that different cast functions might be chosen for the equivalent syntax forms because of the difference in function resolution, particularly the treatment on unqualified names. Although the process of function resolution is similar for both, in the CAST specification as described above, DB2 uses the schema name of the target data type to locate the function. Therefore, if an unqualified data type name is specified as the target data type, DB2 uses the SQL path to resolve the schema name of the distinct type and then searches for the function in that schema. In function notation, when an unqualified function name is specified, DB2 searches the schemas in the SQL path to find an appropriate function match, as described under “Function resolution” on page 175. For example, assume that you defined the following distinct types, which implicitly gives you both USAGE authority on the distinct types and EXECUTE authority on the cast functions that are generated for them:

```
CREATE TYPE SCHEMA1.AGE AS DECIMAL(2,0);
    one of the generated cast functions is:
    FUNCTION SCHEMA1.AGE(SYSIBM.DECIMAL(2,0)) RETURNS SCHEMA1.AGE
CREATE TYPE SCHEMA2.AGE AS INTEGER;
    one of the generated cast functions is:
    FUNCTION SCHEMA2.AGE(SYSIBM.INTEGER) RETURNS SCHEMA2.AGE
```

If *STU_AGE*, an INTEGER host variable, is cast to the distinct type with either of the following statements and the SQL path is SYSIBM, SCHEMA1, SCHEMA2:

```
Syntax 1: CAST(:STU_AGE AS AGE);
Syntax 2: AGE(:STU_AGE);
```

different cast functions are chosen. For syntax 1, DB2 first resolves the schema name of distinct type AGE as SCHEMA1 (the first schema in the path that contains a distinct type named AGE for which you have EXECUTE authority for the appropriate generated cast function). Then it looks for a suitable function in that schema and chooses SCHEMA1.AGE because the data type of *STU_AGE*, which is INTEGER, is promotable to the data type of the function argument, which is DECIMAL(2,0). For syntax 2, DB2 searches all the schemas in the path for an appropriate function and chooses SCHEMA2.AGE. DB2 selects SCHEMA2.AGE over SCHEMA1.AGE because the data type of its argument (INTEGER) is an exact match for *STU_AGE* (INTEGER) and, therefore, a better match than the argument for SCHEMA1.AGE, which is DECIMAL(2,0).

Example 1: Assume that an application needs only the integer portion of the SALARY column, which is defined as DECIMAL(9,2) from the EMPLOYEE table. The following query for the employee number and the integer value of SALARY could be prepared.

```
SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE;
```

Example 2: Assume that two distinct types exist in schema SCHEMAX. Distinct type D_AGE was based on SMALLINT and is the data type for the AGE column in

the PERSONNEL table. Distinct type D_YEAR was based on INTEGER and is the data type for the RETIRE_YEAR column in the same table. The following UPDATE statement could be prepared.

```
UPDATE PERSONNEL SET RETIRE_YEAR =?
      WHERE AGE = CAST( ? AS SCHEMAX.D_AGE);
```

The first parameter is an untyped parameter marker that has a data type of RETIRE_YEAR. However, the application will use an integer for the parameter marker. The parameter marker does not need to be cast because the SET is an assignment.

The second parameter marker is a typed parameter marker that is cast to the distinct type D_AGE. Casting the parameter marker satisfies the requirement that comparisons must be performed with compatible data types. The application will use the source data type, SMALLINT, to process the parameter marker.

Example 3: A CAST specification can be used to explicitly specify the data type of a parameter in a context where a parameter marker must be typed. In the following example, the CAST specification is used to tell DB2 to assume that the value that will be provided as input to the TIME function will be CHAR(20). See “PREPARE” on page 1405 for a list of contexts when invoking functions where parameter markers can be untyped. For all other contexts when invoking a function, the CAST specification can be used to explicitly specify the type of a parameter marker.

```
INSERT INTO ADMF001.CASTSQLJ VALUES( TIME(CAST(? AS CHAR(20)) ) )
```

Example 4: Assume that an application wants to cast an EBCDIC string to Unicode UTF-8. The string contains the value 'Jürgen', which is 6 bytes in ASCII or EBCDIC and is 7 bytes in Unicode UTF-8. In the following query, the CAST specification is invoked with the *length* clause with CODEUNITS32 specified to ensure that the data is not truncated. (In this case, CODEUNITS16 could also be specified as the string unit.)

```
SELECT CAST('Jürgen' AS VARCHAR(6 CODEUNITS32) CCSID UNICODE)
      FROM SYSIBM.SYSDUMMY1;
```

For this query, the data is converted from EBCDIC to Unicode UTF-16, the length clause is applied, and then the UTF-16 result is converted to UTF-8.


```
| XMLCAST(XMLQUERY('/PRODUCT/QUANTITY'  
| PASSING xmlcol) AS INTEGER)
```

| *Example 3:* Convert a value extracted from an XMLQUERY expression into a
| varying-length character string:

```
| XMLCAST(XMLQUERY('/PRODUCT/NAME'  
| PASSING xmlcol) AS VARCHAR(20))
```

| Note that in the above two examples, if the XMLQUERY returns a sequence of
| more than one node, the XMLCAST specification will return an error.

| *Example 4:* Convert a value extracted from an SQL scalar subquery into an XML
| value:

```
| XMLCAST((SELECT quantity FROM product AS p  
| WHERE p.id = 1077) AS XML)
```

|

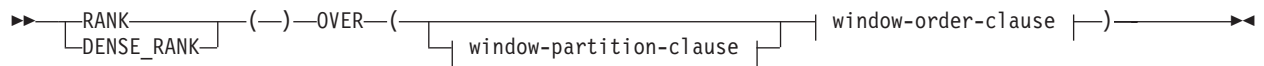
OLAP specification

Online analytical processing (OLAP) specifications provide the ability to return ranking and row numbering information as a scalar value in the result of a query. An OLAP specification can be included in an expression, in a *select-list*, or in the ORDER BY clause of a *select-statement*. The query result to which the OLAP specification is applied is the result table of the innermost subselect that includes the OLAP specification.

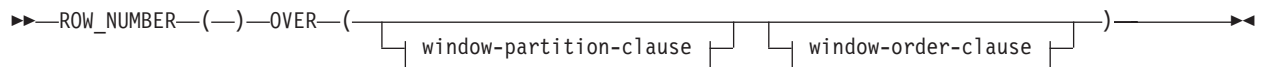
OLAP-specification



ordered-OLAP-specification



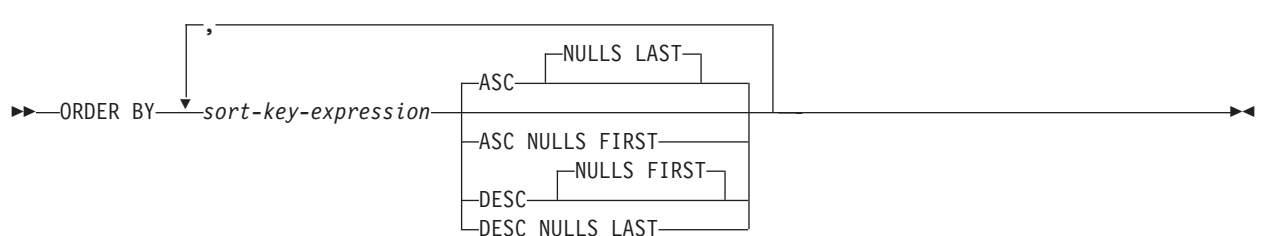
numbering-specification



window-partition-clause



window-order-clause



`RANK`, `DENSE_RANK`, and `ROW_NUMBER` are sometimes called window functions.

An OLAP specification is not valid in a **WHERE**, **VALUES**, **GROUP BY**, **HAVING**, or **SET** clause. An OLAP specification cannot be used as an argument of an aggregate function.

When invoking an OLAP specification, a window is specified that defines the rows over which the function is applied and in which order.

The result of a **RANK**, **DENSE_RANK**, or **ROW_NUMBER** specification is **BIGINT**. The result cannot be null.

RANK or DENSE_RANK

Specifies that the ordinal rank of a row within the specified window is computed. Rows that are not distinct with respect to the ordering within the specified window are assigned the same rank. The results of the ranking can be defined with or without gaps in the numbers that result from duplicate values.

RANK

Specifies that the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, there will be one or more gaps in the sequential rank numbering.

DENSE_RANK

Specifies that the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

ROW_NUMBER

Specifies that a sequential row number is computed for the row that is defined by the ordering, starting with 1 for the first row. If the **ORDER BY** clause is not specified in the window, the row numbers are assigned to the rows in an arbitrary order, as the rows are returned (not according to any **ORDER BY** clause in the *select-statement*).

PARTITION BY (*partitioning-expression*,...)

Defines the partition within which the OLAP operation is applied. A *partitioning-expression* is an expression that is used in defining the partitioning of the result table. Each column name that is referenced in a *partitioning-expression* must unambiguously reference a column of the result table of the subselect that contains the OLAP specification. A *partitioning-expression* cannot include a *scalar-fullselect* or any function that is not deterministic or has an external action.

ORDER BY (*sort-key-expression*,...)

Defines the ordering of rows within a partition that is used to determine the value of the OLAP specification. It does not define the ordering of the result table.

sort-key-expression

Specifies an expression to use in defining the ordering of the rows within a window partition. Each column name that is referenced in a *sort-key-expression* must unambiguously reference a column of the result table of the subselect, including the OLAP specification. A *sort-key-expression* cannot include a scalar fullselect or any function that is not deterministic or that has an external action.

ASC

Specifies that the values of *sort-key-expression* are used in ascending order.

DESC

Specifies that the values of *sort-key-expression* are used in descending order.

NULLS FIRST

Specifies that the window ordering considers null values before all non-null values in the sort order.

NULLS LAST

Specifies that the window ordering considers null values after all non-null values in the sort order.

Notes

Syntax alternatives and synonyms: For compatibility, the keywords **DENSERANK** and **ROWNUMBER** can be used as synonyms for **DENSE_RANK** and **ROW_NUMBER** respectively.

Example 1: Display the ranking of employees that have a total salary of more than \$30,000, in order by last name:

```
SELECT EMPNO, LASTNAME, FIRSTNAME, SALARY+BONUS AS TOTAL_SALARY,  
       RANK() OVER(ORDER BY SALARY+BONUS DESC) AS RANK_SALARY  
FROM EMP WHERE SALARY+BONUS > 30000  
ORDER BY LASTNAME;
```

If the result is to be ordered by rank, **ORDER BY LASTNAME** would be replaced with **ORDER BY RANK_SALARY**.

Example 2: Rank the departments according to their average total salary:

```
SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,  
       RANK() OVER(ORDER BY AVG(SALARY+BONUS) DESC) AS RANK_AVG_SAL  
FROM EMP  
GROUP BY WORKDEPT  
ORDER BY RANK_AVG_SAL;
```

Example 3: Rank the departments according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value:

```
SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNAME, EDLEVEL,  
       DENSE_RANK() OVER  
         (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC) AS RANK_EDLEVEL  
FROM EMP  
ORDER BY WORKDEPT, LASTNAME;
```

Example 4: Provide row numbers in the results of a query:

```
SELECT ROW_NUMBER() OVER(ORDER BY WORKDEPT, LASTNAME) AS NUMBER,  
       LASTNAME, SALARY  
FROM EMP  
ORDER BY WORKDEPT, LASTNAME;
```

Example 5: List the top five wage earners:

```
SELECT EMPNO, LASTNAME, FIRSTNAME, TOTAL_SALARY, RANK_SALARY  
FROM (SELECT EMPNO, LASTNAME, FIRSTNAME, SALARY+BONUS AS TOTAL_SALARY,  
       RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY  
FROM EMP) AS RANKED_EMPLOYEE  
WHERE RANK_SALARY < 6  
ORDER BY RANK_SALARY;
```

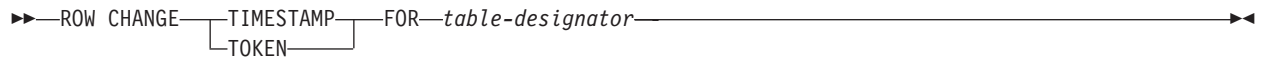
A nested table expression is used to first compute the result, including the ranking, before the rank can be used in the **WHERE** clause. A common table expression

| could also have been used.

ROW CHANGE expression

A ROW CHANGE expression returns a token or a timestamp that represents the last change to a row.

ROW CHANGE expression



TIMESTAMP

Specifies that a timestamp is returned that represents the last time when a row was changed. If the row has not been changed, the result is the time that the initial value was inserted.

TOKEN

Specifies that a token that is a BIGINT value is returned that represents a relative point in the modification sequence of a row. If the row has not been changed, the result is a token that represents when the initial value was inserted.

FOR *table-designator*

Identifies the table in which the expression is referenced. *table-designator* must uniquely identify a base table, a view, or a nested table expression of a subselect. If *table-designator* identifies a view or a nested table expression, the ROW CHANGE expression returns the TIMESTAMP or TOKEN of the base table of the view or the nested table expression. The view or nested table expression must contain only one base table in its outer subselect. *table-designator* must not identify a materialized view or a nested table expression that is materialized.

The result can be null. The **ROW CHANGE TIMESTAMP** and **ROW CHANGE TOKEN** expressions are not deterministic.

Example 1:

The following example returns all the rows that have been changed in the last day:

```
SELECT * FROM ORDERS
WHERE ROW CHANGE TIMESTAMP FOR ORDERS >
      CURRENT_TIMESTAMP - 24 HOURS;
```

Example 2:

The following example returns a timestamp value that corresponds to the most recent change to each row from the EMP table for those employees in department 20:

```
SELECT ROW CHANGE TIMESTAMP FOR EMP
FROM EMP WHERE DEPTNO = 20;
```

Example 3:

The following example returns a BIGINT value that corresponds to a relative point in the modification sequence of EMP with employee number '3500':

```
SELECT ROW CHANGE TOKEN FOR EMP
FROM EMP WHERE EMPNO = '3500';
```

Sequence reference

A sequence is referenced by using the NEXT VALUE and PREVIOUS VALUE expressions specifying the name of the sequence.

sequence-reference

►► *nextval-expression* ————— ►►
 └─ *prevval-expression* ─┘

nextval-expression

►► NEXT VALUE FOR *sequence-name* ————— ►►

prevval-expression

►► PREVIOUS VALUE FOR *sequence-name* ————— ►►

nextval-expression

A NEXT VALUE expression generates and returns the next value for a specified sequence. A new value is generated for a sequence when a NEXT VALUE expression specifies the name of the sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the sequence value is incremented only once for each row of the result, and all instances of **NEXT VALUE** return the same value for a row of the result. The **NEXT VALUE** expression is a not deterministic with external actions since it causes the sequence value to be incremented.

When the next value for the sequence is generated, if the maximum value for an ascending sequence or the minimum value for a descending sequence of the logical range of the sequence is exceeded and the **NO CYCLE** option is in effect, then an error occurs. To avoid this error, either alter the sequence attributes to extend the range of value or to enable cycles for the sequence or drop and recreate the sequence with a different data type that allows a larger range of values.

The data type and length attributes of the result of a NEXT VALUE expression are the same as for the specified sequence. The result cannot be null.

prevval-expression

A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current application process. This value can be repeatedly referenced by using PREVIOUS VALUE expressions to specify the name of the sequence. There can be multiple instances of PREVIOUS VALUE expressions specifying the same sequence name within a single statement and they all return the same value.

A PREVIOUS VALUE expression can be used only if a NEXT VALUE expression specifying the same sequence name has already been referenced in the current application process.

The data type and length attributes of the result of a PREVIOUS VALUE expression are the same as for the specified sequence. The result cannot be null.

sequence-name

Identifies the sequence that is to be referenced. The combination of name and the implicit or explicit schema name must identify an existing sequence at the current server. *sequence-name* must not be the name of an internal sequence object that is generated by DB2 for an identity column. The contents of the SQL PATH are not used to determine the implicit qualifier of a sequence name.

Authorization: If a sequence is referenced in a statement, the privileges that are held by the authorization ID of the statement must include at least one of the following:

- For the sequence identified in the statement:
 - The USAGE privilege on the sequence
 - Ownership of the sequence
- SYSADM or SYSCTRL authority

Generating values with NEXT VALUE: When a value is generated for a sequence, that value is consumed, and the next time that a value is requested, a new value will be generated. This is true even when the statement containing the NEXT VALUE expression fails or is rolled back.

Scope of NEXT VALUE and PREVIOUS VALUE: The value of PREVIOUS VALUE cannot be directly set and is a result of executing the NEXT VALUE expression for the sequence. The value of PREVIOUS VALUE persists until the next value is generated for the sequence in the current session, the sequence is dropped or altered, or the application session ends.

The value for the sequence cannot persist across a COMMIT or ROLLBACK for a local or remote application if, after the COMMIT or ROLLBACK, the DB2 application thread or server thread is assigned to another user or DB2 connection because of some form of thread reuse, re-signon, or connection pooling is in effect. For example, this can occur for CICS-DB2 applications and for client applications or middleware products that save the state of a session and then restore the state of a session for subsequent processing because they are not able to restore the NEXT or PREVIOUS VALUES for a sequence. In these situations, the availability of the value for a sequence should only be relied on until the end of the transaction. Examples of where this type of situation can occur include applications that do the following:

- issue an EXEC CICS SYNCPOINT command
- use XA protocols
- use connection pooling
- use the connection concentrator
- use Sysplex workload balancing
- connect to a z/OS server that uses DDF inactive threads

When there is a need to preserve the value that is associated with NEXT VALUE or PREVIOUS VALUE expressions across transaction boundaries for local or

distributed applications that are subject to thread reuse, re-signon, or connection pooling, take one of the following actions to prevent the local or server thread from re-signon, being reused by a different user, or from being pooled:

- Define at least one cursor as WITH HOLD and leave it as OPEN.
- Specify the bind option KEEP DYNAMIC(YES).
- For CICS users of local DB2 threads, set the DB2 system parameter PREVALKEEP = YES (the default value is NO). Note that the system parameter PREVALKEEP is deprecated and might be removed from a version of DB2 after DB2 Version 9, and only the previous two choices will be available.

Use as a unique key value: The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row (this generates the sequence value), and a PREVIOUS VALUE expression for the other rows (the instance of PREVIOUS VALUE refers to the sequence value most recently generated in the current session), as shown in the following example:

```
INSERT INTO ORDER (ORDERNO, CUSTNO)
VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
INSERT INTO LINE_ITEM (ORDERNO, PARTNO, QUANTITY)
VALUES (PREVIOUS VALUE FOR ORDER_SEQ, 987654, 1);
```

Allowed use of NEXT VALUE and PREVIOUS VALUE: The NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:

- Within the *select-clause* of a SELECT statement or SELECT statement that does not contain a **DISTINCT** keyword, a **GROUP BY** clause, an **ORDER BY** clause, or a set operator.
- Within a **VALUES** clause of an INSERT statement, including a multiple row INSERT statement with multiple **VALUES** clauses and the insert operation of a MERGE statement, which can include a NEXT VALUE expression for a particular sequence name for each **VALUES** clause.
- Within the *select-clause* of the fullselect of an INSERT statement.
- Within the **SET** clause of a searched or positioned UPDATE statement, including the update operation of the MERGE statement, though NEXT VALUE cannot be specified in the *select-clause* of the fullselect of an expression in the **SET** clause.

A PREVIOUS VALUE expression can be specified anywhere with a **SET** clause of an update operation (the UPDATE or MERGE statement), but a NEXT VALUE expression can be specified only in a **SET** clause if it is not within the *select-clause* of the fullselect of an expression. For instance, the following uses of sequence references are supported:

```
UPDATE T SET C1 = (SELECT PREVIOUS VALUE FOR S1 FROM T);
UPDATE T SET C1 = PREVIOUS VALUE FOR S1;
UPDATE T SET C1 = NEXT VALUE FOR S1;
```

The following uses of sequence references are not supported:

```
UPDATE T SET C1 = (SELECT NEXT VALUE FOR S1 FROM T);
SET :C2 = (SELECT NEXT VALUE FOR S1 FROM T);
```

- In a **SET host-variable** or *assignment-statement*, except within the *select-clause* of the fullselect of an expression.

The following uses of sequence references are supported:

```
SET ORDERNUM = NEXT VALUE FOR INVOICE;
SET ORDERNUM = PREVIOUS VALUE FOR INVOICE;
```

The following uses of sequence references are not supported:


```
SET X = (SELECT NEXT VALUE FOR S1 FROM T);
SET X = (SELECT PREVIOUS VALUE FOR S1 FROM T);
```

- In a **VALUES** or **VALUES INTO** statement though not within the *select-clause* of the fullselect of an expression.
- Within the *SQL-routine-body* of a CREATE or ALTER PROCEDURE statement for a SQL procedure.
- Within the *RETURN-statement* of a CREATE FUNCTION statement for an SQL function.
- Within the *SQL-trigger-body* of a CREATE TRIGGER statement (PREVIOUS VALUE is not allowed).

PREVIOUS VALUE is defined to have a linear scope within an application session. Therefore, in a nested application on entry to a nested function, procedure, or trigger, the nested application inherits the most recently generated value for a sequence. That is, an invocation of PREVIOUS VALUE in a nested application reflects sequence activity done in the invoking environment prior to entering the nested application. In addition, on return from a function, procedure, or trigger, the invoking application is affected by any sequence activity in the lower level applications. That is, an invocation of PREVIOUS VALUE in the invoking application after returning from the nested application reflects any sequence activity that occurred in the lower level applications.

Restrictions on the use of NEXT VALUE and PREVIOUS VALUE: Some of the places where the NEXT VALUE and PREVIOUS VALUE expressions cannot be specified include the following:

- Join condition of a full outer join
- DEFAULT value for a column in a CREATE TABLE or ALTER TABLE statement
- Materialized query table definition in a CREATE TABLE or ALTER TABLE statement
- Condition of a CHECK constraint
- Input value specification for LOAD
- CREATE VIEW statement
- The SELECT list of a subselect that contains a NOT ATOMIC data change statement
- ORDER BY clause when used in an OLAP specification

In addition, the NEXT VALUE expression cannot be specified in the following places:

- CASE expression
- Parameter list of an aggregate function
- Subquery in a context other than those explicitly allowed
- SELECT statement for which the outer SELECT contains a DISTINCT operator or a GROUP BY clause
- SELECT statement for which the outer SELECT is combined with another SELECT statement using a set operator
- Join condition of a join
- Nested table expression
- Parameter list of a table function
- *select-clause* of the fullselect of an expression in the SET clause of an UPDATE, a DELETE, or a MERGE statement.

- WHERE clause of the outer-most SELECT statement or a DELETE, , an UPDATE, or a MERGE statement
- ORDER BY clause of the outer-most SELECT statement
- IF, WHILE, DO UNTIL, or CASE statements in an SQL routine

Using sequence expressions with a cursor: Normally, a SELECT NEXT VALUE FOR ORDER_SEQ FROM T1 would produce a result table containing as many generated values from the sequence ORDER_SEQ as the number of rows retrieved from T1. A reference to a NEXT VALUE expression in the SELECT statement of a cursor refers to a value that is generated for a row of the result table. A sequence value is generated for a NEXT VALUE expression each time a row is retrieved.

If blocking is done at a client in a DRDA environment, sequence values might get generated at the DB2 server before the processing of an application's FETCH statement. If the client application does not explicitly fetch all the rows that have been retrieved from the database, the application will never see all those values of the sequence that are generated but not fetched (as many values as the rows that are not fetched). These generated but not fetched values might constitute a gap in the sequence. If it is important to prevent such a gap, do the following:

- Use NEXT VALUE only in places where it would function without being controlled by a cursor and where block-fetching by the client will have no effect on it.
- If you must use NEXT VALUE in the SELECT statement of a cursor-definition, weigh the importance of preventing the gap against performance and other implications of taking the following actions:
 - Use FETCH FOR 1 ROW ONLY clause with the SELECT statement.
 - Try preventing block-fetch by other means documented in *DB2 Application Programming and SQL Guide*,

Using the PREVIOUS VALUE expression with a cursor: A reference to the PREVIOUS VALUE expression in a SELECT statement of a cursor is evaluated at OPEN time. In other words, a reference to the PREVIOUS VALUE expression in the SELECT statement of a cursor refers to the last value generated by this application process for the specified sequence prior to the opening of the cursor and, once evaluated at OPEN time, the value returned by PREVIOUS VALUE within the select statement of the cursor will not change from FETCH to FETCH, even if NEXT VALUE is invoked with the select statement of the cursor. After the cursor is closed, the value of PREVIOUS VALUE will be the last NEXT VALUE that is generated by the application process.

IF PREVIOUS VALUE is used in the SELECT statement of a cursor while the cursor is open, the PREVIOUS VALUE value would be the last NEXT VALUE for the sequence generated before the cursor was opened. After the cursor is closed, the PREVIOUS VALUE value would be the last NEXT VALUE generated by the application process.

Syntax alternatives and synonyms: For compatibility, the keywords NEXTVAL and PREVVAL can be used as synonyms for NEXT VALUE and PREVIOUS VALUE respectively.

Example: Assume that there is a table called ORDER, and that a sequence called ORDER_SEQ is created as follows:

```

CREATE SEQUENCE ORDER_SEQ START WITH 1
                        INCREMENT BY 1
                        NO MAXVALUE
                        NO CYCLE
                        CACHE 24

```

The following examples illustrate how to generate an ORDER_SEQ sequence number with a NEXT VALUE expression:

```

INSERT INTO ORDER (ORDERNO, CUSTNO)
VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
UPDATE ORDER SET ORDERNO = NEXT VALUE FOR ORDER_SEQ
WHERE CUSTNO = 123456;
VALUES NEXT VALUE FOR ORDER_SEQ INTO :HV_SEQ;

```

Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given row or group.

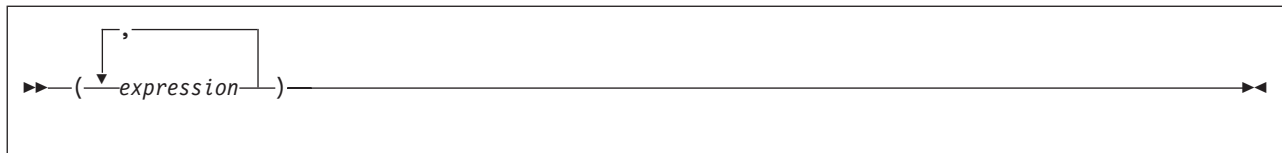
The types of predicates are:



The following rules apply to predicates of any type:

- Predicates are evaluated after the expressions that are operands of the predicate.
- All values that are specified in the same predicate must be compatible.
- Except for the EXISTS predicate, a subquery in a predicate must specify a single column unless the operand on the other side of the comparison operator is a fullselect.
- The value of a host variable can be null (that is, the variable can have a negative indicator variable).
- The CCSID conversion of operands of predicates that involve two or more operands is done according to “Conversion rules for operations that combine strings” on page 122.
- Use of an XML value is limited to the NULL or XMLEXISTS predicates.

Row-value-expression: The operand of several predicates (basic, quantified, DISTINCT, and IN) can be a *row-value-expression*:

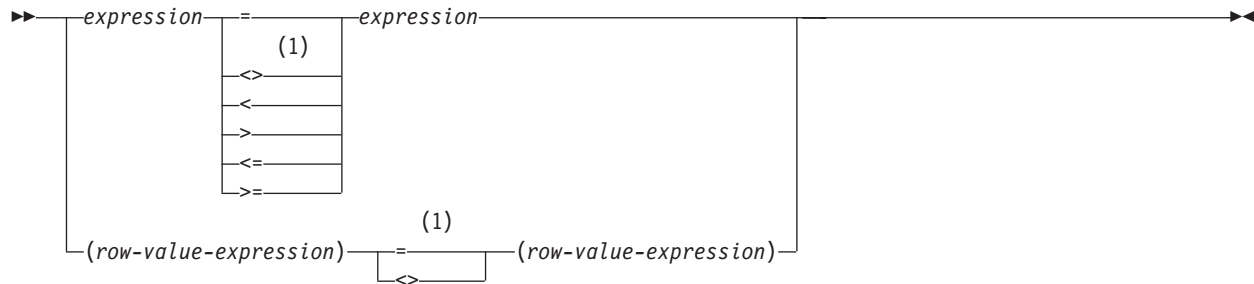


A *row-value-expression* returns a single row that consists of one or more column values. The values can be specified as a list of expressions. The number of columns that are returned by the *row-value-expression* is equal to the number of expressions that are specified in the list.

Other predicate examples: In addition to the examples of predicates in the following topics, see information on distinct type comparisons in “Assignment and comparison” on page 102, which contains several examples of predicates that use distinct types. “Distinct type comparisons” on page 117, which contains several examples of predicates that use distinct types.

Basic predicate

A *basic predicate* compares two values or compares a set of values with another set of values.



Notes:

- 1 Other comparison operators are also supported.¹⁶

When *expression* is a fullselect, the fullselect must return a single result column with a single value, whether null or not null. If the value of either operand is null or the result of the fullselect is empty, the result of the predicate is unknown. Otherwise, the result is either true or false.

When a *row-value-expression* is specified on the left side of the operator (= or <>), another *row-value-expression*, with an identical number of value expressions, must be specified on the right side. The data types of the corresponding expressions or columns of the *row-value-expressions* must be compatible. The value of each expression on the left side is compared with the value of its corresponding expression on the right side. The result of the predicate depends on the operator, as in the following two cases:

- If the operator is =, the result of the predicate is:
 - True if all pairs of corresponding value expressions evaluate to true.
 - False if any one pair of corresponding value expressions evaluates to false.
 - Otherwise, unknown (that is, if at least one comparison of corresponding value expressions is unknown because of a null value and no pair of corresponding value expressions evaluates to false).
- If the operator is <>, the result of the predicate (x1,x2,...,xn) <> (y1,y2,...,yn) is:
 - True, if and only if xi=yi evaluates to false for some value of i. (that is, there is at least one pair of non-null values, xi and yi, that are not equal to each other)

16. The following forms of the comparison operators are also supported in basic and quantified predicates in coded pages where the exclamation point is X'5A': !=, !<, and !> . In addition, in code pages 437, 819, and 850, the forms !=, !=, and != are supported. All these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators and are not recommended for use when writing new SQL statements.

A not sign (¬) or the character that must be used in its place in certain countries, can cause parsing errors in statements passed from one DBMS to another. The problem occurs if the statement undergoes character conversion with certain combinations of source and target CCSIDs. To avoid this problem, substitute an equivalent operator for any operator that includes a not sign. For example, substitute '<>' for '!=', '<=' for '>', and '>=' for '<'.

- | – False, if and only if $x_i=y_i$ evaluates to true for every value of i . (that is,
- | $(x_1,x_2,...,x_n)=(y_1,y_2,...,y_n)$ is true)
- | – Otherwise, unknown (that is, x_i or y_i is a null value for some value of i , and
- | there is no value of j such that $x_j=y_j$ evaluates to false).

Table 42. For values x and y

Predicate	Is true if and only if ...
$x = y$	x is equal to y
$x \neq y$	x is not equal to y
$x < y$	x is less than y
$x > y$	x is greater than y
$x \leq y$	x is less than or equal to y
$x \geq y$	x is greater than or equal to y

Examples for values x and y :

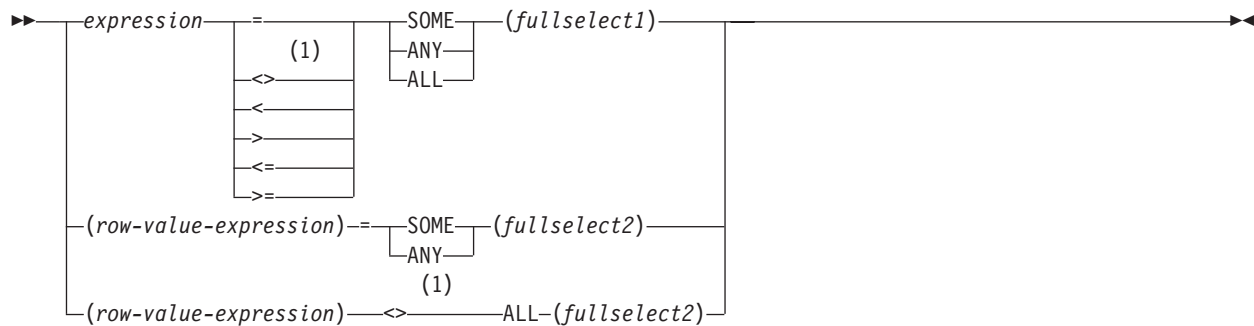
```
EMPNO = '528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY >= (SELECT AVG(SALARY) FROM DSN8910.EMP)
```

Example: List the name, first name, and salary of the employee who is responsible for the 'SECRET' project. This employee might appear in either the PROJA1 or PROJA2 tables. A UNION is used in case the employee appears in both tables to eliminate duplicate RESPEMP values.

```
SELECT LASTNAME, FIRSTNAME, SALARY
  FROM DSN8910.EMP X
 WHERE EMPNO = (
SELECT RESPEMP
  FROM PROJA1 Y
 WHERE MAJPROJ = 'SECRET'
UNION
SELECT RESPEMP
  FROM PROJA2 Z
 WHERE MAJPROJ = 'SECRET');
```

Quantified predicate

A *quantified predicate* compares a value or values with a collection of values.



Notes:

- 1 Other comparison operators are also supported.¹⁶

When *expression* is specified, *fullselect1* must return a single result column, and can return any number of values, whether null or not null. The result depends on the operator that is specified:

- When the operator is **ALL**, the result of the predicate is:
 - True – if the result of the fullselect is empty or if the specified relationship is true for every value returned by the fullselect.
 - False – if the specified relationship is false for at least one value returned by the fullselect.
 - Unknown – if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of a null value.
- When the operator is **SOME** or **ANY**, the result of the predicate is:
 - True – if the specified relationship is true for at least one value returned by the fullselect.
 - False – if the result of the fullselect is empty or if the specified relationship is false for every value returned by the fullselect.
 - Unknown – if the specified relationship is not true for any of the values returned by the fullselect and at least one comparison is unknown because of a null value.

When *row-value-expression* is specified, the number of result columns returned by *fullselect2* must be the same as the number of value expressions specified by *row-value-expression*, and *fullselect2* can return any number of rows of values. The data types of the corresponding expressions of the row value expressions must be compatible. The value of each expression from *row-value-expression* is compared with the value of the corresponding result column from *fullselect2*. The value of the predicate depends on the operator that is specified:

- When the operator is **ALL**, the result of the predicate is:
 - True – if the result of *fullselect2* is empty or if the specified relationship is true for every row returned by *fullselect2*.
 - False – if the specified relationship is false for at least one row returned by *fullselect2*.

- Unknown – if the specified relationship is not false for any row returned by *fullselect2* and at least one comparison is unknown because of a null value.
- When the operator is SOME or ANY, the result of the predicate is:
 - True – if the specified relationship is true for at least one row returned by *fullselect2*
 - False – if the result of the fullselect is empty or if the specified relationship is false for every row returned by *fullselect2*.
 - Unknown – if the specified relationship is not true for any of the rows returned by *fullselect2* and at least one comparison is unknown because of a null value.

Quantified predicates are equivalent to IN predicates. See Table 47 on page 235 for some examples of equivalent quantified and IN predicates.

Examples: Use the following tables when referring to the following examples. In all examples, “row *n* of TBLA” refers to the row in TBLA for which COLA has the value *n*.

Table 43. TBLA

COLA
1
2
3
4

Table 44. TBLB

COLB	COLC
2	2
3	--

Table 45. TBLC

COLB	COLC
2	2

Example 1: In the following predicate, the fullselect returns the values 2 and 3. The predicate is false for rows 1, 2, and 3 of TBLA, and is true for row 4.

```
COLA > ALL(SELECT COLB FROM TBLB
           UNION
           SELECT COLB FROM TBLC)
```

Example 2: In the following predicate, the fullselect returns the values 2 and 3. The predicate is false for rows 1 and 2 of TBLA, and is true for rows 3 and 4.

```
COLA > ANY(SELECT COLB FROM TBLB
           UNION
           SELECT COLB FROM TBLC)
```

Example 3: In the following predicate, the fullselect returns the values 2 and null. The predicate is false for rows 1 and 2 of TBLA, and is unknown for rows 3 and 4. The result is an empty table.

```
COLA > ALL(SELECT COLC FROM TBLB  
           UNION  
           SELECT COLC FROM TBLC)
```

Example 4: In the following predicate, the fullselect returns the values 2 and null. The predicate is unknown for rows 1 and 2 of TBLA, and is true for rows 3 and 4.

```
COLA > SOME(SELECT COLC FROM TBLB  
           UNION  
           SELECT COLC FROM TBLC)
```

Example 5: In the following predicate, the fullselect returns an empty result column. Hence, the predicate is true for all rows of TBLA.

```
COLA < ALL(SELECT COLB FROM TBLB WHERE COLB>3  
           UNION  
           SELECT COLB FROM TBLC WHERE COLB>3)
```

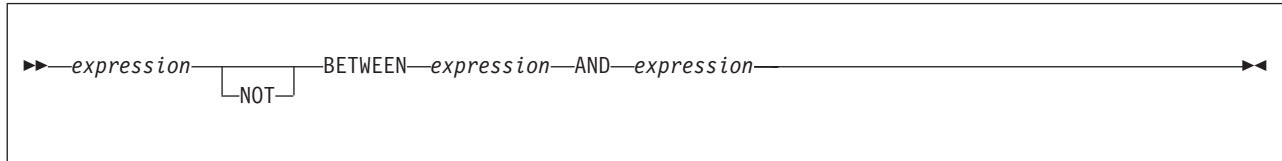
Example 6: In the following predicate, the fullselect returns an empty result column. Hence, the predicate is false for all rows of TBLA.

```
COLA < ANY(SELECT COLB FROM TBLB WHERE COLB>3  
           UNION  
           SELECT COLB FROM TBLC WHERE COLB>3)
```

If COLA were null in one or more rows of TBLA, the predicate would still be false for all rows of TBLA.

BETWEEN predicate

The BETWEEN predicate determines whether a given value lies between two other given values that are specified in ascending order.



Each of the predicate's two forms has an equivalent search condition, as shown in the following table:

Table 46. BETWEEN predicate and equivalent search conditions

BETWEEN predicate	Equivalent search condition
<i>value1</i> BETWEEN <i>value2</i> AND <i>value3</i>	<i>value1</i> >= <i>value2</i> AND <i>value1</i> <= <i>value3</i> ¹
<i>value1</i> NOT BETWEEN <i>value2</i> AND <i>value3</i>	<i>value1</i> < <i>value2</i> OR <i>value1</i> > <i>value3</i> ¹

or, equivalently:

NOT(*value1* BETWEEN *value2* AND *value3*)

Note: 1. Might not be equivalent if *value1*, *value2*, or *value3* are columns or derived values based on columns that are not the same CCSID set because the clause is evaluated in Unicode.

Search conditions are discussed in “Search conditions” on page 247.

If the operands include a mixture of datetime values and valid string representations of datetime values, all values are converted to the data type of the datetime operand.

Example: Consider the following predicate:

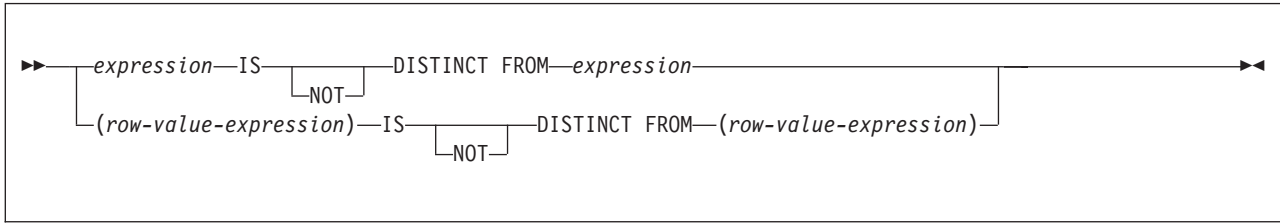
A BETWEEN *B* AND *C*

The following table shows the value of the predicate for various values of *A*, *B*, and *C*.

Value of <i>A</i>	Value of <i>B</i>	Value of <i>C</i>	Value of predicate
1,2, or 3	1	3	true
0 or 4	1	3	false
0	1	null	false
4	null	3	false
null	any value	any value	unknown
2	1	null	unknown
3	null	4	unknown

DISTINCT predicate

A distinct predicate compares a value with another value or a set of values with another set of values.



The number of elements that are returned by the *row-value-expression* that specified after the distinct operator must match the number of elements that are returned by the *row-value-expression* that is specified prior to the distinct operator. The data types of the corresponding columns or expressions of the *row-value-expressions* must be compatible. When the predicate is evaluated, the value of each expression on the left side is compared with the value of its corresponding expression on the right side. The result of the predicate depends on the form of the predicate.

When the predicate is **IS DISTINCT**, the result of the predicate is true if at least one comparison of a pair of corresponding value expressions evaluates to false. Otherwise, the result of the predicate is false. The result cannot be unknown.

When the predicate **IS NOT DISTINCT FROM**, the result of the predicate is true if all pairs of corresponding value expressions evaluate to true (null values are considered equal to null values). Otherwise, the predicate is false. The result cannot be unknown.

The DISTINCT predicate cannot be used in the following contexts:

- The **ON** *join-condition* of a full outer join
- A check constraint
- A quantified predicate

The following DISTINCT predicate is logically equivalent to the following search conditions:

```
value 1 IS NOT DISTINCT FROM value2
-- is logically equivalent to
(value1 IS NOT NULL AND value2 IS NOT NULL AND value1 = value 2)
-- or is logically equivalent to
(value1 IS NULL AND value2 IS NULL)
```

The following DISTINCT predicate is logically equivalent to the following search condition:

```
value 1 IS DISTINCT FROM value2
-- is logically equivalent to
NOT (value1 IS NOT DISTINCT FROM value2)
```

Example 1: Assume that T1 is a single-column table with three rows. Column C1 has the following values: 1, 2, and null. Consider the following query:

```
SELECT * FROM T1
      WHERE C1 IS DISTINCT FROM :HV;
```

The following table shows the value of the predicate for various values of C1 and the host variable.

Value of C1	Value of <i>HV</i>	Result of predicate
1	2	True
2	2	False
null	2	True
1	null	True
2	null	True
null	null	False

Example 2: Assume the same table as in the first example, but now consider the negative form of the predicate in the query:

```
SELECT * FROM T1
WHERE C1 IS NOT DISTINCT FROM :HV;
```

The following table shows the value of the predicate for various values of C1 and the host variable.

Value of C1	Value of <i>HV</i>	Result of predicate
1	2	False
2	2	True
null	2	False
1	null	False
2	null	False
null	null	True

EXISTS predicate

The EXISTS predicate tests for the existence of certain rows. The fullselect can specify any number of columns, and can result in true or false.

►►—EXISTS—(fullselect)—►►

The result of the EXISTS predicate:

- Is true only if the number of rows that is specified by the fullselect is not zero.
- Is false only if the number of rows specified by the fullselect is zero.
- Cannot be unknown.

The SELECT clause in the fullselect can specify any number of columns because the values returned by the fullselect are ignored. For convenience, use:

```
SELECT *
```

Unlike the NULL, LIKE, and IN predicates, the EXISTS predicate has no form that contains the word NOT. To negate an EXISTS predicate, precede it with the logical operator NOT, as follows:

```
NOT EXISTS (fullselect)
```

The result is then false if the EXISTS predicate is true, and true if the predicate is false. Here, NOT is a logical operator and not a part of the predicate. Logical operators are discussed in “Search conditions” on page 247.

Example 1: The following query lists the employee number of everyone represented in DSN8910.EMP who works in a department where at least one employee has a salary less than 20000. Like many EXISTS predicates, the one in this query involves a correlated variable.

```
SELECT EMPNO
FROM DSN8910.EMP X
WHERE EXISTS (SELECT * FROM DSN8910.EMP
              WHERE X.WORKDEPT=WORKDEPT AND SALARY<20000);
```

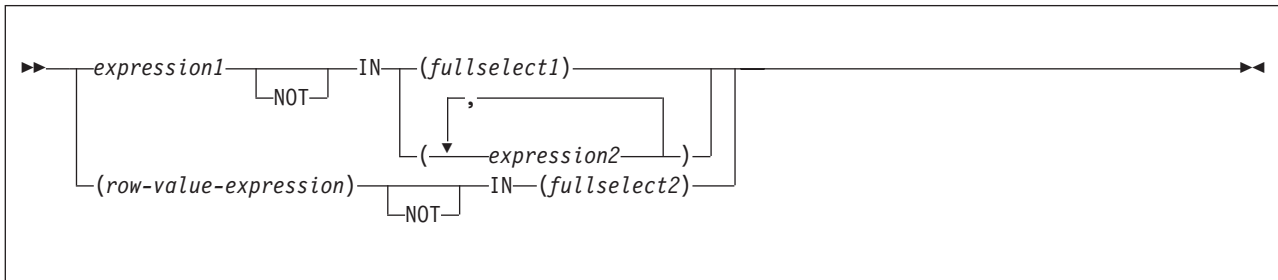
Example 2: List the subscribers (SNO) in the state of California who made at least one call during the first quarter of 2009. Order the results according to SNO. Each MONTHnn table has columns for SNO, CHARGES, and DATE. The CUST table has columns for SNO and STATE.

```
SELECT C.SNO
FROM CUST C
WHERE C.STATE = 'CA'
AND EXISTS (
  SELECT *
  FROM MONTH1
  WHERE DATE BETWEEN '01/01/2009 AND '01/31/2009'
  AND C.SNO = SNO
UNION ALL
  SELECT *
  FROM MONTH2
  WHERE DATE BETWEEN '02/01/2009 AND '02/28/2009'
  AND C.SNO = SNO
UNION ALL
  SELECT *
```

```
FROM MONTH3  
WHERE DATE BETWEEN '03/01/2009 AND '03/31/2009'  
AND C.SNO = SNO  
)  
ORDER BY C.SNO;
```


IN predicate

The IN predicate compares a value or values with a set of values.



When *expression1* is specified, the IN predicate compares a value with a set of values. When *fullselect1* is specified, the fullselect must return a single result column, and can return any number of values, whether null or not null. The data type of *expression1* and the data type of the result column of *fullselect1* or *expression2* must be compatible. If *expression* is a single host variable, the host variable can identify a structure. Any host variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables.

When a *row-value-expression* is specified, the IN predicate compares values with a collection of values. The result table of the *fullselect2* must have the same number of columns as the *row-value-expression*. The data type of each expression in *row-value-expression* and the data type of its the corresponding result column of *fullselect2* must be compatible. The value of each expression in *row-value-expression* is compared with the value of its corresponding result column of *fullselect2*. The value of the predicate depends on the operator that is specified:

- When the operator is IN, the result of the predicate is:
 - True if at least one row returned from *fullselect2* is equal to the *row-value-expression*.
 - False if the result of *fullselect2* is empty or if no row returned from *fullselect2* is equal to the *row-value-expression*.
 - Otherwise, unknown (that is, if the comparison of *row-value-expression* to the row returned from *fullselect2* evaluates to unknown because of a null value for at least one row returned from *fullselect2* and no row returned from *fullselect2* is equal to the *row-value-expression*).
- When the operator is NOT IN, the result of the predicate is:
 - True if the result of *fullselect2* is empty or if the *row-value-expression* is not equal to any of the rows returned by *fullselect2*.
 - False if the *row-value-expression* is equal to at least one row returned by *fullselect2*.
 - Otherwise, unknown (that is, if the comparison of *row-value-expression* to the row returned from *fullselect2* evaluates to unknown because of a null value for at least one row returned from *fullselect2* and the comparison of *row-value-expression* to the row returned from *fullselect2* is not true for any row returned by the *fullselect2*).

The IN predicate is equivalent to the quantified predicate as follows:

Table 47. IN predicate and equivalent quantified predicates

IN predicate	Equivalent quantified predicate
<i>expression1</i> IN (<i>expression2</i>)	<i>expression1</i> = <i>expression2</i>
<i>expression</i> IN (<i>fullselect1</i>)	<i>expression</i> = ANY (<i>fullselect1</i>)
<i>expression</i> NOT IN (<i>fullselect1</i>)	<i>expression</i> <> ALL (<i>fullselect1</i>)
<i>expression1</i> IN (<i>expressiona</i> , <i>expressionb</i> , ...)	<i>expression1</i> IN (SELECT * FROM R) When T is a table with a single row and R is a result table formed by the following fullselect: SELECT value1 FROM T UNION SELECT value2 FROM T UNION . . . UNION SELECT valuen FROM T
<i>row-value-expression</i> IN (<i>fullselect2</i>)	<i>row-value-expression</i> = SOME (<i>fullselect2</i>)
<i>row-value-expression</i> IN (<i>fullselect2</i>)	<i>row-value-expression</i> = ANY (<i>fullselect2</i>)
<i>row-value-expression</i> NOT IN (<i>fullselect2</i>)	<i>row-value-expression</i> <> ALL (<i>fullselect2</i>)

If the operands of the IN predicate are strings with different CCSIDs, the rules used to determine which operands are converted are those for operations that combine strings. See “Character and graphic string comparisons” on page 116.

Example 1: The following predicate is true for any row whose employee is in department D11, B01, or C01.

```
WORKDEPT IN ('D11', 'B01', 'C01')
```

Example 2: The following predicate is true for any row whose employee works in department E11.

```
EMPNO IN (SELECT EMPNO FROM DSN8910.EMP
WHERE WORKDEPT = 'E11')
```

Example 3: The following predicate is true if the date that a project is estimated to start (PRENDATE) is within the next two years.

```
YEAR(PRENDATE) IN (YEAR(CURRENT DATE),
YEAR(CURRENT DATE + 1 YEAR),
YEAR(CURRENT DATE + 2 YEARS))
```

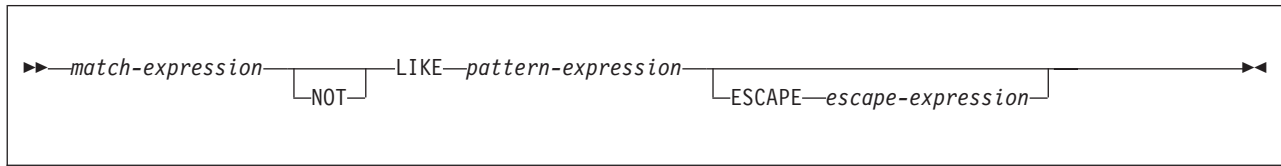
Example 4: The following example obtains the phone number of an employee in DSN8910.EMP where the employee number (EMPNO) is a value specified within the COBOL structure defined below.

```
77 PHNUM PIC X(6).
01 EMPNO-STRUCTURE.
05 CHAR-ELEMENT-1 PIC X(6) VALUE '000140'.
05 CHAR-ELEMENT-2 PIC X(6) VALUE '000340'.
05 CHAR-ELEMENT-3 PIC X(6) VALUE '000220'.
.
.
.
EXEC SQL DECLARE PHCURS CURSOR FOR
SELECT PHONENO FROM DSN8910.EMP
```

```
        WHERE EMPNO IN
        (:EMPNO-STRUCTURE.CHAR-ELEMENT-1,
         :EMPNO-STRUCTURE.CHAR-ELEMENT-2,
         :EMPNO-STRUCTURE.CHAR-ELEMENT-3)
END-EXEC.
EXEC SQL OPEN PHCURS
END-EXEC.
EXEC SQL FETCH PHCURS INTO :PHNUM
END-EXEC.
```

LIKE predicate

The LIKE predicate searches for strings that have a certain pattern.



The *match-expression* is the string to be tested for conformity to the pattern specified in *pattern-expression*. Underscore and percent sign characters in the pattern have a special meaning instead of their literal meanings unless *escape-expression* is specified, as discussed under the description of *pattern-expression*.

The following rules summarize how a predicate in the form of *m* LIKE *p* is evaluated:

- If *m* or *p* is null, the result of the predicate is unknown.
- If *m* and *p* are both empty, the result of the predicate is true.
- If *m* is empty and *p* is not, the result of the predicate is unknown unless *p* consists of one or more percent signs.
- If *m* is not empty and *p* is empty, the result of the predicate is false.
- Otherwise, if *m* matches the pattern in *p*, the result of the predicate is true. The description of *pattern-expression* provides a detailed explanation on how the pattern is matched to evaluate the predicate to true or false. See the “rigorous description of the pattern” for this information.

The values for *match-expression*, *pattern-expression*, and *escape-expression* must all be character or graphic strings or a mixture of both or they must all be binary strings (BLOBs). None of the expressions can yield a distinct type; however, an expression can be a function that casts a distinct type to its source type.

There are slight differences in what expressions are supported for each argument. The description of each argument lists the supported expressions:

match-expression

An expression that specifies the string to be tested for conformity to a certain pattern of characters.

LIKE *pattern-expression*

An expression that specifies the pattern of characters to be matched.

The expression can be specified by any one of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above

The expression must also meet these restrictions:

- The maximum length of *pattern-expression* must not be larger than 4000 bytes.

- If a host variable is used in *pattern-expression*, the host variable must be defined in accordance with the rules for declaring string host variables and must not be a structure.
- If *escape-expression* is specified, *pattern-expression* must not contain the escape character identified by *escape-expression* except when immediately followed by the escape character, '%', or '_'. For example, if '+' is the escape character, any occurrences of '+' other than '++', '+_', or '+ in the pattern is an error.

When the pattern specified in a LIKE predicate is a parameter marker and a fixed-length character host variable is used to replace the parameter marker, specify a value for the host variable that is the correct length. If you do not specify the correct length, the select does not return the intended results. For example, if the host variable is defined as CHAR(10) and the value WYSE% is assigned to that host variable, the host variable is padded with blanks on assignment. The pattern used is 'WYSE ', which requests DB2 to search for all values that start with WYSE and end with five blank spaces. If you intended to search for only the values that start with 'WYSE ', you should assign the value 'WYSE%%%%%%%%' to the host variable.

If the pattern is specified in a fixed-length string variable, any trailing blanks are interpreted as part of the pattern. Therefore, it is better to use a varying-length string variable with an actual length that is the same as the length of the pattern. If the host language does not allow varying-length string variables, place the pattern in a fixed-length string variable whose length is the length of the pattern.

For more on the use of host variables with specific programming languages, see *DB2 Application Programming and SQL Guide*.

The pattern is used to specify the conformance criteria for values in the *match-expression* where:

- The underscore character (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents a single occurrence of itself.

If the *pattern-expression* needs to include either the underscore or the percent character, the *escape-expression* is used to specify a character to precede either the underscore or percent character in the pattern. For character strings, the terms *character*, *percent sign*, and *underscore* refer to SBCS characters. For graphic strings, the terms refer to double-byte or UTF-16 characters.

A rigorous description of the pattern: This more rigorous description of the pattern ignores the use of the *escape-expression*.

Let *m* denote the value of *match-expression* and let *p* denote the value of *pattern-expression*. The string *p* is interpreted as a sequence of the minimum number of substring specifiers so each character of *p* is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any non-empty sequence of characters other than an underscore or a percent sign.

The result of the predicate is unknown if *m* or *p* is the null value. Otherwise, the result is either true or false. The result is true if *m* and *p* are both empty strings or there exists a partitioning of *m* into substrings such that:

- A substring of *m* is a sequence of zero or more contiguous characters and each character of *m* is part of exactly one substring.
- If the *n*th substring specifier is an underscore, the *n*th substring of *m* is any single character.
- If the *n*th substring specifier is a percent sign, the *n*th substring of *m* is any sequence of zero or more characters.
- If the *n*th substring specifier is neither an underscore nor a percent sign, the *n*th substring of *m* is equal to that substring specifier and has the same length as that substring specifier.
- The number of substrings of *m* is the same as the number of substring specifiers.

It follows that if *p* is an empty string and *m* is not an empty string, the result is false. Similarly, if *m* is an empty string and *p* is not an empty string consisting of a value other than percentage signs, the result is false.

The predicate *m* NOT LIKE *p* is equivalent to the search condition NOT (*m* LIKE *p*).

Mixed data patterns: If *match-expression* represents mixed data, the pattern is assumed to be mixed data. For ASCII and EBCDIC, the special characters in the pattern are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one MBCS character.
- A percent sign (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

For EBCDIC, redundant shift bytes in *match-expression* or *pattern-expression* are ignored.

For Unicode, the special characters in the pattern are interpreted as follows:

- An SBCS or DBCS underscore refers to one character (either SBCS or MBCS).
- A percent sign (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

When the LIKE predicate is used with Unicode data, the Unicode percent sign and underscore use the code points indicated in the following table:

Character	UTF-8	UTF-16
Half-width %	X'25'	X'0025'
Full-width %	X'EFBC85'	X'FF05'
Half-width_	X'5F'	X'005F'
Full-width_	X'EFBCBF'	X'FF3F'

The full-width or half-width % matches zero or more characters. The full-width or half width_ character matches exactly one character. (For ASCII or EBCDIC

data, a full-width `_` character matches one DBCS character.)

ESCAPE *escape-expression*

An expression that specifies the escape character to be used to modify the special meaning of the underscore (`_`) and percent (`%`) characters in *pattern-expression*. Specifying an expression, which is optional, allows the `LIKE` predicate to explicitly test that the value contains a `"%"` or `"_"` in the desired character positions. The escape character consists of a single SBCS (1 byte) or DBCS (2 bytes) character. An escape clause is allowed for Unicode mixed (UTF-8) data, but is restricted for ASCII and EBCDIC mixed data.

The expression can be specified by any one of:

- A constant
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above

The following rules also apply to the use of the **ESCAPE** clause and *escape-expression*:

- The result of *escape-expression* must be one SBCS or DBCS character or a binary string that contains exactly 1 byte.
- The **ESCAPE** clause cannot be used if *match-expression* is mixed data.
- If *escape-expression* is specified by a host variable, the host variable must be defined in accordance with the rules for declaring fixed-length string host variables.¹⁷ If the host variable has a negative indicator variable, the result of the predicate is unknown.
- The pattern must not contain the escape character except when followed by the escape character, `"%"` or `"_"`. For example, if `"+"` is the escape character, any occurrences of `"+"` other than `"++"`, `"_+"`, or `"%+"` in the pattern is an error.

The following table shows the effect of successive occurrences of the escape character, which in this case is the plus sign (`+`).

Table 48. Effect of successive occurrences of the escape character

When the pattern string is...	The actual pattern is...
<code>+%</code>	A percent sign
<code>++%</code>	A plus sign followed by zero or more arbitrary characters
<code>+++%</code>	A plus sign followed by a percent sign

Examples

Example 1: The following predicate is true when the string to be tested in `NAME` has the value `SMITH`, `NESMITH`, `SMITHSON`, or `NESMITHY`. It is not true when the string has the value `SMYTHE`:

```
NAME LIKE '%SMITH%'
```

Example 2: In the predicate below, a host variable named `PATTERN` holds the string for the pattern:

```
NAME LIKE :PATTERN ESCAPE '+'
```

17. If it is NUL-terminated, a C character string variable of length 2 can be specified.

Assume that the string in PATTERN has the value:

AB+_C_%

Observe that in this string, the plus sign preceding the first underscore is an escape character. The predicate is true when the string being tested in NAME has the value AB_CD or AB_CDE. It is false when this string has the value AB, AB_, or AB_C.

Example 3: The following two predicates are equivalent; three of the four percent signs in the first predicate are redundant.

```
NAME LIKE 'AB%%%%CD'
NAME LIKE 'AB%CD'
```

Example 4: Assume that a distinct type named ZIP_TYPE with a source data type of CHAR(5) exists and an ADDRZIP column with data type ZIP_TYPE exists in some table TABLEY. The following statement selects the row if the zip code (ADDRZIP) begins with '9555'.

```
SELECT * FROM TABLEY
WHERE CHAR(ADDRZIP) LIKE '9555'
```

Example 5: The RESUME column in sample table DSN8910.EMP_PHOTO_RESUME is defined as a CLOB. The following statement selects the RESUME column when the string JONES appears anywhere in the column.

```
SELECT RESUME FROM DSN8910.EMP_PHOTO_RESUME
WHERE RESUME LIKE ' % JONES %'
```

Example 6: In the following table, assume COL1 is a column that contains mixed EBCDIC data. The table shows the results when the predicate in the first column is evaluated using the COL1 value in the second column:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa_ AB% C_ '	'aaa_ ABDZC_ '	True
WHERE COL1 LIKE 'aaa_ AB_ %_ C_ '	'aaa_ AB_ dzx_ C_ '	True
WHERE COL1 LIKE 'a%_ C_ '	'a_ C_ '	True
	'ax_ C_ '	True
	'ab_ DE_ fg_ C_ '	True
WHERE COL1 LIKE 'a_ _ C_ '	'a%_ C_ '	True
	'a_ XC_ '	False
WHERE COL1 LIKE 'a_ _ C_ '	'a_ XC_ '	True
	'ax_ C_ '	False
WHERE COL1 LIKE ' ' _ '	Empty string	True
WHERE COL1 LIKE 'ab_ C_ _ '	'ab_ C_ d'	True
	'ab_ _ C_ d'	True

Example 7: In the following table, assume COL1 is a column that contains mixed ASCII data. The table shows the results when the predicate in the first column is evaluated using the COL1 value in the second column:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa AB %C'	'aaa ABDZC '	True
WHERE COL1 LIKE 'aaa AB %C'	'aaa AB dzx C'	True

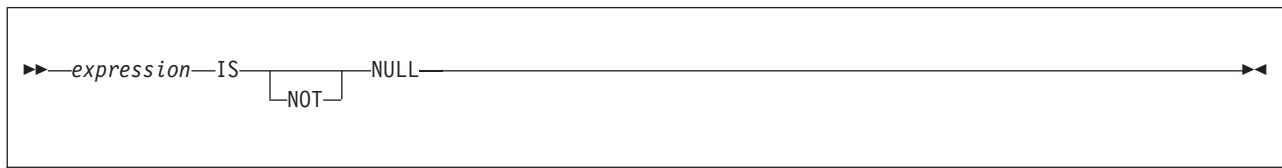
Example 8: In the following table, assume COL1 is a column that contains Unicode data. The table shows the results when the predicate in the first column is evaluated using the COL1 value in the second column:

Table 49. COL1 contain Unicode data

Predicates	COL1 values	Result
WHERE COL1 LIKE 'aaaAB%C'	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	False
WHERE COL1 LIKE 'aaaAB %C'	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	False
WHERE COL1 LIKE ''	'aaaABDZC'	False
	'aaaABdzxC'	False
	empty string	True
WHERE COL1 LIKE '%'	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	True
WHERE COL1 LIKE ' %'	'aaaABDZC'	True
	'aaaABdzxC'	True
	empty string	False
WHERE COL1 LIKE ' '	'aaaABDZC'	False
	'aaaABdzxC'	False
	empty string	False

NULL predicate

The NULL predicate tests for null values.



The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

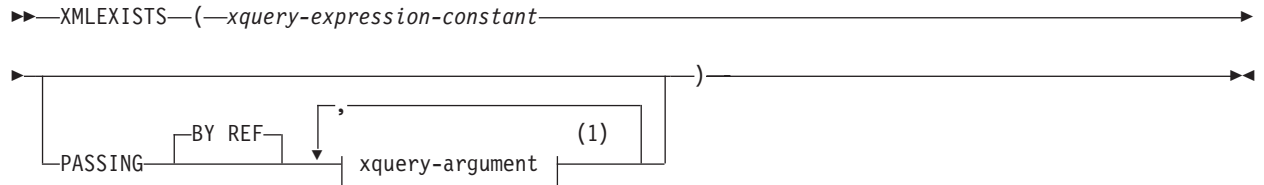
A parameter marker must not be specified for or within the expression.

Example: The following predicate is true whenever PHONENO has the null value, and is false otherwise.

```
PHONENO IS NULL
```

XMLEXISTS predicate

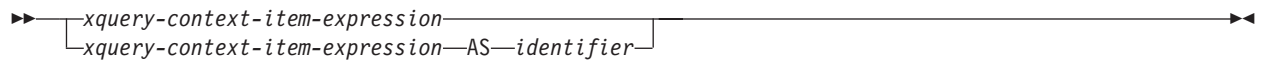
The XMLEXISTS predicate tests whether an XPath expression returns a sequence of one or more items.



Notes:

- 1 *xquery-context-item-expression* must not be specified more than one time.

xquery-argument



xquery-expression-constant

Specifies a character string constant that is interpreted as an XPath expression using supported XPath language syntax. See *DB2 XML Guide* for information about the XPath language syntax. The XPath expression is evaluated with the arguments specified in *xquery-argument*. *xquery-expression-constant* must not be an empty string or a string of all blanks.

PASSING

Specifies input values and the manner in which these values are passed to the XPath expression specified by *xquery-expression-constant*.

BY REF

Specifies that the XML input value arguments are to be passed by reference. When XML values are passed by reference, the XPath evaluation uses the input node trees, preserving all properties including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons that involve some nodes that are contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

xquery-argument

Specifies an argument to use in the evaluation of the XPath expression specified by *xquery-expression-constant*. A query argument is an expression that returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-argument* must not return ROWID, TIMESTAMP, binary string, REAL, DECFLOAT data types, or a character string data type that is bit data, and must not reference a sequence expression or a *OLAP-specification*.

An argument specifies a value and the manner in which that value is to be passed. How an argument in the PASSING clause is used in the XPath expression depends on whether the argument is specified as the *xquery-context-item-expression* or an *xquery-variable-expression*. The argument includes an SQL expression that is evaluated before passing the result to the XPath expression.

- If the resulting value is an XML value, it becomes an *input-xml-value*. It is passed by reference which means that the original values, not copies, are used in the evaluation of the XPath expression.
- If the resulting value is not an XML value, the result of the expression must be able to be cast to an XML value. The cast value becomes an *input-xml-value*. An empty string is converted to an XML empty string.
- If the resulting value is a null value, it is converted to an XML empty sequence if the argument is *xquery-variable-expression*. If the argument is *xquery-context-expression*, the XMLEXISTS predicates returns unknown.

xquery-context-item-expression

xquery-context-item-expression specifies the *initial context item* in the XPath expression specified by *xquery-expression-constant*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-context-item-expression must not be a sequence of more than one item. If the result of *xquery-context-item-expression* is an empty string, the XPath expression is evaluated with the initial context item set to an XML empty string.

If the *xquery-context-item-expression* is not specified or is an empty string, the initial context item in the XPath expression is undefined, and the XPath expression must not reference the initial context item. An XPath variable is not created for the context item expression.

If the *xquery-context-expression* is not specified or the *input-xml-value* that results from the *xquery-context-expression* is an XML empty sequence, the initial context item is undefined. If the XPath expression refers to the initial context item, it must be specified with a value that is not an XML empty sequence.

xquery-variable-expression

xquery-variable-expression specifies an argument to the XPath expression. An XPath variable is created for each *xquery-variable-expression*, and the XPath variable is set to the result of *xquery-argument-expression* cast to XML. If the result of *xquery-variable-expression* is an empty string, the XPath variable is set to an XML empty string. If *xquery-variable-expression* is null, the XPath variable is set to an XML empty sequence. For example, PASSING T.A + T.B as "sum" creates an XPath variable named sum. The scope of the XPath variables created from the PASSING clause is the XPath expression that is specified by *xquery-expression-constant*.

AS identifier

Specifies that the value that is generated by *xquery-variable-expression* will be passed to *xquery-expression-constant* as an XPath variable named *identifier*. The length of the name must not be longer than 128 bytes. The leading dollar sign (\$) that precedes variable names in the XPath language is not included in *identifier*. The name must be an XML NCName that is not the same as *identifier* for another *xquery-variable-expression* in the same PASSING clause.

The result of the predicate is determined as follows:

- The result is unknown if *xquery-context-item-expression* specified in the PASSING clause is a NULL value
- the result is false if the result of the XPath expression is an empty sequence
- the result is true in all other cases

If the evaluation of the XPath expression results in an error, XMLEXISTS returns an error. The XMLEXISTS predicate is not supported in ON clause of outer joins.

Example: Find all the purchase orders that buy a baby monitor. This example finds the product number for baby monitors from the product table and joins the result to the PurchaseOrders table. It then evaluates the XPath expression `//item[@partnum = $n]` for each row and returns those rows that contain an item element node with a partNum attribute that is equal to the product number of 'Baby Monitor'. The context item for the XPath expression is PO.POrder. An XPath variable, *\$n*, is created and initialized to the value of S.prodno:

```

SELECT S.prodno, count(*) as result
FROM PurchaseOrders PO, Products S
WHERE XMLEXISTS ('//item[@partNum = $n]'
                PASSING PO.POrder,
                        S.prodno AS "n")
AND S.prod_name = 'Baby Monitor';

```

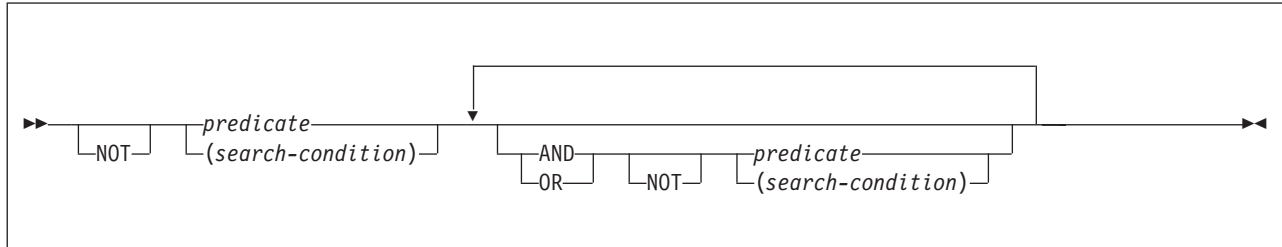
The results might be similar to the following:

Prodno	result

926-AA	1

Search conditions

A *search condition* specifies a condition that is true, false, or unknown about a given row or group. When the condition is true, the row or group qualifies for the results. When the condition is false or unknown, the row or group does not qualify.



The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in the following table, in which *P* and *Q* are any predicates:

Table 50. Truth table for AND and OR

<i>P</i>	<i>Q</i>	<i>P and Q</i>	<i>P or Q</i>
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false and NOT(false) is true, but NOT(unknown) is still unknown. The NOT logical operator has no affect on an unknown condition. The result of NOT(unknown) is still unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

Example 1: In the first of the search conditions below, AND is applied before OR. In the second, OR is applied before AND.

```
SALARY>:SS AND COMM>:CC OR BONUS>:BB
SALARY>:SS AND (COMM>:CC OR BONUS>:BB)
```


Example 2: In the first of the search conditions below, NOT is applied before AND. In the second, AND is applied before NOT.

```
NOT SALARY>:SS AND COMM>:CC
NOT (SALARY>:SS AND COMM>:CC)
```

Example 3: For the following search condition, AND is applied first. After the application of AND, the OR operators could be applied in either order without changing the result. DB2 can therefore select the order of applying the OR operators.

```
SALARY>:SS AND COMM>:CC OR BONUS>:BB OR SEX=:GG
```

Options affecting SQL

Certain DB2 precompiler options, DB2 subsystem parameters (set through the installation panels), bind options, options for CREATE PROCEDURE and ALTER PROCEDURE statements for native SQL procedures, and special registers affect how SQL statements can be composed or determine how SQL statements are processed.

The following table summarizes the effect of these options and shows where to find more information. (Some of the items are described in detail following the table, while other items are described elsewhere.)

Table 51. Summary of items affecting composition and processing of SQL statements

Precompiler option	Other ¹	Affects
	DYNAMICRULES bind option or the native SQL procedures option	The rules that DB2 applies to dynamic SQL statements. For details about authorization, see “Authorization IDs and dynamic SQL” on page 64. The option can also affect decimal point representation, string delimiters, and decimal arithmetic.
		For details about how DB2 applies the options to dynamic SQL statements when DYNAMICRULES bind, define, or invoke behavior is in effect, see “Precompiler options for dynamic statements” on page 250.
	USE FOR DYNAMICRULES	Use of options for dynamic statements when DYNAMICRULES bind, define, or invoke behavior is in effect. For details, see “Precompiler options for dynamic statements” on page 250.
COMMA PERIOD	DECIMAL POINT IS	Representation of decimal points in SQL statements. For details, see page “Decimal point representation” on page 251.
APOSTSQL QUOTESQL	SQL STRING DELIMITER	Representation of string delimiters in SQL statements. For details, see page “Apostrophes and quotation marks as string delimiters” on page 253.
	ASCII CODED CHAR SET	A numeric value that determines the CCSID of ASCII string data. For details, see page “Mixed data in character strings” on page 254.

Table 51. Summary of items affecting composition and processing of SQL statements (continued)

Precompiler option	Other ¹	Affects
CCSID	EBCDIC CODED CHAR SET	<p>A numeric value that determines the CCSID of EBCDIC string data and whether Katakana characters can be used in ordinary identifiers.</p> <p>For details, see page “Katakana characters for EBCDIC” on page 254.</p>
	UNICODE CCSID	<p>A numeric value that determines the CCSID of Unicode string data.</p> <p>For details, see page “Mixed data in character strings” on page 254.</p>
	MIXED DATA	<p>Use of ASCII or EBCDIC character strings with a mixture of SBCS and DBCS characters.</p> <p>For details, see page “Mixed data in character strings” on page 254.</p>
DATE TIME	DATE FORMAT TIME FORMAT LOCAL DATE LENGTH LOCAL TIME LENGTH	<p>Formatting of datetime strings.</p> <p>For details, see page “Formatting of datetime strings” on page 255.</p>
STDSQL		<p>Conformance with the SQL standard.</p> <p>For details, see page “SQL standard language” on page 255.</p>
NOFOR or STDSQL		<p>Whether the FOR UPDATE clause must be specified (in the SELECT statement of the DECLARE CURSOR statement).</p> <p>For details, see page “Positioned updates of columns” on page 256.</p>
CONNECT		<p>Whether the rules for the CONNECT(1) or CONNECT(2) precompiler option apply.</p> <p>For details about the precompiler option, see <i>DB2 Application Programming and SQL Guide</i>.</p>
	SQLRULES bind option	<p>Whether a CONNECT statement is processed with DB2 rules or SQL standard rules.</p>

Table 51. Summary of items affecting composition and processing of SQL statements (continued)

Precompiler option	Other ¹	Affects
	CURRENT RULES special register	<p>Whether the statements ALTER TABLE, CREATE TABLE, GRANT, and REVOKE are processed with DB2 rules or SQL standard rules. For details, see “CURRENT RULES” on page 145.</p> <p>Whether DB2 automatically creates the LOB table space, auxiliary table, and index on the auxiliary table for a LOB column in a base table. For details, see Creating a table with LOB columns.</p> <p>Whether DB2 automatically creates an index on a ROWID column that is defined with GENERATED BY DEFAULT. For details, see the description of the clause for “CREATE TABLE” on page 1079.</p> <p>Whether an external stored procedure runs as a main or subprogram. For details, see “CREATE PROCEDURE (external)” on page 1016.</p>
	SQLRULES bind option or CURRENT RULES special register	<p>Whether SQLCODE +236 is issued when the SQLDA provided on DESCRIBE or PREPARE INTO is too small and the result columns do not involve LOBs or distinct types. For details, see “DESCRIBE” on page 1237 and “SQL descriptor area (SQLDA)” on page 1656.</p> <p>Whether the SELECT privilege is required in a searched DELETE or UPDATE. For details, see “DELETE” on page 1224 or “UPDATE” on page 1521.</p>
DEC	DECIMAL ARITHMETIC or CURRENT PRECISION special register	<p>Whether DEC15 or DEC31 rules are used when both operands in a decimal operation have 15 digits or less.</p> <p>For details, see “Arithmetic with two decimal operands” on page 183.</p>

Note: ¹ The entries in this column are fields on installation panels unless otherwise noted.

For further details on precompiler options, see *DB2 Application Programming and SQL Guide*. For more details on bind options, see *DB2 Command Reference*.

Precompiler options for dynamic statements

Generally, dynamic statements use the application programming defaults specified on installation panel DSNTIPF. However, if the value of installation panel field USE FOR DYNAMICRULES is NO and DYNAMICRULES bind, define, or invoke behavior is in effect, certain precompiler options are used instead of the application programming defaults.

The following precompiler options are used instead of the application programming defaults:

- COMMA or PERIOD
- APOST or QUOTE
- APOSTSQL or QUOTESQL
- DEC(15) or DEC(31)

For some languages, the precompiler option defaults to a value and no alternative is allowed. If the value of installation panel field USE FOR DYNAMICRULES is YES, dynamic statements use the application programming defaults regardless of the value of DYNAMICRULES option.

For additional information on the effect of precompiler options and application programming defaults on:

- Decimal point representation, see “Decimal point representation.”
- String delimiters, see “Apostrophes and quotation marks as string delimiters” on page 253.
- Decimal arithmetic, see “Arithmetic with two decimal operands” on page 183.

For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 64.

DECFLOAT rounding mode

All views and SQL functions referenced in an SQL statement must either not have rounding mode information stored in the SYSENVIRONMENT catalog, or they must all have the same rounding mode information in the SYSENVIRONMENT catalog.

Decimal point representation

Decimal points in SQL statements are represented with either periods or commas.

Two values control the representation:

- The value of field DECIMAL POINT IS on installation panel DSNTIPF, which can be a comma (,) or period (.)
- COMMA or PERIOD, which are mutually exclusive DB2 precompiler options for COBOL

These values apply to SQL statements as follows:

- For a distributed operation, the decimal point is the first of the following values that applies:
 - The decimal point value specified by the requester
 - The value of field DECIMAL POINT IS on panel DSNTIPF at the DB2 where the package is bound
- Otherwise:
 - For static SQL statements:
 - In a COBOL program, the DB2 precompiler option COMMA or PERIOD determines the decimal point representation for every static SQL statement. If neither precompiler option is specified, the value of DECIMAL POINT IS at precompilation time determines the representation.
 - In non-COBOL programs, the decimal representation for static SQL statements is always the period.
 - For dynamic SQL statements:
 - If DYNAMICRULES run behavior applies, the decimal point is the value of field DECIMAL POINT IS on installation panel DSNTIPF at the local DB2 when the statement is prepared.

For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 64.
 - If DYNAMICRULES bind, define, or invoke behavior applies, and the value of install panel field USE FOR DYNAMICRULES is YES, the decimal point is the value of field DECIMAL POINT IS.

If bind, define, or invoke behavior applies, and field USE FOR DYNAMIC RULES is NO, the precompiler option determines the decimal point representation. For COBOL programs, which supports precompiler option COMMA or PERIOD, the decimal point representation is determined as described above for static SQL statements in COBOL programs. For programs written in other host languages, the default precompiler option, which can only be PERIOD, is used.

If the comma is the decimal point, these rules apply:

- In any context, a comma intended as a separator must be followed by a space. Such commas could appear, for example, in a VALUES clause, an IN predicate, or an ORDER BY clause in which numbers are used to identify columns.
- In any context, a comma intended as a decimal point must not be followed by a space.
- If the DECIMAL POINT IS field (and not the precompiler option) determines the comma as the decimal point, DB2 will recognize either a comma or a period as the decimal point in numbers in dynamic SQL.

Apostrophes and quotation marks as string delimiters

Precompiler options and DB2 installation panel fields control the representation of string delimiters in COBOL and SQL statements.

The following precompiler options control the representation of string delimiters:

- APOST and QUOTE are mutually exclusive DB2 precompiler options for COBOL. Their meanings are exactly what they are for the COBOL compilers:
 - APOST names the apostrophe (') as the string delimiter in COBOL statements.
 - QUOTE names the quotation mark (") as the string delimiter.Neither option applies to SQL syntax. Do not confuse them with the APOSTSQL and QUOTESQL options.
- APOSTSQL and QUOTESQL are mutually exclusive DB2 precompiler options for COBOL. Their meanings are:
 - APOSTSQL names the apostrophe (') as the string delimiter and the quotation mark (") as the escape character in SQL statements.
 - QUOTESQL names the quotation mark (") as the string delimiter and the apostrophe (') as the escape character in SQL statements.

These values apply to SQL statements as follows:

- For a distributed operation, the string delimiter is the first of the following values that applies:
 - The SQL string delimiter value specified by the requester
 - The value of the field SQL STRING DELIMITER on installation panel DSNTIPF at the DB2 where the package is bound
- Otherwise:
 - For static SQL statements:

In a COBOL program, the DB2 precompiler option APOSTSQL or QUOTESQL determines the string delimiter and escape character. If neither precompiler option is specified, the value of field SQL STRING DELIMITER on installation panel DSNTIPF determines the string delimiter and escape character.

In a non-COBOL program, the string delimiter is the apostrophe, and the escape character is the quotation mark.
 - For dynamic SQL statements:
 - If DYNAMICRULES run behavior applies, the string delimiter and escape character is the value of field SQL STRING DELIMITER on installation panel DSNTIPF at the local DB2 when the statement is prepared.

For a list of the DYNAMICRULES option values that specify run, bind, define, or invoke behavior, see Table 6 on page 64.
 - If DYNAMICRULES bind, define, or invoke behavior applies and the value of install panel field USE FOR DYNAMICRULES is YES, the string delimiter and escape character is the value of field SQL STRING DELIMITER.

If bind, define, or invoke behavior applies and USE FOR DYNAMICRULES is NO, the precompiler option determines the string delimiter and escape character. For COBOL programs, precompiler option APOSTSQL or QUOTESQL determines the string delimiter and escape character. If neither precompiler option is specified, the value of field SQL STRING DELIMITER determines them. For programs written in other host languages, the default precompiler option, which can only be APOSTSQL, determines the string delimiter and escape character.

Katakana characters for EBCDIC

Ordinary identifiers with an EBCDIC encoding scheme can contain Katakana characters if the DB2 installation is set to allow it.

The field EBCDIC CODED CHAR SET on installation panel DSNTIPF determines the system CCSIDs for EBCDIC-encoded data. Ordinary identifiers with an EBCDIC encoding scheme can contain Katakana characters if the field contains the value 5026 or 930. There are no corresponding precompiler options. EBCDIC CODED CHAR SET applies equally to static and dynamic statements. For dynamically prepared statements, the applicable value is always the one at the local DB2.

Mixed data in character strings

Mixed character data and graphic data are always allowed for Unicode, but for EBCDIC and ASCII, the specific installation of DB2 determines whether mixed data can be used.

The field MIXED DATA on installation panel DSNTIPF can have the value YES or NO for ASCII or EBCDIC character strings. The value YES indicates that character strings can contain a mixture of SBCS and DBCS characters. The value NO indicates that they cannot. Mixed character data and graphic data are always allowed for Unicode; that is the MIXED DATA field does not have an effect on Unicode data.

For static SQL statements, the value of the CCSID precompiler option or the derived CCSID for the DB2 coprocessor determines whether ASCII or EBCDIC character strings can contain mixed data. If a mixed CCSID is used, mixed strings are allowed. If a single-byte CCSID is used, mixed strings are not allowed.

For dynamic SQL statements, the CCSID that is selected to convert the dynamic statement text to UTF-8 determines whether ASCII or EBCDIC character strings can contain mixed data. The CCSID for a dynamic statement is determined from the SQLDA override (if any) for the host variable on the PREPARE statement, the value of the CURRENT ENCODING SCHEME special register, and the ENCODING bind option.

The value of MIXED DATA affects the parsing of SQL character string constants, the execution of the LIKE predicate, and the assignment of character strings to host variables when truncation is needed. It can also affect concatenation, as explained in “With the concatenation operator” on page 188. A value that applies to a statement executed at the local DB2 also applies to any statement executed at another server. An exception is the LIKE predicate, for which the applicable value of MIXED DATA is always the one at the statement's server.

The value of MIXED DATA also affects the choice of system CCSIDs for the local DB2 and the choice of data subtypes for character columns. When this value is YES, multiple CCSIDs are available for ASCII and EBCDIC data (SBCS, DBCS, and MIXED). The CCSID specified in the ASCII CODED CHAR SET or EBCDIC CODED CHAR SET field is the MIXED CCSID. In this case, DB2 derives the SBCS and MIXED CCSIDs from the DBCS CCSID specified installation panel DSNTIPF. Moreover, a character column can have any one of the allowable data subtypes—BIT, SBCS, or MIXED.

On the other hand, when MIXED DATA is NO, the only ASCII or EBCDIC system CCSIDs are those for SBCS data. Therefore, only BIT and SBCS can be data subtypes for character columns.

Formatting of datetime strings

The format for a datetime string that is in effect for a statement that is executed at the local DB2 is not necessarily in effect for a statement that is executed at a different server.

Fields on installation panel DSNTIP4 (DATE FORMAT, TIME FORMAT, LOCAL DATE LENGTH, and LOCAL TIME LENGTH) and DB2 precompiler options affect the formatting of datetime strings.

The formatting of datetime strings is described in “String representations of datetime values” on page 89. Unlike the subsystem parameters and options previously described, a value in effect for a statement executed at the local DB2 is not necessarily in effect for a statement executed at a different server. See “Restrictions on the use of local datetime formats” on page 92 for more information.

SQL standard language

DB2 SQL and the SQL standard are not identical. The STDSQL precompiler option addresses some of the differences.

- STDSQL(NO) indicates that conformance with the SQL standard is not intended. The default is the value of field STD SQL LANGUAGE on installation panel DSNTIP4 (which has a default of NO).
- STDSQL(YES)¹⁸ indicates that conformance with the SQL standard is intended.

When a program is precompiled with the STDSQL(YES) option, the following rules apply:

Declaring host variables: All host variable declarations except in Java and REXX must lie between pairs of BEGIN DECLARE SECTION and END DECLARE SECTION statements:

```
BEGIN DECLARE SECTION
-- one or more host variable declarations
END DECLARE SECTION
```

Separate pairs of these statements can bracket separate sets of host variable declarations.

Declarations for SQLCODE and SQLSTATE: The programmer must declare host variables for either SQLCODE or SQLSTATE, or both. SQLCODE should be defined as a fullword integer and SQLSTATE should be defined as a 5-byte character string. SQLCODE and SQLSTATE cannot be part of any structure. The variables must be declared in the DECLARE SECTION of a program; however, SQLCODE can be declared outside of the DECLARE SECTION when no host variable is defined for SQLSTATE. For PL/I, an acceptable declaration can look like this:

```
DECLARE SQLCODE BIN FIXED(31);
DECLARE SQLSTATE CHAR(5);
```

18. STDSQL(86) is a synonym, but STDSQL(YES) should be used.

In Fortran programs, the variable SQLCOD should be used for SQLCODE, and either SQLSTATE or SQLSTA can be used for SQLSTATE.

Definitions for the SQLCA: An SQLCA must not be defined in your program, either by coding its definition manually or by using the INCLUDE SQLCA statement. When STDSQL(YES) is specified, the DB2 precompiler automatically generates an SQLCA that includes the variable name SQLCADE instead of SQLCODE and SQLSTAT instead of SQLSTATE. After each SQL statement executes, DB2 assigns status information to SQLCODE and SQLSTATE, whose declarations are described above, as follows:

- SQLCODE: DB2 assigns the value in SQLCADE to SQLCODE. In Fortran, SQLCAD and SQLCOD are used for SQLCADE and SQLCODE, respectively.
- SQLSTATE: DB2 assigns the value in SQLSTAT to SQLSTATE. (In Fortran, SQLSTT and SQLSTA are used for SQLSTAT and SQLSTATE, respectively.)
- No declaration for either SQLSTATE or SQLCODE: DB2 assigns the value in SQLCADE to SQLCODE.

If the precompiler encounters an INCLUDE SQLCA statement, it ignores the statement and issues a warning message. The precompiler also does not recognize hand-coded definitions, and a hand-coded definition creates a compile-time conflict with the precompiler-generated definition. A similar conflict arises if definitions of SQLCADE or SQLSTAT, other than the ones generated by the DB2 precompiler, appear in the program.

Positioned updates of columns

Certain precompiler options affect the use of the FOR UPDATE clause to achieve positioned column updates.

The NOFOR precompiler option affects the use of the FOR UPDATE clause. The NOFOR option is in effect when either of the following are true:

- The NOFOR option is specified.
- The STDSQL(YES) option is in effect.

Otherwise, the NOFOR option is not in effect. The following table summarizes the differences when the option is in effect and when the option is not in effect:

Table 52. The NOFOR precompiler option

When NOFOR is in effect	When NOFOR is not in effect
The use of the FOR UPDATE clause in the SELECT statement of the DECLARE CURSOR statement is optional. This clause restricts updates to the specified columns and causes the acquisition of update locks when the cursor is used to fetch a row. If no columns are specified, positioned updates can be made to any updatable columns in the table or view that is identified in the first FROM clause in the SELECT statement. If the FOR UPDATE clause is not specified, positioned updates can be made to any columns that the program has DB2 authority to update.	The FOR UPDATE clause must be specified.

Table 52. The NOFOR precompiler option (continued)

When NOFOR is in effect	When NOFOR is not in effect
DBRMs must be built entirely in virtual storage, which might possibly increase the virtual storage requirements of the DB2 precompiler. However, creating DBRMs entirely in virtual storage might cause concurrency problems with DBRM libraries.	DBRMs can be built incrementally using the DB2 precompiler.

Precompiler options do not affect ODBC behavior.

Mappings from SQL to XML

DB2 maps SQL to XML data according to industry standards and performs several different mappings.

To construct XML data from SQL data, the following mappings are performed:

- SQL character sets to XML character sets
- SQL identifiers to XML names
- SQL data values to XML data values

DB2 maps SQL to XML data according to industry standards. For complete information, see *Information technology - Database languages - SQL- Part 14: XML-Related Specifications (SQL/XML) ISO/IEC 9075-14:2003*.

Mapping SQL character sets to XML character sets

The character set used for XML data is Unicode UTF-8. SQL character data is converted into Unicode when it is used in XML built-in functions.

Mapping SQL identifiers to XML names

Many SQL identifiers that contain certain characters must be escaped when the SQL identifier is converted into an XML name.

Strings that start with 'XML', in any case combination, are reserved for standardization, and characters such as '#', '{', and '}' are not allowed in XML names. Many SQL identifiers containing these characters have to be escaped when converting into XML names.

Full escaping is applied to SQL identifiers that are column names to derive an XML name. The mapping converts a colon (:) to `_x003A_`, `_x` to `_X005F_x`, and other restricted characters to a string of the form `_xUUUU_` where `xUUUU_` is the Unicode value for the character. An identifier with an initial 'xml' (in any case combination) is escaped by mapping the initial 'x' or 'X' to `_x0058_` or `_0078_`, respectively, while the partially escaped variant does not.

Mapping SQL data values to XML data values

SQL data values are mapped to XML values based on SQL data types.

The following data types are not supported and cannot be used as arguments to XML value constructors:

- ROWID
- Character strings that are defined with the FOR BIT DATA attribute
- Binary strings

- A string or a binary string distinct type that is based on a ROWID, FOR BIT DATA character string, or BLOB

For supported data types, the encoding scheme for XML values is Unicode.

Chapter 3. Functions

A *function* is an operation denoted by a function name followed by zero or more input values that are enclosed in parentheses. It represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*.

The types of functions are aggregate, scalar, and table. A built-in function is classified as a aggregate function or a scalar function. A user-defined function can be a column, scalar, or table function.

OLAP specification and functions

The RANK, DENSE_RANK, and ROW_NUMBER specifications are sometimes referred to as built-in 'functions'. Refer to "OLAP specification" on page 212 for more information on these specifications.

DB2 MQSeries functions

DB2 MQSeries functions integrate MQSeries messaging operations within SQL statements. The functions help you integrate MQSeries messaging with database applications. You can use the functions to access MQSeries messaging from within SQL statements and to combine MQSeries messaging with DB2 database access.

To use the MQSeries functions that provide support for XML messages, you must have IBM DB2 XML Extender installed. DB2 XML Extender introduces the user-defined data types DB2XML.XMLVARCHAR and DB2XML.XMLCLOB that are used to support XML data in message processing. XMLVARCHAR is defined as VARCHAR(3000) and XMLCLOB is defined as CLOB(2G). Refer to *DB2 XML Extender for z/OS Administration and Programming* for more information about DB2 XML Extender.

The functions can be scalar or table functions. For more information on using MQSeries functions, see the information on enabling MQSeries functions in *DB2 Installation Guide* and on programming techniques in *DB2 Application Programming and SQL Guide*.

Administrative scheduler functions

The administrative scheduler table functions provide information and status about the tasks that are scheduled to run using the administrative scheduler. The administrative scheduler provides the ability to run stored procedures, JCL jobs, and other administrative tasks according to a time or an event-based schedule. Refer to *DB2 Administration Guide* for additional information about the administrative scheduler.

The following table lists the functions that DB2 supports.

Table 53. Supported functions

Function name	Description
ABS	Returns the absolute value of its argument
ACOS	Returns the arc cosine of an argument as an angle, expressed in radians
ADD_MONTHS	Returns a date that represents the date argument plus the number of months argument

Table 53. Supported functions (continued)

Function name	Description
ADMIN_TASK_LIST	Returns a table with one row for each of the tasks that are defined in the administrative scheduler task list
ADMIN_TASK_STATUS	Returns a table with one row for each task in the administrative scheduler task list that contains the status for the last time the task was run
ASCII	Returns the ASCII code value of the leftmost character of the argument as an integer
ASCII_CHR	Returns the character that corresponds to the ASCII code value that is specified by the argument
ASCII_STR	Returns an ASCII version of the character or graphic string argument.
ASIN	Returns the arc sine of an argument as an angle, expressed in radians
ATAN	Returns the arc tangent of an argument as an angle, expressed in radians
ATANH	Returns the hyperbolic arc tangent of an argument as an angle, expressed in radians
ATAN2	Returns the arc tangent of <i>x</i> and <i>y</i> coordinates as an angle, expressed in radians
AVG	Returns the average of a set of numbers
BLOB	Returns a BLOB representation of its argument
BIGINT	Returns a big integer representation of its argument
BINARY	Returns a fixed-length binary string representation of its argument
CCSID_ENCODING	Returns the encoding scheme of a CCSID with a value of ASCII, EBCDIC, UNICODE, or UNKNOWN
CEILING	Returns the smallest integer greater than or equal to the argument
CHAR	Returns a fixed-length character string representation of its argument
CHARACTER_LENGTH	Returns the length of its argument in the number of string units that are specified
CLOB	Returns a CLOB representation of its argument
COALESCE	Returns the first argument in a set of arguments that is not null
COLLATION_KEY	Returns a string that represents the collation key of the argument in the specified collation
COMPARE_DECFLOAT	Returns a SMALLINT value that indicates whether two arguments are equal, or unordered, or whether one argument is greater than the other.
CONCAT	Returns the concatenation of two strings
CONTAINS	Returns a result about whether or not a match was found during a search of a text search index
CORRELATION	Returns the coefficient of the correlation of a set of number pairs
COS	Returns the cosine of an argument that is expressed as an angle in radians
COSH	Returns the hyperbolic cosine of an argument that is expressed as an angle in radians
COUNT	Returns the number of rows or values in a set of rows or values
COUNT_BIG	Same as COUNT, except the result can be greater than the maximum value of an integer

Table 53. Supported functions (continued)

Function name	Description
COVARIANCE or COVARIANCE_SAMP	Returns the (population) covariance of a set of number pairs
DATE	Returns a date derived from its argument
DAY	Returns the day part of its argument
DAYOFMONTH	Similar to DAY
DAYOFWEEK	Returns an integer in the range of 1 to 7, where 1 represents Sunday
DAYOFWEEK_ISO	Returns an integer in the range of 1 to 7, where 1 represents Monday
DAYOFYEAR	Returns an integer in the range of 1 to 366, where 1 represents January 1
DAYS	Returns an integer representation of a date
DBCLOB	Returns a DBCLOB representation of its argument
DECIMAL or DEC	Returns a decimal representation of its argument
DECFLOAT	Returns a DECFLOAT representation of its argument
DECFLOAT_SORTKEY	Returns a binary value that can be used when sorting DECFLOAT values
DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, or DECRYPT_DB	Returns the decrypted value of an encrypted argument
DEGREES	Returns the number of degrees for an argument that is expressed in radians
DIFFERENCE	Returns a value that represents the difference between the sounds of two strings based on applying the SOUNDEX function to the strings.
DIGITS	Returns a character string representation of a number
DOUBLE or DOUBLE_PRECISION	Returns a double precision floating-point representation of its argument
DSN_XMLVALIDATE	Returns an XML value that is the result of applying XML schema validation to the first argument.
DSN_XMLVALIDATE	Returns a varying length binary value that can only be used with the XMLPARSE function
EBCDIC_CHR	Returns the character that corresponds to the EBCDIC code value that is specified by the argument
EBCDIC_STR	Returns an EBCDIC version of the string argument
ENCRYPT_TDES	Returns the argument as an encrypted value
EXP	Returns the exponential function of an argument
EXTRACT	Returns a portion of a date or timestamp based on its arguments
FLOAT	Same as DOUBLE
FLOOR	Returns the largest integer that is less than or equal to the argument
GENERATE_UNIQUE	Returns a character string of bit data that is unique compared to any other execution of the function
GETHINT	Returns the embedded password hint from encrypted data, if one exists
GETVARIABLE	Returns a varying-length character string representation of the value of a session variable
GRAPHIC	Returns a fixed-length graphic string representation of its argument
HEX	Returns a hexadecimal representation of its argument

Table 53. Supported functions (continued)

Function name	Description
HOUR	Returns the hour part of its argument
IDENTITY_VAL_LOCAL	Returns the most recently assigned value for an identity column
IFNULL	Returns the first argument in a set of two arguments that is not null
INSERT	Returns a string that is composed of an argument inserted into another argument at the same position where some number of bytes have been deleted
INTEGER or INT	Returns an integer representation of its argument
JULIAN_DAY	Returns an integer that represents the number of days from January 1, 4712 B.C.
LAST_DAY	Returns a date that represents the last day of the month of the date argument
LCASE	Returns a string with the characters converted to lowercase
LEFT	Returns a string that consists of the specified number of leftmost bytes or the specified string units
LENGTH	Returns the length of its argument
LN	Returns the natural logarithm of an argument
LOCATE	Returns the starting position of one string within another string
LOCATE_IN_STRING	Returns the starting position of the first occurrence of one string within another string
LOG10	Returns the base 10 logarithm of an argument
LOWER	Returns a string with the characters converted to lowercase
LPAD	Returns a string that is padded on the left with blanks or a specified string
LTRIM	Returns the characters of a string with the leading blanks or hexadecimal zeros removed
MAX (aggregate)	Returns the maximum value in a set of column values
MAX (scalar)	Returns the maximum value in a set of values
MICROSECOND	Returns the microsecond part of its argument
MIDNIGHT_SECONDS	Returns an integer in the range of 0 to 86400 that represents the number of seconds between midnight and the argument
MIN (aggregate)	Returns the minimum value in a set of column values
MIN (scalar)	Returns the minimum value in a set of values
MINUTE	Returns the minute part of its argument
MOD	Returns the remainder of one argument divided by a second argument
MONTH	Returns the month part of its argument
MONTHS_BETWEEN	Returns an estimate of the number of months between two arguments
MQPUBLISH	Publishes a message to the specified MQSeries publisher, and returns a varying-length character string that indicates whether the function was successful or unsuccessful
MQPUBLISHXML	Publishes the XML data in a message to the specified MQSeries publisher

Table 53. Supported functions (continued)

Function name	Description
MQREAD	Returns a message from a specified MQSeries location (return value of VARCHAR) without removing the message from the queue
MQREADALL	Returns a table containing the messages and message metadata from a specified MQSeries location with a VARCHAR column and without removing the messages from the queue
MQREADALLCLOB	Returns a table containing the messages and message metadata from a specified MQSeries location with a CLOB column and without removing the messages from the queue
MQREADALLXML	Returns a table containing the messages and message metadata from a specified MQSeries location with a column that contains XML data without removing the messages from the queue
MQREADCLOB	Returns a message from a specified MQSeries location (return value of CLOB) without removing the message from the queue
MQREADXML	Returns an XML message from a specified MQSeries location without removing the message from the queue
MQRECEIVE	Returns a message from a specified MQSeries location (return value of VARCHAR) with removal of message from the queue
MQRECEIVEALL	Returns a table containing the messages and message metadata from a specified MQSeries location with a VARCHAR column and with removal of messages from the queue
MQRECEIVEALLCLOB	Returns a table containing the messages and message metadata from a specified MQSeries location with a CLOB column and with removal of messages from the queue
MQRECEIVEALLXML	Returns a table containing the messages and message metadata from a specified MQSeries location with column that contains XML data and with removal of messages from the queue
MQRECEIVECLOB	Returns a message from a specified MQSeries location (return value of CLOB) with removal of message from the queue
MQRECEIVEXML	Returns a message from a specified MQSeries location (return value of XML data in a user-defined data type that is based on VARCHAR) with removal of message from the queue
MQSEND	Sends data to a specified MQSeries location, and returns a varying-length character string that indicates whether the function was successful or unsuccessful
MQSENDXML	Sends XML data to a specified MQSeries location
MQSENDXMLFILE	Sends data contained in an XML file that is up to 3K in size to a specified MQSeries location
MQSENDXMLFILECLOB	Sends data contained in an XML file that is up to 1M in size to a specified MQSeries location
MQSUBSCRIBE	Registers a subscription to MQSeries messages that are published on a specified topic, and returns a varying-length character string that indicates whether the function was successful or unsuccessful
MQUNSUBSCRIBE	Unregisters an existing subscription to MQSeries messages that are published on a specified topic, and returns a varying-length character string that indicates whether the function was successful or unsuccessful
MULTIPLY_ALT	Returns the product of the two arguments as a decimal value, used when the sum of the argument precisions exceeds 31

Table 53. Supported functions (continued)

Function name	Description
NEXT_DAY	Returns a timestamp that represents the first weekday, specified by the second argument, after the date argument
NORMALIZE_DECFLOAT	Returns a DECFLOAT value that is the result of normalizing the input argument
NORMALIZE_STRING	Returns a string value that is the result of normalizing the input Unicode value
NULLIF	Returns NULL if the arguments are equal; else the first argument
OVERLAY	Returns a string that is composed of an argument inserted into another argument at the same position where some number of bytes have been deleted
POSITION	Returns the position of the first occurrence of an argument within another argument where the position is expressed in terms of the string units that are specified
POSSTR	Returns the position of the first occurrence of an argument within another argument
POWER	Returns the value of one argument raised to the power of a second argument
QUANTIZE	Returns a DECFLOAT value that is equal in value (except for any rounding) and sign to the first argument and which has an exponent set to be equal to the exponent of the second argument
QUARTER	Returns an integer in the range of 1 to 4 that represents the quarter of the year for the date specified in the argument
RADIANS	Returns the number of radians for an argument that is expressed in degrees
RAISE_ERROR	Raises an error in the SQLCA with the specified SQLSTATE and error description
RAND	Returns a double precision floating-point random number
REAL	Returns a single precision floating-point representation of its argument
REPEAT	Returns a character string composed of an argument repeated a specified number of times
REPLACE	Returns a string in which all occurrences of an argument within a second argument are replaced with a third argument
RID	Returns the RID of a row
RIGHT	Returns a string that consists of the specified number of rightmost bytes or specified string unit
ROUND	Returns a number rounded to the specified number of places to the right or left of the decimal place
ROUND_TIMESTAMP	Returns a timestamp rounded to the unit specified by the timestamp format string
ROWID	Returns a row ID representation of its argument
RPAD	Returns a string that is padded on the right with blanks or a specified string
RTRIM	Returns the characters of an argument with the trailing blanks or hexadecimal zeros removed
SCORE	Returns a relevance score that measures how well a document matches the query used to search a text search index

Table 53. Supported functions (continued)

Function name	Description
SECOND	Returns the second part of its argument
SIGN	Returns the sign of an argument
SIN	Returns the sine of an argument that is expressed as an angle in radians
SINH	Returns the hyperbolic sine of an argument that is expressed as an angle in radians
SMALLINT	Returns a small integer representation of its argument
SOAPHTTPC or SOAPHTTPV	Returns a CLOB or VARCHAR representation of XML data from a request to a web service
SOAPHTTPNC or SOAPHTTPNV	Returns a complete CLOB or VARCHAR representation of XML data from a complete request to a web service
SOUNDEX	Returns a value that represents the sound of the words in the argument.
SPACE	Returns a string that consists of the number of blanks the argument specifies
SQRT	Returns the square root of its argument
STDDEV or STDDEV_SAMP	Returns the standard deviation ($/n$), or the sample standard deviation ($/n-1$), of a set of numbers
STRIP	Returns the characters of a string with the blanks (or specified character) at the beginning, end, or both beginning and end of the string removed
SUBSTR	Returns a substring of a string
SUBSTRING	Returns a substring of a string using the specified string units
SUM	Returns the sum of a set of numbers
TAN	Returns the tangent of an argument that is expressed as an angle in radians
TANH	Returns the hyperbolic tangent of an argument that is expressed as an angle in radians
TIME	Returns a time derived from its argument
TIMESTAMP	Returns a timestamp derived from its arguments
TIMESTAMPADD	Returns a timestamp derived from adding the specified interval to a timestamp
TIMESTAMP_FORMAT	Returns a timestamp for a character string expression, using a specified format to interpret the string
TIMESTAMP_ISO	Returns a timestamp derived from its arguments
TIMESTAMPDIFF	Returns an estimated number of the specified intervals based on the difference between two timestamps
TOTALORDER	Returns a SMALLINT value that indicates the comparison order of two arguments
TRANSLATE	Returns a string with one or more characters translated
TRUNCATE	Returns a number truncated to the specified number of places to the right or left of the decimal point
TRUNC_TIMESTAMP	Returns a timestamp truncated to the unit specified by the timestamp format string
UCASE	Returns a string with the characters converted to uppercase

Table 53. Supported functions (continued)

Function name	Description
UNICODE	Returns the Unicode (UTF-16) code value of the leftmost character of the argument as an integer
UNICODE_STR	Returns a string in Unicode (UTF-8 or UTF-16) that represents a Unicode encoding of the argument
UPPER	Returns a string with the characters converted to uppercase
VALUE	Same as COALESCE
VARBINARY	Returns a varying-length binary string representation of its argument
VARCHAR	Returns the varying-length character string representation of its argument
VARCHAR_FORMAT	Returns a varying-length character string representation of a timestamp, with the string in a specified format
VARGRAPHIC	Returns a varying-length graphic string representation of its argument
VARIANCE or VARIANCE_SAMP	Returns the variance, or sample variance, of a set of numbers
WEEK	Returns an integer that represents the week of the year with Sunday as the first day of the week
WEEK_ISO	Returns an integer that represents the week of the year with Monday as first day of a week
XMLAGG	Returns an XML type that represents a concatenation of XML elements from a collection of XML elements
XMLATTRIBUTES	Returns an XML sequence that contains an XQuery attribute node for each non-null argument
XMLCOMMENT	Returns an XML value with a single comment node from a string expression
XMLCONCAT	Returns an XML value that represents a forest of XML elements generated by concatenating a variable number of arguments
XMLDOCUMENT	Returns an XML value with a single document node and zero or more nodes as its children
XMLELEMENT	Returns an XML value that represents an XML element
XMLFOREST	Returns an XML value that represents a forest of XML elements that all share a specific pattern
XMLNAMESPACES	Returns the declaration of one or more XML namespaces
XMLPARSE	Returns an XML value from parsing the argument as an XML document
XMLPI	Returns an XML value with a single processing instruction node
XMLQUERY	Returns an XML value from the evaluation of an XPath expression against a set of arguments
XMLSERIALIZE	Returns an SQL character string or a BLOB value from an XML value
XMLTABLE	Returns a result table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables
XMLTEXT	Returns an XML value with a single text node that contains the value of the argument
YEAR	Returns the year part of its argument

Aggregate functions

An aggregate function receives a set of values for each argument (such as the values of a column) and returns a single-value result for the set of input values. Certain rules apply to all aggregate functions.

The following information applies to all aggregate functions, except for the COUNT(*) and COUNT_BIG(*) variations of the COUNT and COUNT_BIG functions and the XMLAGG function.

The argument of an aggregate function is a set of values derived from an expression. The expression must not include another aggregate function. The scope of the set is a group or an intermediate result table.

If a GROUP BY clause is specified in a query and the intermediate result from the FROM, WHERE, GROUP BY, and HAVING clauses is the empty set, then the aggregate functions are not applied and the result of the query is the empty set.

If the GROUP BY clause is not specified in a query and the intermediate result table of the FROM, WHERE, and HAVING clauses is the empty set, then the aggregate functions are applied to the empty set.

For example, the result of the following SELECT statement is the number of distinct values of JOB for employees in department D11:

```
SELECT COUNT(DISTINCT JOB)
FROM DSN8910.EMP
WHERE WORKDEPT = 'D11';
```

The keyword DISTINCT is not an argument of the function but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, redundant duplicate values are eliminated. If ALL is implicitly or explicitly specified, redundant duplicate values are not eliminated. DISTINCT must not be specified preceding an XML value.

When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

An aggregate function can be used in a WHERE clause only if that clause is part of a subquery of a HAVING clause and the column name specified in the expression is a correlated reference to a group. If the expression includes more than one column name, each column name must be a correlated reference to the same group.

The result of the COUNT and COUNT_BIG functions cannot be the null value. As specified in the description of AVG, MAX, MIN, STDDEV, SUM, and VARIANCE, the result is the null value when the function is applied to an empty set. However, the result is also the null value when the function is specified in an outer select list, the argument is given by an arithmetic expression, and any evaluation of the expression causes an arithmetic exception (such as division by zero).

If the argument values of an aggregate function are strings from a column with a field procedure, the function is applied to the encoded form of the values and the

result of the function inherits the field procedure.

AVG

The AVG function returns the average of a set of numbers.



The schema is SYSIBM.

The argument values can be of any built-in numeric data type, and their sum must be within the range of the data type of the result.

The data type of the result is determined as follows:

- DECFLOAT(34) if the argument is DECFLOAT(*n*).
- Large integer is the argument is small integer.
- Double precision floating-point is the argument is single precision floating-point.
- Otherwise, the result is the same as the data type of the argument.

The result can be null.

If the data type of the argument values is decimal with precision *p* and scale *s*, the precision (P) and scale (S) of the result depend on *p* and the decimal precision option:

- If *p* is greater than 15 or the DEC31 option is in effect, P is 31 and S is $\max(0, 28-p+s)$.
- Otherwise, P is 15 and S is $15-p+s$.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are also eliminated.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the average value of the set. The order in which the summation part of the operation is performed is undefined but every intermediate result must be within the range of the result data type.

If the type of the result is integer, the fractional part of the average is lost.

Example: Assuming DEC15, set the DECIMAL(15,2) variable AVERAGE to the average salary in department D11 of the employees in the sample table DSN8910.EMP.

```
EXEC SQL SELECT AVG(SALARY)
        INTO :AVERAGE
        FROM DSN8910.EMP
        WHERE WORKDEPT = 'D11';
```

CORRELATION

The CORRELATION function returns the coefficient of the correlation of a set of number pairs.

►►CORRELATION(*expression-1*,*expression-2*)◄◄

The schema is SYSIBM.

The argument values must each be the value of any built-in numeric data type.

If an argument is DECFLOAT(*n*), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point. The result is between -1 and 1. The result can be null.

The function is applied to the set of (*expression-1*, *expression-2*) pairs derived from the argument values by the elimination of all pairs for which either *expression-1* or *expression-2* is null.

If the function is applied to an empty set, or if either STDDEV(*expression-1*) or STDDEV(*expression-2*) is equal to zero, the result is a null value. Otherwise, the result is the correlation coefficient for the value pairs in the set. The result is equivalent to the following expression:

$$\frac{\text{COVARIANCE}(\textit{expression-1}, \textit{expression-2})}{(\text{STDDEV}(\textit{expression-1}) * \text{STDDEV}(\textit{expression-2}))}$$

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

CORR can be specified as a synonym for CORRELATION.

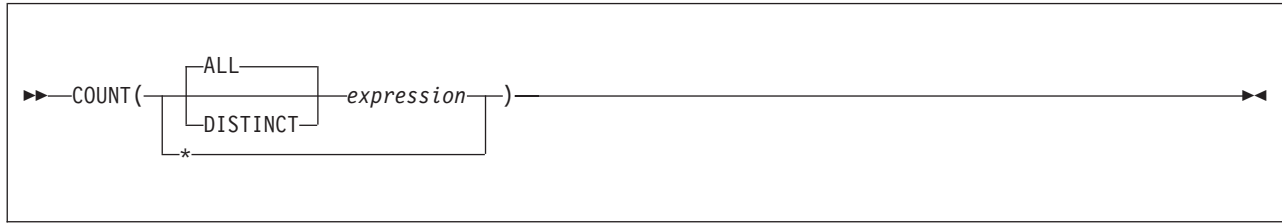
Example: Using sample table DSN8910.EMP, set the host variable :corrln (double-precision floating point) to the correlation between the salary and the bonus for those employees in department (WORKDEPT) 'A00'.

```
SELECT CORRELATION(SALARY, BONUS) INTO :corrln
FROM DSN8910.EMP WHERE WORKDEPT = 'A00';
```

:corrln is set to approximately 9.99853953399538E-001.

COUNT

The COUNT function returns the number of rows or values in a set of rows or values.



The schema is SYSIBM.

|
|

The argument values can be of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

The result is a large integer. The result cannot be null.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set. Any row that includes only null values is included in the count.

The argument of COUNT(expression) or COUNT(ALL expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of nonnull values in the set, including duplicates.

The argument of COUNT(DISTINCT expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values and redundant duplicate values. The result is the number of different nonnull values in the set.

Example 1: Set the integer host variable FEMALE to the number of females represented in the sample table DSN8910.EMP.

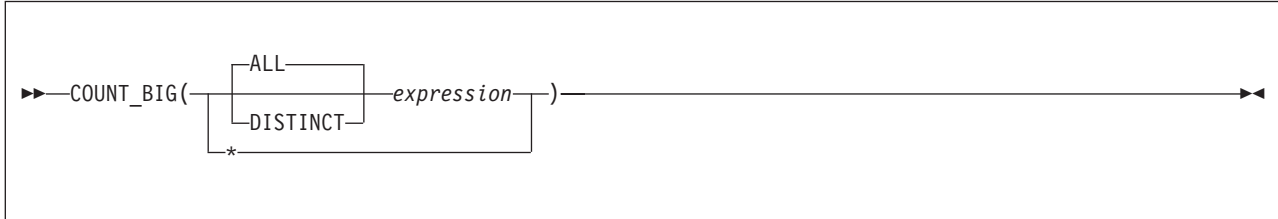
```
EXEC SQL SELECT COUNT(*)  
  INTO :FEMALE  
  FROM DSN8910.EMP  
 WHERE SEX = 'F';
```

Example 2: Set the integer host variable FEMALE_IN_DEPT to the number of departments that have at least one female as a member.

```
EXEC SQL SELECT COUNT(DISTINCT WORKDEPT)  
  INTO :FEMALE_IN_DEPT  
  FROM DSN8910.EMP  
 WHERE SEX = 'F';
```

COUNT_BIG

The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of an integer.



The schema is SYSIBM.

| The argument values can be of any built-in data type other than a BLOB, CLOB,
| DBCLOB, or XML.

The result of the function is a decimal number with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT_BIG(expression) or COUNT_BIG(ALL expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of nonnull values in the set, including duplicates.

The argument of COUNT_BIG(DISTINCT expression) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and redundant duplicate values. The result is the number of different nonnull values in the set.

Example 1: Set the integer host variable FEMALE to the number of females represented in the sample table DSN8910.EMP.

```
EXEC SQL SELECT COUNT_BIG(*)  
        INTO :FEMALE  
        FROM DSN8910.EMP  
        WHERE SEX = 'F';
```

Example 2: Set the integer host variable FEMALE_IN_DEPT to the number of departments that have at least one female as a member.

```
EXEC SQL SELECT COUNT_BIG(DISTINCT WORKDEPT)  
        INTO :FEMALE_IN_DEPT  
        FROM DSN8910.EMP  
        WHERE SEX = 'F';
```

Example 3: To create a sourced function that is similar to the built-in COUNT_BIG function, the definition of the sourced function must include the type of the column that can be specified when the new function is invoked. In this example, the CREATE FUNCTION statement creates a sourced function that takes a CHAR column as input and uses COUNT_BIG to perform the counting. The result is

returned as a double precision floating-point number. The query shown counts the number of unique departments in the sample employee table.

```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE
SOURCE SYSIBM.COUNT_BIG(CHAR());
SET CURRENT PATH RICK, SYSTEM PATH;
SELECT COUNT(DISTINCT WORKDEPT) FROM DSN8910.EMP;
```

The empty parenthesis in the parameter list for the new function (RICK.COUNT) means that the input parameter for the new function is the same type as the input parameter for the function named in the SOURCE clause. The empty parenthesis in the parameter list in the SOURCE clause (SYSIBM.COUNT_BIG) means that the length attribute of the CHAR parameter of the COUNT_BIG function is ignored when DB2 locates the COUNT_BIG function.

COVARIANCE or COVARIANCE_SAMP

The COVARIANCE and COVARIANCE_SAMP functions return the covariance (population) of a set of number pairs.

A diagram showing two function names, COVARIANCE and COVARIANCE_SAMP, enclosed in a bracket. This bracketed pair is followed by the argument (expression-1, expression-2). A long horizontal arrow points from the bracketed functions to the right, ending in a double arrowhead.

The schema is SYSIBM.

The argument values must each be the value of any built-in numeric data type.

If an argument is DECFLOAT(*n*), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point. The result can be null.

The function is applied to the set of (*expression-1*, *expression-2*) pairs that are derived from the argument values by the elimination of all pairs for which either *expression-1* or *expression-2* is null.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the covariance of the value pairs in the set. The result is equivalent to the following:

For COVARIANCE:

1. Let *avgexp1* be the result of AVG(*expression-1*) and let *avgexp2* be the result of AVG(*expression-2*).
2. The result of COVARIANCE(*expression-1*, *expression-2*) is AVG((*expression-1* - *avgexp1*) * (*expression-2* - *avgexp2*))

For COVARIANCE_SAMP:

1. Let *samp_avgexp1* be the result of SUM(*expression-1*)/*n*-1 and let *samp_avgexp2* be the result of SUM(*expression-2*)/*n*-1.
2. The result of COVARIANCE_SAMP(*expression-1*, *expression-2*) is AVG((*expression-1* - *samp_avgexp1*) * (*expression-2* - *samp_avgexp2*))

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

COVAR can be specified as a synonym for COVARIANCE.

COVAR_SAMP can be specified as a synonym for COVARIANCE_SAMP.

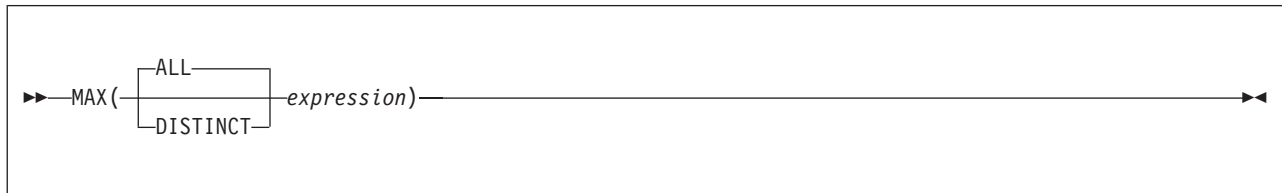
Example: Using sample table DSN8910.EMP, set the host variable *covarnce* (double-precision floating point) to the covariance between the salary and the bonus for those employees in department (WORKDEPT) 'A00'.

```
SELECT COVARIANCE(SALARY, BONUS) INTO :covarnce
FROM EMPLOYEE WHERE WORKDEPT = 'A00';
```

covarnce is set to approximately 1.68888888888889E+006.

MAX

The MAX function returns the maximum value in a set of values.



The schema is SYSIBM.

| The argument values can be of any built-in data type other than a BLOB, CLOB,
| DBCLOB, row ID, or XML. Character string arguments and binary string
| arguments cannot have a length attribute greater than 255, and graphic string
| arguments cannot have a length attribute greater than 127.

| The data type of the result and its other attributes (for example, the length and
| CCSID of a string or a datetime value) are the same as the data type and attributes
| of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

Example 1: Set the DECIMAL(8,2) variable *MAX_SALARY* to the maximum monthly salary of the employees represented in the sample table DSN8910.EMP.

```
EXEC SQL SELECT MAX(SALARY) / 12  
          INTO :MAX_SALARY  
          FROM DSN8910.EMP;
```

Example 2: Find the surname that comes last in the collating sequence for the employees represented in the sample table DSN8910.EMP. Set the VARCHAR(15) variable *LAST_NAME* to that surname.

```
EXEC SQL SELECT MAX(LASTNAME)  
          INTO :LAST_NAME  
          FROM DSN8910.EMP;
```

MIN

The MIN function returns the minimum value in a set of values.



The schema is SYSIBM.

| The argument values can be of any built-in data type other than a BLOB, CLOB,
| DBCLOB, row ID, or XML. Character string arguments and binary string arguments
| cannot have a length attribute greater than 255, and graphic string arguments
| cannot have a length attribute greater than 127.

| The data type of the result and its other attributes (for example, the length and
| CCSID of a string or a datetime value) are the same as the data type and attributes
| of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

Example 1: Set the DECIMAL(15,2) variable *MIN_SALARY* to the minimum monthly salary of the employees represented in the sample table DSN8910.EMP.

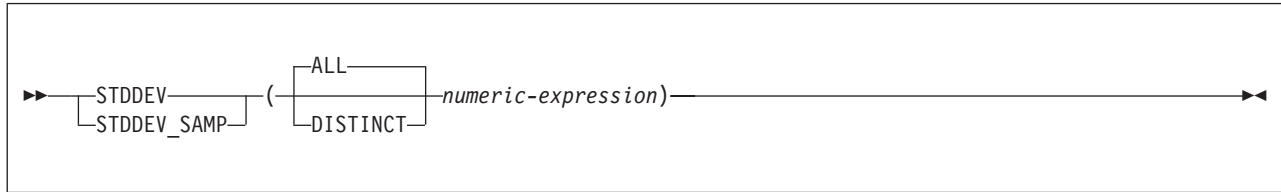
```
EXEC SQL SELECT MIN(SALARY) / 12  
          INTO :MIN_SALARY  
          FROM DSN8910.EMP;
```

Example 2: Find the surname that comes first in the collating sequence for the employees represented in the sample table DSN8910.EMP. Set the VARCHAR(15) variable *FIRST_NAME* to that surname.

```
EXEC SQL SELECT MIN(LASTNAME)  
          INTO :FIRST_NAME  
          FROM DSN8910.EMP;
```

STDDEV or STDDEV_SAMP

The STDDEV or STDDEV_SAMP function returns the standard deviation ($/n$), or the sample standard deviation ($/n-1$), of a set of numbers.



The schema is SYSIBM.

The function returns the biased standard deviation ($/n$) or the sample standard deviation ($/n-1$) of a set of numbers, depending on which keyword is specified:

STDDEV

The formula that is used to calculate the biased standard deviation is logically equivalent to:

$$\text{STDDEV} = \text{SQRT}(\text{VAR})$$

STDDEV_SAMP

The formula that is used to calculate the sample standard deviation is logically equivalent to:

$$\text{STDDEV} = \text{SQRT}(\text{VARIANCE_SAMP})$$

The argument values must each be the value of any built-in numeric data type, and their sum must be within the range of the data type of the result.

If the argument is DECFLOAT(n), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point. The result can be null.

Before the function is applied to the set of values derived from the argument values, null values are eliminated. If DISTINCT is specified, redundant duplicate values are also eliminated.

If the function is applied to an empty set, the result is the null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

STDDEV_POP can be specified as a synonym for STDDEV.

Example: Using sample table DSN8910.EMP, set the host variable *DEV*, which is defined as double precision floating-point, to the standard deviation of the salaries for the employees in department 'A00' (WORKDEPT='A00').

```
SELECT STDDEV(SALARY)
  INTO :DEV
  FROM DSN8910.EMP
 WHERE WORKDEPT = 'A00';
```

For this example, host variable *DEV* is set to a double precision float-pointing number with an approximate value of '9742.43'.

SUM

The SUM function returns the sum of a set of numbers.



The schema is SYSIBM.

The argument values can be of any built-in numeric data type, and their sum must be within the range of the data type of the result.

The data type of the result is determined as follows:

- DECFLOAT(34) if the argument is DECFLOAT(*n*).
- Large integer if the argument is small integer.
- Double precision floating-point if the argument is single precision floating-point.
- Otherwise, the result is the same as the data type of the argument.

The result can be null.

If the data type of the argument values is decimal, the scale of the result is the same as the scale of the argument values, and the precision of the result depends on the precision of the argument values and the decimal precision option:

- If the precision of the argument values is greater than 15 or the DEC31 option is in effect, the precision of the result is $\min(31, P+10)$, where *P* is the precision of the argument values.
- Otherwise, the precision of the result is 15.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are also eliminated.

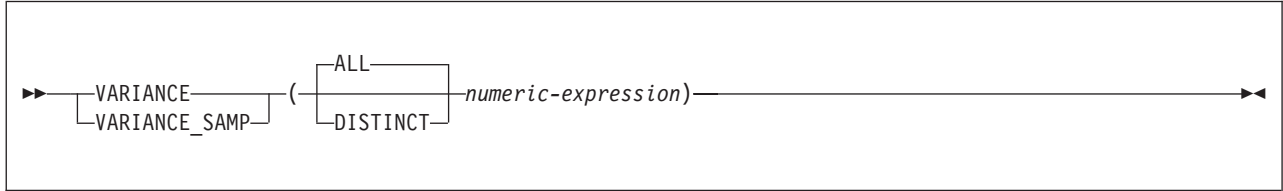
If the function is applied to an empty set, the result is the null value. Otherwise, the result is the sum of the values in the set. The order in which the summation is performed is undefined but every intermediate result must be within the range of the result data type.

Example: Set the large integer host variable *INCOME* to the total income from all sources (salaries, commissions, and bonuses) of the employees represented in the sample table DSN8910.EMP. If DEC31 is not in effect, the resultant sum is DECIMAL(15,2) because all three columns are DECIMAL(9,2).

```
EXEC SQL SELECT SUM(SALARY+COMM+BONUS)
           INTO :INCOME
           FROM DSN8910.EMP;
```


VARIANCE or VARIANCE_SAMP

The VARIANCE function returns the biased variance ($/n$) of a set of numbers. The VARIANCE_SAMP function returns the sample variance ($/n-1$) of a set of numbers.



The schema is SYSIBM.

The function returns the biased variance ($/n$) or the sample variance ($/n-1$) of a set of numbers, depending on which keyword is specified.

VARIANCE

The formula that is used to calculate the biased variance is logically equivalent to:

$$\text{VARIANCE} = \text{SUM}(X**2) / \text{COUNT}(X) - (\text{SUM}(X) / \text{COUNT}(X))**2$$

VARIANCE_SAMP

The formula that is used to calculate the sample variance is logically equivalent to:

$$\text{VARIANCE_SAMP} = (\text{SUM}(X**2) - ((\text{SUM}(X)**2) / (\text{COUNT}(*)))) / (\text{COUNT}(*) - 1)$$

The argument values can be of any built-in numeric type, and their sum must be within the range of the data type of the result. Before the function is applied to the set of values derived from the argument values, null values are eliminated. If DISTINCT is specified, redundant duplicate values are also eliminated.

If the argument is DECFLOAT(n), the result of the function is DECFLOAT(34). Otherwise, the result of the function is double precision floating-point.

The result can be null. If the set of values is empty, the result is the null value. Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Alternative syntax and synonyms:

- VAR or VAR_POP can be specified as synonym for VARIANCE
- VAR_SAMP can be specified as a synonym for VARIANCE_SAMP

Example 1: Using sample table DSN8910.EMP, set host variable VARNCE, which is defined as double precision floating-point, to the variance of the salaries (SALARY) for those employees in department (WORKDEPT) 'A00'.

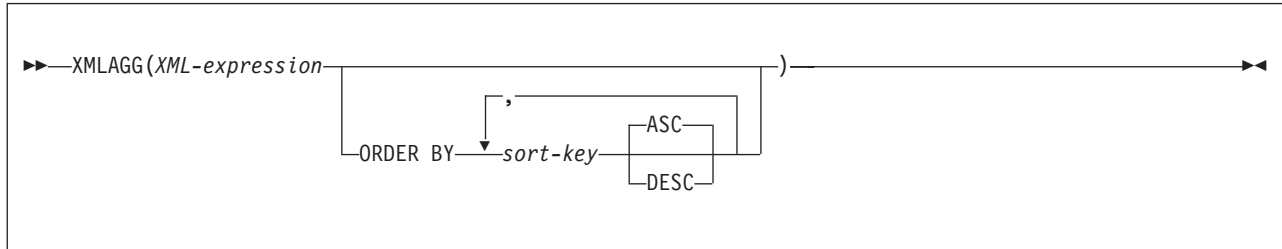
```
SELECT VARIANCE(SALARY)
  INTO :VARNCE
  FROM DSN8910.EMP
 WHERE WORKDEPT = 'A00';
```

The result in *VARNCE* is set to a double precision-floating point number with an approximate value of '94915000.00'.

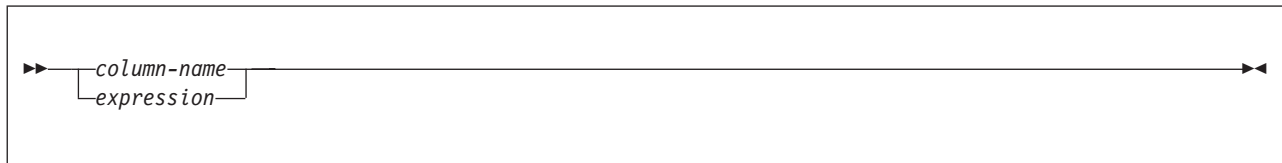
If *VARIANCE_SAMP* had been specified to find the sample variance of the salaries, the result in *VARNCE* would be set to a double precision-floating point number with an approximate value of '94915000.00'.

XMLAGG

The XMLAGG function returns an XML sequence that contains an item for each non-null value in a set of XML values.



sort-key



The schema is SYSIBM.

XML-expression

An expression that returns an XML value.

Unlike the arguments for other aggregate functions, a scalar fullselect is allowed in *XML-expression*.

ORDER BY

Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is not specified, or if the ORDER BY clause cannot differentiate the order of the sort key value, the rows in the same grouping set are arbitrarily ordered.

sort-key

Specifies a sort key value that is either a column name or an expression.

The data type of the column or expression must not be a LOB or an XML value. A character string expression cannot have a length greater than 4000 bytes. If the sort key value is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but is simply a constant, which implies no sort key.

The ordering is based on the values of the sort keys, which might or might not be used in *XML-expression*.

If the sort key value is a character string that uses an encoding scheme other than Unicode, the ordering might be different. For example, a column PRODCODE uses EBCDIC. For two values, ('P001' and 'PA01'), relationship 'P001' > 'PA01' is true in EBCDIC, whereas 'P001' < 'PA01' is true in UTF-8. If the same sort key values are used in *XML-expression*, use the CAST specification to convert the sort key to Unicode to keep the ordering of XML values consistent with that of the sort key.

The function is applied to the set of values derived from the argument values by the elimination of null values.

The result can be null; if all *XML-expression* arguments are null. If the function is applied to an empty set, the result is the null value. Otherwise, the result is an XML sequence that contains an item for each value in the set.

Example: Group employees by their department, generate a 'Department' element for each department with its name as the attribute, nest all the 'emp' elements for employees in each department, and order the 'emp' elements by 'lname.'

```
SELECT XMLSERIALIZE(XMLDOCUMENT
  ( XMLELEMENT
    ( NAME "Department",
      XMLATTRIBUTES ( e.dept AS "name" ),
      XMLAGG ( XMLELEMENT ( NAME "emp", e.lname)
        ORDER BY e.lname)
      ) ) AS "dept_list"
    AS CLOB(1M))
  FROM employees e
  GROUP BY dept;
```

The result of the query would look similar to the following result:

```
dept_list
-----
<Department name="Accounting">
  <emp>SMITH</emp>
  <emp>Yates</emp>
</Department>
<Department name="Shipping">
  <emp>Martin</emp>
  <emp>Oppenheimer</emp>
</Department>
-----
```

Scalar functions

A scalar function can be used wherever an expression can be used. The restrictions on the use of aggregate functions do not apply to scalar functions, because a scalar function is applied to single set of parameter values rather than to sets of values. The argument of a scalar function can be a function. However, the restrictions that apply to the use of expressions and aggregate functions also apply when an expression or aggregate function is used within a scalar function. For example, the argument of a scalar function can be a aggregate function only if a aggregate function is allowed in the context in which the scalar function is used.

If the argument of a scalar function is a string from a column with a field procedure, the function applies to the decoded form of the value and the result of the function does not inherit the field procedure.

Example: The following SELECT statement calls for the employee number, last name, and age of each employee in department D11 in the sample table DSN8910.EMP. To obtain the ages, the scalar function YEAR is applied to the expression:

```
CURRENT DATE - BIRTHDATE
```

in each row of DSN8910.EMP for which the employee represented is in department D11:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BIRTHDATE)
FROM DSN8910.EMP
WHERE WORKDEPT = 'D11';
```

ABS

The ABS function returns the absolute value of a number.

►►—ABS(*numeric-expression*)—◄◄

The schema is SYSIBM.

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

The result of the function has the same data type and length attribute as the argument. The result can be null. If the argument is null, the result is the null value.

ABSVAL can be specified as a synonym for ABS. DB2 supports this keyword to provide compatibility with previous releases.

Example: Assume that host variable PROFIT is a large integer with a value of -50000. The following statement returns a large integer with a value of 50000.

```
SELECT ABS(:PROFIT)
FROM SYSIBM.SYSDUMMY1;
```

ACOS

The ACOS function returns the arc cosine of the argument as an angle, expressed in radians. The ACOS and COS functions are inverse operations.

►►—ACOS(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type except for DECFLOAT. The value must be greater than or equal to -1 and
| less than or equal to 1. If the argument is not a double precision floating-point
| number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable ACOSINE is DECIMAL(10,9) with a value of 0.070737202. The following statement:

```
SELECT ACOS(:ACOSINE)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 1.49.

ADD_MONTHS

The ADD_MONTHS function returns a date that represents *expression* plus a specified number of months.

►►—ADD_MONTHS(*expression*,*numeric-expression*)—◀◀

The schema is SYSIBM.

expression

An expression that specifies the starting date. *expression* must return a value that is a date, timestamp, or a valid string representation of a date or timestamp. A string representation is a value that is a built-in character string data type or graphic string data type, that is not a LOB, and that has an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

numeric-expression

An expression that returns a value of any built-in numeric data type. The integer portion of *numeric-expression* specifies the number of months to add to the starting date specified by *expression*.

The result of the function is a DATE.

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

If *expression* is the last day of the month or if the resulting month has fewer days than the day component of *expression*, the result is the last day of the resulting month. Otherwise, the result has the same day component as *expression*.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example 1: Assume today is January 31, 2007. Set the host variable ADD_MONTH with the last day of January plus 1 month.

```
SET :ADD_MONTH = ADD_MONTHS(LAST_DAY(CURRENT_DATE), 1);
```

The host variable ADD_MONTH is set with the value representing the end of February, 2007-02-28.

Example 2: Assume DATE is a host variable with the value July 27, 1965. Set the host variable ADD_MONTH with the value of that day plus 3 months.

```
SET :ADD_MONTH = ADD_MONTHS(:DATE,3);
```

The host variable ADD_MONTH is set with the value representing the day plus 3 months, 1965-10-27.

Example 3: It is possible to achieve similar results with the ADD_MONTHS function and date arithmetic. The following examples demonstrate the similarities and contrasts.

```
SET :DATEHV = DATE('2008-2-28') + 4 MONTHS;
```



```
SET :DATEHV = ADD_MONTHS('2008-2-28', 4);
```

In both cases, the host variable DATEHV is set with the value '2008-06-29'.

Now consider the same examples but with the date '2008-2-29' as the argument.

```
SET :DATEHV = DATE('2008-2-29') + 4 MONTHS;
```

The host variable DATEHV is set with the value '2008-06-29'.

```
SET :DATEHV = ADD_MONTHS('2008-2-29', 4);
```

The host variable DATEHV is set with the value '2008-06-30'.

In this case, the ADD_MONTHS function returns the last day of the month, which is June 30, 2008, instead of June 29, 2008. The reason is that February 29 is the last day of the month. So, the ADD_MONTHS function returns the last day of June.

ASCII

The ASCII function returns the leftmost character of the argument as an integer.

►►—ASCII(*string-expression*)—◄◄

The schema is SYSIBM.

The argument can be any built-in character or graphic string data type, except for CLOB or DBCLOB. If the argument is an EBCDIC, Unicode, or graphic string, it is first converted to an SBCS ASCII character string (CCSID 367)¹⁹ before the function is executed.

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

Example: The following statement returns the ASCII value for the character 'A':

```
SET :hv = ASCII('A');
```

The host variable, :hv, is set to an integer with the value 65.

19. If the conversion does not exist, the ASCII function will return an error, or a substitution character might be returned.

ASCII_CHR

The ASCII_CHR function returns the character that has the ASCII code value that is specified by the argument.



►—ASCII_CHR(*expression*)—◄

The schema is SYSIBM.

expression

An expression that returns a built-in data type of BIGINT, INTEGER, or SMALLINT.

The result of the function is a fixed length character string encoded in the SBCS ASCII CCSID (regardless of the setting of the MIXED option in DSNHDECP). The length of the result is 1. If the value of *expression* is not in the range of 0 to 255, (0 to 127 if the SBCS ASCII CCSID for this system is CCSID 367) the null value is returned.

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

CHR can be specified as a synonym for ASCII_CHR.

Example: Set :hv with the Euro symbol "€" in CCSID 923:

```
SET :hv = ASCII_CHR(164); -- x'A4'
```

Set :hv with the Euro symbol "€" in CCSID 1252:

```
SET :hv = ASCII_CHR(128); -- x'80'
```

In both cases, the "€" is assigned to :hv, but because the Euro symbol is located at different code points for the two CCSIDs, the input value is different.

ASCII_STR

The ASCII_STR function returns an ASCII version of the string in the system ASCII CCSID. The system ASCII CCSID is the SBCS ASCII CCSID on a MIXED=NO system or the MIXED ASCII CCSID on a MIXED=YES system.

►►—ASCII_STR(*string-expression*)—►►

The schema is SYSIBM.

string-expression

An expression that returns a value of a built-in character or graphic string. If the string is a character string, it cannot be bit data. *string-expression* must be an ASCII, EBCDIC, or Unicode string. ASCII_STR returns an ASCII version of the string. Non-ASCII characters are converted to the form \xxxx, where xxxx represents a UTF-16 code unit.

The length attribute of the result will be $\text{MIN}((5 \times n), 32704)$. Where n is the result of applying the formulas in Table 35 on page 126 based on input and output data types.

The result of the function is a varying-length character string in the system ASCII CCSID. If the actual length of the result string exceeds the maximum for the return type, an error occurs.

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

ASCIISTR can be specified as a synonym for ASCII_STR.

Example: The following example returns the ASCII string equivalent of the text string "Hi my name is А н д р е й (Andrei)"

```
SET :HV1 = ASCII_STR('Hi, my name is А н д р е й (Andrei)');
```

:HV1 is assigned the value "Hi, my name is \0410\043D\0434\0440\0435\0439 (Andrei)"

ASIN

The ASIN function returns the arc sine of the argument as an angle, expressed in radians. The ASIN and SIN functions are inverse operations.

►►—ASIN(*numeric-expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type except for DECFLOAT. The value must be greater than or equal to -1 and less than or equal to 1. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

The result is greater than or equal to $-\pi/2$ and less than or equal to $\pi/2$.

Example: Assume that host variable ASINE is DECIMAL(10,9) with a value of 0.997494987. The following statement:

```
SELECT ASIN(:ASINE)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 1.50.

ATAN

The ATAN function returns the arc tangent of the argument as an angle, expressed in radians. The ATAN and TAN functions are inverse operations.

►►—ATAN(*numeric-expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. The value must be greater than or equal to -1 and less than or equal to 1. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

The result is greater than or equal to $-\pi/2$ and less than or equal to $\pi/2$.

Example: Assume that host variable ATANGENT is DECIMAL(10,9) with a value of 14.10141995. The following statement returns a double precision floating-point number with an approximate value of 1.50:

```
SELECT ATAN(:ATANGENT)
FROM SYSIBM.SYSDUMMY1;
```

ATANH

The ATANH function returns the hyperbolic arc tangent of a number, expressed in radians. The ATANH and TANH functions are inverse operations.

►►—ATANH(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. The value must be greater than -1 and less than
| 1. If the argument is not a double precision floating-point number, it is converted
| to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HATAN is DECIMAL(10,9) with a value of 0.905148254. The following statement returns a double precision floating-point number with an approximate value of 1.50:

```
SELECT ATANH(:HATAN)
FROM SYSIBM.SYSDUMMY1;
```

ATAN2

The ATAN2 function returns the arc tangent of x and y coordinates as an angle, expressed in radians.

►►—ATAN2(*numeric-expression-1*,*numeric-expression-2*)—◄◄

The schema is SYSIBM.

The first and second arguments specify the x and y coordinates, respectively.

Each argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. Both arguments must not be 0. Any argument that is not a double precision floating-point number is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if any argument is null, the result is the null value.

Example: Assume that host variables HATAN2A and HATAN2B are DOUBLE host variables with values of 1 and 2, respectively. The following statement returns a double precision floating-point number with an approximate value of 1.1071487:

```
SELECT ATAN2(:HATAN2A,:HATAN2B)
FROM SYSIBM.SYSDUMMY1;
```


BIGINT

The BIGINT function returns a big integer representation of either a number or a character or graphic string representation of a number.

Numeric to Big Integer:

►►—BIGINT(*numeric-expression*)—◄◄

String to Big Integer:

►►—BIGINT(*string-expression*)—◄◄

The schema is SYSIBM.

Numeric to Big Integer

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of big integers, an error is returned. The fractional part of the argument is truncated.

String to Big Integer

string-expression

An expression that returns a value of a character or graphic string (except a CLOB and DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from CAST(*string-expression* AS BIGINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer constant. If the whole part of the argument is not within the range of big integers, an error is returned. Any fractional part of the argument is truncated.

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

To increase the portability of applications, use the CAST specification.

Example 1: The following function returns the number 12345 (a BIGINT) for the number 12345.6:

```
SELECT BIGINT(12345.6)
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a BIGINT value of 123456789012 for the number 00123456789012.

```
|      SELECT BIGINT('00123456789012')  
|      FROM SYSIBM.SYSDUMMY1;
```

| **Related reference**

| “CAST specification” on page 202

BINARY

The BINARY function returns a BINARY (fixed-length binary string) representation of a string of any type or of a row ID type.

►► BINARY(*string-expression* [, *integer*]) ►►

The schema is SYSIBM.

string-expression

An expression that returns a value that is a built-in character string, graphic string, binary string, or a row ID type.

integer

An integer value that specifies the length attribute of the resulting binary string. The value must be an integer between 1 and 255 inclusive.

If *integer* is not specified:

- If the *string-expression* is the empty string constant, an error occurs
- Otherwise, the length attribute of the result is the same as the length attribute of *string-expression*, except when the input is graphic data. In this case, the length attribute of the result is twice the length of *string-expression*.

The result of the function is a fixed-length binary string. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The actual length is the same as the length attribute of the result. If the length of the *string-expression* is less than the length of the result, the result is padded with hexadecimal zeroes up to the length of the result. If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed. A warning is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks, or the first input argument is a binary string and all the truncated bytes are hexadecimal zeroes.

Following examples assume EBCDIC encoding of the input string constants.

Example 1: The following function returns a fixed-length binary string with a length attribute 1 and a value BX'00'.

```
SELECT BINARY(' ',1)
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a fixed-length binary string with a length attribute 5 and a value BX'D2C2C80000'

```
SELECT BINARY('KBH',5)
FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'D2C2C8'

```
|      SELECT BINARY('KBH')
|      FROM SYSIBM.SYSDUMMY1;
```

Example 4: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'D2C2C8'

```
|      SELECT BINARY('KBH ',3)
|      FROM SYSIBM.SYSDUMMY1;
```

Example 5: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'D2C2C8', a warning is also returned.

```
|      SELECT BINARY('KBH 93',3)
|      FROM SYSIBM.SYSDUMMY1;
```

Example 6: The following function returns a fixed-length binary string with a length attribute 3 and a value BX'C1C2C3', a warning is also returned.

```
|      SELECT BINARY(BINARY('ABC',5),3)
|      FROM SYSIBM.SYSDUMMY1;
```

```
|
```

BLOB

The BLOB function returns a BLOB representation of a string of any type or of a row ID type.

A diagram showing the syntax of the BLOB function. It consists of the text 'BLOB(string-expression' followed by a bracketed section containing ', -integer'. This is followed by a closing parenthesis ')'. A long horizontal line with arrowheads at both ends extends from the closing parenthesis to the right, indicating the return type or result.

The schema is SYSIBM.

string-expression

An expression that returns a value that is a built-in character string, graphic string, binary string, or a row ID type.

integer

An integer value that specifies the length attribute of the resulting binary string. The value must be an integer between 1 and the maximum length of a BLOB.

Do not specify *integer* if *string-expression* is a row ID type.

If you do not specify *integer* and *string-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *string-expression*, except when the input is graphic data. In this case, the length attribute of the result is twice the length of *expression*.

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

The actual length of the result is the minimum of the length attribute of the result and the actual length of *string-expression* (or twice the length of *string-expression* when the input is graphic data). If the length of *string-expression* is greater than the length attribute of the result, truncation is performed. A warning is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks.

Example 1: The following function returns a BLOB for the string 'This is a BLOB'.

```
SELECT BLOB('This is a BLOB')
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a BLOB for the large object that is identified by locator myclob_locator.

```
SELECT BLOB(:myclob_locator)
FROM SYSIBM.SYSDUMMY1;
```

Example 3: Assume that a table has a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME. Locate any maps that contain the string 'Engles Island' and return a single binary string with the map name concatenated in front of the actual map.

```
SELECT BLOB(MAP_NAME || ' : ') || TOPOGRAPHIC_MAP
FROM ONTARIO_SERIES_4
WHERE TOPOGRAPHIC_MAP LIKE BLOB('%Engles Island%')
```

CCSID_ENCODING

The CCSID_ENCODING function returns a string value that indicates the encoding scheme of a CCSID that is specified by the argument.

►►—CCSID_ENCODING(*expression*)—◄◄

The schema is SYSIBM.

expression

expression must be an expression that returns a value of a built-in numeric, character, or graphic string data type that is not a LOB. A character string must not have a length attribute greater than 255, and a graphic string must not have a length attribute greater than 127. If *expression* is a character or graphic string, the string must contain a valid string representation of a number. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a numeric constant.

The function returns a value of ASCII, EBCDIC, UNICODE, or UNKNOWN depending on the CCSID specified by *expression*.

The result of the function is a fixed-length character string of length 8, which is padded on the right if necessary.

If the argument can be null, the result can be null. If the argument is null, the result is the null value. The CCSID of the result is determined from the context in which the function was invoked. For more information, refer to “Determining the encoding scheme and CCSID of a string” on page 43.

Example 1: The following function returns a CCSID with a value for EBCDIC data.

```
SELECT CCSID_ENCODING(37) AS CCSID
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a CCSID with a value for ASCII data.

```
SELECT CCSID_ENCODING(850) AS CCSID
FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following function returns a CCSID with a value for Unicode data.

```
SELECT CCSID_ENCODING(1208) AS CCSID
FROM SYSIBM.SYSDUMMY1;
```

Example 4: The following function returns a CCSID with a value of UNKNOWN.

```
SELECT CCSID_ENCODING(1) AS CCSID
FROM SYSIBM.SYSDUMMY1;
```

Example 5: The following function returns a CCSID with a value for EBCDIC data. The input data is a character string.

```
SELECT CCSID_ENCODING('37') AS CCSID
FROM SYSIBM.SYSDUMMY1;
```

CEILING

The CEILING function returns the smallest integer value that is greater than or equal to the argument.

►►—CEILING—(*numeric-expression*)—◄◄

The schema is SYSIBM.

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

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) results in DECIMAL(5,0). The result can be null. If the argument is null, the result is the null value.

CEIL can be specified as a synonym for CEILING.

Example 1: The following statement shows the use of CEILING on positive and negative values:

```
SELECT CEILING(3.5), CEILING(3.1), CEILING(-3.1), CEILING(-3.5)
FROM FROM SYSIBM.SYSDUMMY1;
```

This example returns: 04., 04., -03., -03.

Example 2: Using sample table DSN8910.EMP, find the highest monthly salary for all the employees. Round the result up to the next integer. The SALARY column has a decimal data type.

```
SELECT CEILING(MAX(SALARY)/12)
FROM DSN8910.EMP;
```

This example returns 04396. because the highest paid employee is Christine Haas who earns \$52750.00 per year. Her average monthly salary before applying the CEILING function is 4395.83.

CHAR

The CHAR function returns a fixed-length character string representation of the argument.

Integer to Character:

►►—CHAR(*integer-expression*)—►►

Decimal to Character:

►►—CHAR(*decimal-expression* [, —*decimal-character*])—►►

Floating-Point to Character:

►►—CHAR(*floating-point-expression*)—►►

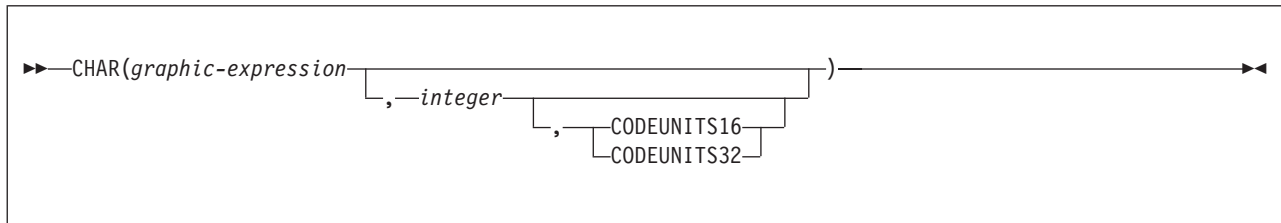
Decimal floating-point to Character:

►►—CHAR(*decimal-floating-point-expression*)—►►

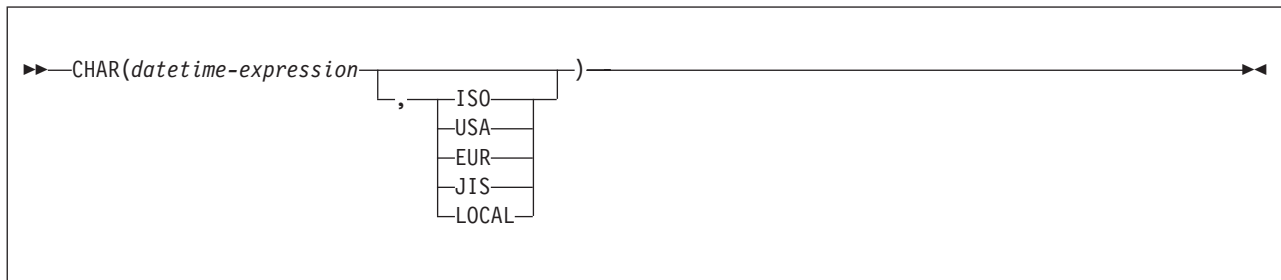
Character to Character:

►►—CHAR(*character-expression* [, —*integer* [, —*CODEUNITS16* [—*CODEUNITS32* [—*OCTETS*]]])—►►

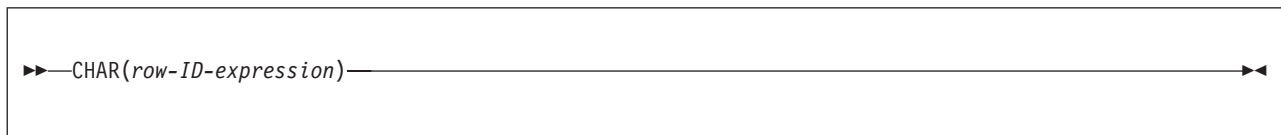
Graphic to Character:



Datetime to Character:



Row ID to Character:



The schema is SYSIBM.

The CHAR function returns a fixed-length character string representation of one of the following values:

- An integer number if the first argument is a small, large, or big integer
- A decimal number if the first argument is a decimal number
- A floating-point number if the first argument is a single or double precision floating-point number
- A decimal floating-point number if the first argument is a decimal floating-point number
- A character string value if the first argument is any type of string
- A datetime value if the first argument is a date, time, or timestamp
- A row ID value if the first argument is a row ID

The result of the function is a fixed-length character string (CHAR).

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Integer to Character

integer-expression

An expression that returns a value that is a built-in integer data type (SMALLINT, INTEGER, or BIGINT).

The result is the fixed-length character string representation of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument. If the argument is negative, the result has a preceding minus sign. The result is left justified, and its length depends on whether the argument is a small or large integer:

- For a small integer, the length of the result is 6. If the number of characters in the result is less than 6, the result is padded on the right with blanks to a length of 6.
- For a large integer, the length of the result is 11; if the number of characters in the result is less than 11, the result is padded on the right with blanks to a length of 11.

A positive value always includes one trailing blank.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Decimal to Character

decimal-expression

An expression that returns a value that is a built-in decimal data type. To specify a different precision and scale for the value of the expression, apply the DECIMAL function before applying the CHAR function.

decimal-character

Specifies the single-byte character constant (CHAR or VARCHAR) that is used to delimit the decimal digits in the result character string. The character must not be a digit, a plus sign (+), a minus sign (-), or a blank. The default is the period (.) or comma (,). For information on what factors govern the choice, see “Decimal point representation” on page 251.

The result is the fixed-length character string representation of the argument. The result includes a decimal character and up to p digits, where p is the precision of the *decimal-expression* with the preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned. If the scale of *decimal-expression* is zero, the decimal character is not returned. If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with blanks.

The leading blank is not returned for `CAST(decimal-expression AS CHAR(n))`.

The length of the result is $2 + p$, where p is the precision of the *decimal-expression*.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Floating-Point to Character

floating-point-expression

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

The result is the fixed-length character string representation of the argument in the form of an SQL floating-point constant. The length of the result is 24 bytes.

If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit. If the value of the argument is zero, the result is 0E0. Otherwise, the result includes the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit, other than zero, followed by a period and a sequence of digits.

If the number of characters in the result is less than 24, the result is padded on the right with blanks to length of 24.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Decimal floating-point to Character

decimal-floating-point-expression

An expression that returns a value that is a built-in decimal floating-point data type (DECFLOAT).

The result is the fixed-length character string representation of the argument in the form of an SQL decimal floating-point constant. The length of the result is 42 bytes. If the number of characters in the result is less than 42, the result is padded on the right with blanks to length of 42.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Character to Character

character-expression

An expression that returns a value of a built-in character string.

integer

The length attribute for the resulting fixed-length character string. The value must be an integer constant between 1 and 255.

If the length is not specified, the length attribute of the result is the minimum of 255 and the length attribute of *character-expression*. If *character-expression* is an empty string constant, an error occurs.

If **CODEUNITS16** or **CODEUNITS32** is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the unit that is used to express *integer*. If *character-expression* is a character string that is defined as bit data, **CODEUNITS16** and **CODEUNITS32** cannot be specified.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *integer* is expressed in terms of bytes.

For more information about **CODEUNITS16**, **CODEUNITS32**, and **OCTETS**, see “String unit specifications” on page 76.

The actual length is the same as the length attribute of the result. If the length of *character-expression* is less than the length attribute of the result, the result is padded with blanks to the length of the result. If the length of *character-expression* is greater than the length attribute of the result, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

If *character-expression* is bit data, the result is bit data. Otherwise, the CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to Character

graphic-expression

An expression that returns a value of a built-in graphic string.

integer

The length attribute for the resulting fixed-length character string. The value must be an integer constant between 1 and 255.

If the length is not specified, the length attribute of the result is the minimum of 255 and the length attribute of *graphic-expression*. The length attribute of *graphic-expression* is $(3 * \text{length}(\text{graphic-expression}))$. If *graphic-expression* is an empty string constant, an error occurs.

If **CODEUNITS16** or **CODEUNITS32** is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about **CODEUNITS16** and **CODEUNITS32**, see “String unit specifications” on page 76.

The actual length is the same as the length attribute of the result. If the length of *graphic-expression* is less than the length attribute of the result, the result is padded with blanks to the length of the result. If the length of *graphic-expression* is greater than the length attribute of the result, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

The CCSID of the result is the character mixed CCSID that corresponds to the graphic CCSID of *graphic-expression*.

Datetime to Character

datetime-expression

An expression that is one of the following built-in data types:

date The result is the character string representation of the date in the format that is specified by the second argument. If the second argument is omitted, the DATE precompiler option, if one is provided, otherwise field DATE FORMAT on installation panel DSNTIP4 specifies the format. If the format is **LOCAL**, field LOCAL DATE LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length of the result is 10.

LOCAL denotes the local format at the DB2 subsystem that executes the SQL statement. If **LOCAL** is used for the format, a date exit routine must be installed at that DB2 subsystem.

An error occurs if the second argument is specified and is not a valid value.

time The result is the character string representation of the time in the format that is specified by the second argument. If the second argument is omitted, the TIME precompiler option, if one is provided, otherwise field TIME FORMAT on installation panel DSNTIP4 specifies the format. If the format is **LOCAL**, the field LOCAL TIME LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length of the result is 8.

LOCAL denotes the local format at the DB2 subsystem that executes the SQL statement. If **LOCAL** is used for the format, a time exit routine must be installed at that DB2 subsystem.

An error occurs if the second argument is specified and is not a valid value.

timestamp

The result is the character string representation of the timestamp. The length of the result is 26. The second argument must not be specified.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

ISO, EUR, USA, JIS, or LOCAL

Specifies the date or time format of the resulting character string. For more information, see “String representations of datetime values” on page 89.

Row ID to Character

row-ID-expression

An expression that returns a value that is a built-in row ID data type.

The result is the fixed-length character string representation of the argument. The result is bit data.

The length of the result is 40. If the length of *row-ID-expression* is less than 40, the result is padded on the right with hexadecimal zeros to a length of 40.

Recommendation: To increase the portability of applications, use the CAST specification when the first argument is numeric, or the first argument is a string and the length argument is specified. For more information, see “CAST specification” on page 202.

Example 1: HIREDATE is a DATE column in sample table DSN8910.EMP. When it represents the date 15 December 1976 (as it does for employee 140), the following example returns the string value '12/15/1976' in character string variable DATESTRING:

```
EXEC SQL SELECT CHAR(HIREDATE, USA)
        INTO :DATESTRING
        FROM DSN8910.EMP
        WHERE EMPNO = '000140';
```

Example 2: Host variable HOUR has a data type of DECIMAL(6,0) and contains a value of 50000. Interpreted as a time duration, this value is 5 hours. Assume that STARTING is a TIME column in some table. Then, when STARTING represents 17 hours, 30 minutes, and 12 seconds after midnight, the following example returns the value '10:30 PM':

```
CHAR(STARTING+:HOURS, USA)
```

Example 3: Assume that RECEIVED is defined as a TIMESTAMP column in table TABLEY. When the value of the date portion of RECEIVED represents the date 10 March 1997 and the time portion represents 6 hours and 15 seconds after midnight, the following example returns the string value '1997-03-10-06.00.15.000000':

```
SELECT CHAR(RECEIVED)
        FROM TABLEY
        WHERE INTCOL = 1234;
```

Example 4: For sample table DSN8910.EMP, the following SQL statement sets the host variable AVERAGE, which is defined as CHAR(33), to the character string representation of the average employee salary.

```
EXEC SQL SELECT CHAR(AVG(SALARY))
        INTO :AVERAGE
        FROM DSN8910.EMP;
```

With DEC31, the result of AVG applied to a decimal number is a decimal number with a precision of 31 digits. The only host languages in which such a large decimal variable can be defined are Assembler and C. For host languages that do not support such large decimal numbers, use the method shown in this example.

Example 5: For the rows in sample table DSN8910.EMP, return the values in column LASTNAME, which is defined as VARCHAR(15), as a fixed-length character string and limit the length of the results to 10 characters.

```
SELECT CHAR(LASTNAME,10)
        FROM DSN8910.EMP;
```

For rows that have a LASTNAME with a length greater than 10 characters (excluding trailing blanks), a warning that the value is truncated is returned.

Example 6: FIRSTNAME is a VARCHAR(12) column in a Unicode table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has the values shown under 'Function', the results are shown under 'Returns':

Function ...	Returns ...
CHAR(FIRSTNAME,3,CODEUNITS32)	'Jür' -- x'4AC3BC7220202020202020'
CHAR(FIRSTNAME,3,CODEUNITS16)	'Jür' -- x'4AC3BC722020202020'
CHAR(FIRSTNAME,3,OCTETS)	'Jü' -- x'4AC3BC'

Example 7: For the rows in sample table DSN8910.EMP, return the values in column EDLEVEL, which is defined as SMALLINT, as a fixed-length character string.

```
SELECT CHAR(EDLEVEL)
FROM DSN8910.EMP;
```

An EDLEVEL of 18 is returned as CHAR(6) value '18' (18 followed by four blanks).

Example 8: In sample table DSN8910.EMP, the SALARY column is defined as DECIMAL(9,2). For those employees who have a salary of 52750.00, return the hire date and the salary, using a comma as the decimal character in the salary (52750,00).

```
SELECT HIREDATE, CHAR(SALARY, ',')
FROM DSN8910.EMP
WHERE SALARY = 52750.00;
```

The salary is returned as the string value '52750,00'.

Example 9: Repeat the scenario in Example 8 except subtract the SALARY column from 60000.00 and return the salary with the default decimal character.

```
SELECT HIREDATE, CHAR (60000.00 - SALARY)
FROM DSN8910.EMP
WHERE SALARY = 52750.00;
```

The salary is returned as the string value '7250.00'.

Example 10: Assume that host variable SEASONS_TICKETS is defined as INTEGER and has a value of 10000. Use the DECIMAL and CHAR functions to change the value into the character string '10000.00'.

```
SELECT CHAR(DECIMAL(:SEASONS_TICKETS,7,2))
FROM SYSIBM.SYSDUMMY1;
```

Example 11: Assume that columns COL1 and COL2 in table T1 are both defined as REAL and that T1 contains a single row with the values 7.1E+1 and 7.2E+2 for the two columns. Add the two columns and represent the result as a character string.

```
SELECT CHAR(COL1 + COL2)
FROM T1;
```

The result is the character value '1.43E2'.

CHARACTER_LENGTH

The CHARACTER_LENGTH function returns the length of the first argument in the specified string unit.

Character string:

►► CHARACTER_LENGTH (—*character-expression*—, CODEUNITS16
CODEUNITS32
OCTETS) ►►

Graphic string:

►► CHARACTER_LENGTH (—*graphic-expression*—, CODEUNITS16
CODEUNITS32) ►►

The schema is SYSIBM.

Character string:

character-expression

An expression that returns a value of a built-in character string.
character-expression cannot be bit data.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the unit that is used to express the length of the result.

CODEUNITS16

Specifies that the result is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that the result is expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies the result is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

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 result is the length of *character-expression* expressed in the number of string units that were specified. The length of fixed-length strings includes trailing blanks. The length of varying-length strings is the actual length and not the maximum length.

Graphic string:

graphic-expression

An expression that returns a value of a built-in graphic string.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express the length of the result.

CODEUNITS16

Specifies that the result is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that the result is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32 see “String unit specifications” on page 76.

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 result is the length of *graphic-expression* expressed in the number of string units that were specified. The length of fixed-length strings includes trailing blanks. The length of varying-length strings is the actual length and not the maximum length.

Example: Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'. The following two queries return the value 6:

```
SELECT CHARACTER_LENGTH(NAME, CODEUNITS32)
FROM T1 WHERE NAME = 'Jürgen';
SELECT CHARACTER_LENGTH(NAME, CODEUNITS16)
FROM T1 WHERE NAME = 'Jürgen';
```

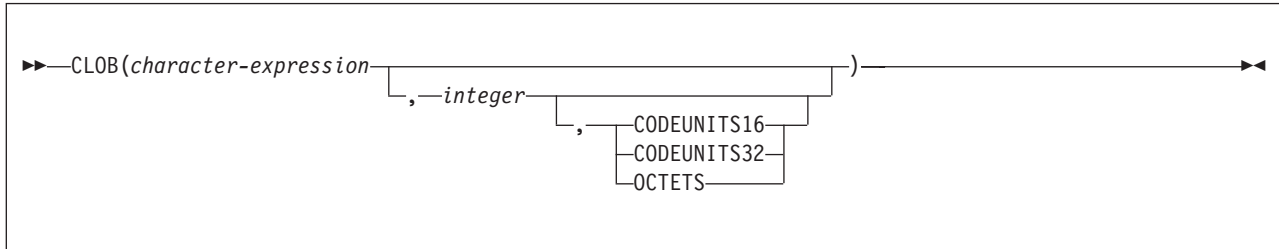
The following two queries return the value 7:

```
SELECT CHARACTER_LENGTH(NAME, OCTETS)
FROM T1 WHERE NAME = 'Jürgen';
SELECT LENGTH(NAME)
FROM T1 WHERE NAME = 'Jürgen';
```

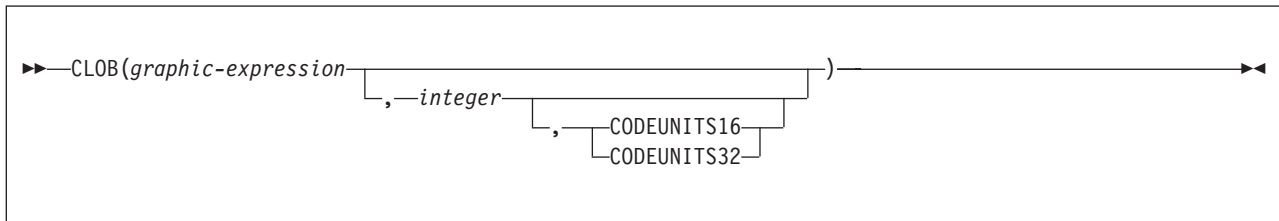
CLOB

The CLOB function returns a CLOB representation of a string.

Character to CLOB:



Graphic to CLOB:



The schema is SYSIBM.

Character to CLOB

character-expression

An expression that returns a value of a character string. If *character-expression* is bit data, an error occurs.

integer

An integer constant that specifies the length attribute of the resulting CLOB data type. The value must be between 1 and the maximum length of a CLOB, expressed in the units that are either implicitly or explicitly specified.

If you do not specify *integer* and *character-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *character-expression*.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information on how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the unit that is used to express *integer*.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *integer* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

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

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to CLOB

graphic-expression

An expression that returns a value of a graphic string.

integer

An integer constant that specifies the length attribute of the resulting CLOB data type. The value must be between 1 and the maximum length of a CLOB, expressed in the units that are either implicitly or explicitly specified.

If you do not specify *integer* and *graphic-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is $(3 * \text{length}(\text{graphic-expression}))$.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information on how to calculate the length attribute of the result string.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

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

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of *graphic-expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

The CCSID of the result is the character mixed CCSID that corresponds to the graphic CCSID of *graphic-expression*.

Example 1: The following function returns a CLOB for the string 'This is a CLOB'.

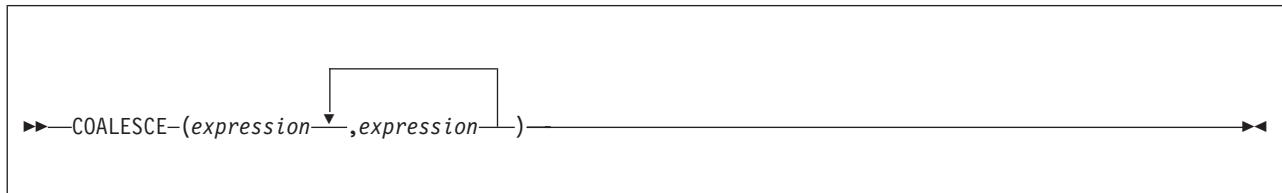
```
SELECT CLOB('This is a CLOB')
FROM SYSIBM.SYSDUMMY1;
```

Example 2: FIRSTNME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNME has this value:

Function ...	Returns ...
CLOB(FIRSTNME,3,CODEUNITS32)	'Jür' -- x'4AC3BC72'
CLOB(FIRSTNME,3,CODEUNITS16)	'Jür' -- x'4AC3BC72'
CLOB(FIRSTNME,3,OCTETS)	'Jü' -- x'4AC3BC'

COALESCE

The COALESCE function returns the value of the first nonnull expression.



The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 20 on page 102. The arguments can be of either a built-in or user-defined data type.²⁰

The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. The result can be null only if all arguments can be null. The result is null only if all arguments are null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined using the “Rules for result data types” on page 119. If the COALESCE function has more than two arguments, the rules are applied to the first two arguments to determine a candidate result type. The rules are then applied to that candidate result type and the third argument to determine another candidate result type. This process continues until all arguments are analyzed and the final result type is determined.

The COALESCE function can also handle a subset of the functions provided by CASE expressions. The result of using COALESCE(e1,e2) is the same as using the expression:

```
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END
```

VALUE can be specified as a synonym for COALESCE.

Example 1: Assume that SCORE1 and SCORE2 are SMALLINT columns in table GRADES, and that nulls are allowed in SCORE1 but not in SCORE2. Select all the rows in GRADES for which SCORE1 + SCORE2 > 100, assuming a value of 0 for SCORE1 when SCORE1 is null.

```
SELECT * FROM GRADES
WHERE COALESCE(SCORE1,0) + SCORE2 > 100;
```

Example 2: Assume that a table named DSN8910.EMP contains a DATE column named HIREDATE, and that nulls are allowed for this column. The following query selects all rows in DSN8910.EMP for which the date in HIREDATE is either unknown (null) or earlier than 1 January 1960.

```
SELECT * FROM DSN8910.EMP
WHERE COALESCE(HIREDATE,DATE('1959-12-31')) < '1960-01-01';
```

20. This function cannot be used as a source function when creating a user-defined function.

The predicate could also be coded as `COALESCE(HIREDATE, '1959-12-31')` because, for comparison purposes, a string representation of a date can be compared to a date.

Example 3: Assume that for the years 1993 and 1994 there is a table that records the sales results of each department. Each table, `S1993` and `S1994`, consists of a `DEPTNO` column and a `SALES` column, neither of which can be null. The following query provides the sales information for both years.

```
SELECT COALESCE(S1993.DEPTNO,S1994.DEPTNO) AS DEPT, S1993.SALES, S1994.SALES
FROM S1993 FULL JOIN S1994 ON S1993.DEPTNO = S1994.DEPTNO
ORDER BY DEPT;
```

The full outer join ensures that the results include all departments, regardless of whether they had sales or existed in both years. The `COALESCE` function allows the two join columns to be combined into a single column, which enables the results to be ordered.

COLLATION_KEY

The COLLATION_KEY function returns a varying-length binary string that represents the collation key of the argument in the specified collation.

```
►►—COLLATION_KEY(string-expression,collation-name——, —integer)—►►
```

The schema is SYSIBM.

The result of COLLATION_KEY on one string can be compared in binary form with the result of COLLATION_KEY on another string to determine their order within the specified *collation-name*. For the comparison to be meaningful, the results of the COLLATION_KEY must be from the same *collation-name*.

string-expression

An expression that returns a character or graphic string that is not a LOB for which the collation key is to be determined. If *string-expression* is a character string, it must not be FOR BIT DATA. If *string-expression* is not in Unicode UTF-16 (CCSID 1200), it is converted to Unicode UTF-16 before the corresponding collation key is obtained. The length of *string-expression* must not exceed 32704 bytes of the UTF-16 representation.

collation-name

A string constant or a string host variable that is not a binary string, CLOB, or DBCLOB. *collation-name* specifies the collation to use when determining the collation key. If *collation-name* is not an EBCDIC value, it is converted to EBCDIC. The length of *collation-name* must be between 1 and 255 bytes of the EBCDIC representation. The value of *collation-name* is not case sensitive and must be a left justified, valid "short path" collation setting for the parameter CUNBOPRM_Collation_Keyword in area CUN4BOPR. For detailed information about the "short path" setting in the parameter CUNBOPRM_Collation_Keyword, see *z/OS Support for Unicode: Using Conversion Services*.

The value of the host variable must not be null. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value. *collation-name* must be left justified within the host variable. It must also be padded on the right with blanks if the length is less than that of the host variable and the host variable is a fixed length CHAR or GRAPHIC data type.

collation-name is in the form of CUN4BOPR_Collation_Keyword specification.

The following table lists some supported values:

Table 54. Collation Keywords Reference

Attribute name	Key	Possible values
Locale	L.R.V	<locale>
Strength	S	1, 2, 3, 4, I, D
Case_Level	K	X, O, D
Case_First	C	X, L, U, D

Table 54. Collation Keywords Reference (continued)

Attribute name	Key	Possible values
Alternate	A	N, S, D
Variable_Top	T	<hex digits>
Normalization	N	X, O, D
French	F	X, O, D
Hinayana	H	X, O, D

The following table describes the abbreviations for the collation keywords:

Abbreviation	Definition
D	default
O	on
X	off
1	primary
2	secondary
3	tertiary
4	quaternary
I	identical
S	shifted
N	non-ignorable
L	lower-first
U	upper-first

The following examples show keywords using the above specifications:

'UCA400R1_AS_LSV_S3_CU'

UCA version 4.0.1; ignore spaces, punctuation and symbols; use Swedish linguistic conventions; use case-first upper; compare case-sensitive.

'UCA400R1_AN_LSV_S3_CL_NO'

UCA version 4.0.1; do not ignore spaces, punctuation and symbols; use Swedish linguistic conventions; use case-first lower (or does not set it to mean the same, since lower is used in most locales as the default); normalization ON; compare case-sensitive.

integer

An integer value that specifies the length attribute of the result. If specified, the value must be an integer constant between 1 and 32704.

If the length is not specified, the length attribute of the result is determined as follows:

<i>string-expression</i>	Result length attribute
CHAR(<i>n</i>) or VARCHAR(<i>n</i>)	MIN (VARBINARY(12 <i>n</i>), 32704)
GRAPHIC(<i>n</i>) or VARGRAPHIC(<i>n</i>)	MIN (VARBINARY(12 <i>n</i>), 32704)

Regardless of whether the length is specified, the length of the collation key must be less than or equal to the length attribute of the result. The actual result length of the collation key is approximately six times of the length of *string-expression* where the length of *string-expression* is in Unicode byte representation. For certain *collation-name* such as UCA410_LKO_RKR (for Korean collation) the default length attribute of the result, 12*n*, might not be

large enough and an error will be returned. To avoid such an error, the length attribute of the result must be explicitly specified to a larger constant. For the proper length attribute of the result, see *z/OS Support for Unicode: Using Conversion Services* for information about target buffer length considerations for Collation Services.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The `COLLATION_KEY` function uses Unicode Collation Services in z/OS to return the collation key. Unicode Collation Services support two collation versions:

- UCA400R1. This Collation version support Unicode standard character suite 4.0.0 and use Normalization Service under 4.0.1 Unicode character suite.
- UCA410. This Collation version support Unicode standard character suite 4.1.0 and use Normalization Service under 4.1.0 Unicode character suite.

If Unicode Collation Services are not available when the `COLLATION_KEY` function is run, an error is returned.

Example 1: The following query orders the employees by their surnames using the default Unicode Collation Algorithm V4.0.1(UCA), ignoring spaces, punctuation, and symbols, using Swedish linguistic conventions, and not comparing case:

```
SELECT FIRSTNAME, LASTNAME
FROM DSN8910.EMP
ORDER BY COLLATION_KEY(LASTNAME, 'UCA400R1_AS_LSV_S2');
```

Example 2: The following query uses the `COLLATION_KEY` function on the `LASTNAME` column and the `SALES_PERSON` column to obtain the sort keys from the same collation name in order to do a culturally correct comparison. It finds the departments of employees in Quebec:

```
SELECT E.WORKDEPT
FROM EMPLOYEE AS E INNER JOIN SALES AS S
ON COLLATION_KEY(E.LASTNAME, 'UCA400R1_LFR') =
    COLLATION_KEY(S.SALES_PERSON, 'UCA400R1_LFR')
WHERE S.REGION = 'Quebec';
```

Example 3: Create an index `EMPLOYEE_NAME_SORT_KEY` for table `EMPLOYEE` based on built-in function `COLLATION_KEY` with collation name 'UCA410_LDE' tailored for German.

```
CREATE INDEX EMPLOYEE_NAME_SORT_KEY
ON EMPLOYEE (COLLATION_KEY(LASTNAME, 'UCA410_LDE', 600),
    COLLATION_KEY(FIRSTNAME, 'UCA410_LDE', 600),
    ID);
```

COMPARE_DECFLOAT

The COMPARE_DECFLOAT function returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.

►►—COMPARE_DECFLOAT(*decfloat-expression1*,*decfloat-expression2*)—►►

The schema is SYSIBM.

decfloat-expression1

An expression that returns a DECFLOAT value.

decfloat-expression2

An expression that returns a DECFLOAT value.

decfloat-expression1 is compared with *decfloat-expression2* and the result is returned according to the following rules:

- If both arguments are finite, the comparison is algebraic and follows the procedure for DECFLOAT subtraction. If the difference is exactly zero with either sign, the arguments are equal. If a nonzero difference is positive, the first argument is greater than the second argument. If a nonzero difference is negative, the first argument is less than the second.
- Positive zero and negative zero compare as equal.
- Positive infinity compares equal to positive infinity.
- Positive infinity compares greater than any finite number.
- Negative infinity compares equal to negative infinity.
- Negative infinity compares less than any finite number.
- Numeric comparison is exact and the result is determined for finite operands as if range and precision were unlimited. Overflow or underflow cannot occur.
- If either argument is NaN or sNaN (positive or negative), the result is unordered.

Numeric comparison is exact, and the result is determined for finite operands as if the range and precision were unlimited. An overflow or underflow condition cannot occur.

If one argument is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

One of the following values will be the result:

- 0** The arguments are exactly equal
- 1** *decfloat-expression1* is less than *decfloat-expression2*
- 2** *decfloat-expression1* is greater than *decfloat-expression2*
- 3** The arguments are unordered

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

Examples: The following examples demonstrate the values that will be returned when the function is used:

COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.17))	= 0
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.170))	= 2
COMPARE_DECFLOAT(DECFLOAT(2.170), DECFLOAT(2.17))	= 1
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(0.0))	= 2
COMPARE_DECFLOAT(INFINITY, INFINITY)	= 0
COMPARE_DECFLOAT(INFINITY, -INFINITY)	= 2
COMPARE_DECFLOAT(DECFLOAT(-2), INFINITY)	= 1
COMPARE_DECFLOAT(NAN, NAN)	= 3
COMPARE_DECFLOAT(DECFLOAT(-0.1), SNAN)	= 3

CONCAT

The CONCAT function combines two compatible string arguments.

►►—CONCAT(*string-expression-1*,*string-expression-2*)—◄◄

The schema is SYSIBM.

The arguments must be compatible strings. For more information on compatibility, refer to the compatibility matrix in Table 20 on page 102.

The result of the function is a string that consists of the first string followed by the second string. If either argument can be null, and if either is null, the result is the null value.

The CONCAT function is identical to the CONCAT operator. For more information, see “With the concatenation operator” on page 188.

Example: Using sample table DSN8910.EMP, concatenate column FIRSTNME with column LASTNAME. Both columns are defined as varying-length character strings.

```
SELECT CONCAT(FIRSTNME, LASTNAME)
FROM DSN8910.EMP;
```

CONTAINS

The CONTAINS function searches a text search index using criteria that are specified in a search argument and returns a result about whether or not a match was found.

►►CONTAINS(—*column-name*—,—*search-argument*—⁽¹⁾—)—►
 ,—*string-constant*—⁽²⁾—)

Notes:

- 1 The SQL statement that invokes the CONTAINS function can be dynamically prepared by using a typed parameter marker for the *search-argument*, as in the following example: CONTAINS(C1,CAST(? AS CHAR(10))).
- 2 *string-constant* must conform to the rules for the *search-argument-options*.

search-argument-options:

(1)
 QUERYLANGUAGE=—*value*—
 RESULTLIMIT=—*value*—
 SYNONYM=—☐OFF☒ON—

Notes:

- 1 The same clause must not be specified more than once.

The schema is SYSIBM.

column-name

Specifies a qualified or unqualified name of a column that has a text search index that is to be searched. The column must exist in the table or view that is identified in the FROM clause in the statement and the column of the table, or the column of the underlying base table of the view must have an associated text search index. The underlying expression of the column of a view must be a simple column reference to the column of an underlying table, directly or through another nested view.

search-argument

Specifies an expression that returns a value that is a string value (except a LOB) that contains the terms to be searched for and must not be all blanks or the empty string. The actual length of the string must not exceed 4096 Unicode characters. The value is converted to Unicode before it is used to search the text search index. The maximum number of terms per query must not exceed 1024.

string-constant

Identifies a string constant that specifies the search argument options that are in effect for the function.

The options that can be specified as part of the *search-argument-options* are as follows:

QUERYLANGUAGE = *value*

Specifies the query language. The value can be any of the supported language codes. If the QUERYLANGUAGE option is not specified, the default is the language value of the text search index that is used when this function is invoked. If the language value of the text search index is AUTO, the default value for QUERYLANGUAGE is en_US.

RESULTLIMIT = *value*

Specifies the maximum number of results to be returned from the underlying search engine. The *value* can be an integer value between 1 and 2 147 483 647. If the RESULTLIMIT option is not specified, no result limit is in effect for the query.

This scalar function cannot be called for each row of the result table, depending on the plan that the optimizer chooses. This function can be called once for the query to the underlying search engine, and a result set of all of the primary keys that match are returned from the search engine. This result set is then joined to the table containing the column to identify the result rows. In this case, the RESULTLIMIT value acts like a FETCH FIRST ?? ROWS from the underlying text search engine and can be used as an optimization. If the search engine is called for each row of the result because the optimizer determines that is the best plan, then the RESULTLIMIT option has no effect.

SYNONYM = OFF or SYNONYM = ON

Specifies whether to use a synonym dictionary that is associated with the text search index. Use the Synonym Tool to add a synonym dictionary to the collection. The default is OFF.

OFF Do not use a synonym dictionary.

ON Use the synonym dictionary that is associated with the text search index.

The result of the function is a large integer. If the second argument can be null, the result can be null. If the second argument is null, the result is the null value. If the third argument is null, the result is as if the third argument was not specified.

The result is 1 if the document contains a match for the search criteria that are specified in the search argument. Otherwise, the result is 0.

CONTAINS is a non-deterministic function.

Example 1

The following statement finds all of the employees who have "COBOL" in their resume. The text search argument is not case-sensitive.

```
SELECT EMPNO
FROM EMP_RESUME
WHERE RESUME_FORMAT = 'ascii'
AND CONTAINS(RESUME, 'cobol') = 1
```

Example 2

The search argument does not need to be a string constant. The search argument can be any SQL string expression, including a string contained in a host variable.

The following statement searches for the exact term "ate" in the COMMENT column.

```
char search_arg[100]; /* input host variable */
...
EXEC SQL DECLARE C3 CURSOR FOR
    SELECT CUSTKEY
    FROM K55ADMIN.CUSTOMERS
    WHERE CONTAINS(COMMENT, :search_arg)= 1
    ORDER BY CUSTKEY;
strcpy(search_arg, "ate");
EXEC SQL OPEN C3;
...
```

Example 3

The following statement finds 10 students at random who wrote online essays that contain the phrase "fossil fuel" in Spanish, which is "combustible fósil." These students will be invited for a radio interview. Use the synonym dictionary that was created for the associated text search index. Because only 10 students are needed, you can optimize the query by using the RESULTLIMIT option to limit the number of results from the underlying text search server.

```
SELECT FIRSTNAME, LASTNAME
FROM STUDENT_ESSAYS
WHERE CONTAINS(TERM_PAPER, 'combustible fósil',
    'QUERYLANGUAGE= es_ES RESULTLIMIT = 10 SYNONYM=ON') = 1
```

COS

The COS function returns the cosine of the argument, where the argument is an angle, expressed in radians. The COS and ACOS functions are inverse operations.

►►—COS(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable COSINE is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 0.07:

```
SELECT COS(:COSINE)
FROM SYSIBM.SYSDUMMY1;
```


COSH

The COSH function returns the hyperbolic cosine of the argument, where the argument is an angle, expressed in radians.

►►—COSH(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HCOS is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 2.35:

```
SELECT COSH(:HCOS)
FROM SYSIBM.SYSDUMMY1;
```

DATE

The DATE function returns a date that is derived from a value.

►►—DATE(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or any numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be one of the following:
 - A valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.
 - A character or graphic string with an actual length of 7 that represents a valid date in the form *yyyynnn*, where *yyyy* are digits denoting a year and *nnn* are digits between 001 and 366 denoting a day of that year.
- If *expression* is a number, it must be greater than or equal to one and less than or equal to 3652059.

If the argument is a string, the result is the date represented by the string. If the CCSID of the string is not the same as the corresponding default CCSID at the server, the string is first converted to that CCSID.

The result 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:

If the argument is a timestamp, the result is the date part of the timestamp.

If the argument is a date, the result is that date.

If the argument is a number, the result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.

If the argument is a string, the result is the date represented by the string. If the CCSID of the string is not the same as the corresponding default CCSID at the server, the string is first converted to that CCSID.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example 1: Assume that RECEIVED is a TIMESTAMP column in some table, and that one of its values is equivalent to the timestamp '1988-12-25-17.12.30.000000'. For this value, the following statement returns the internal representation of 25 December 1988.

DATE(RECEIVED)

Example 2: Assume that DATCOL is a CHAR(7) column in some table, and that one of its values is the character string '1989061'. For this value, the following statement returns the internal representation of 2 March 1989.

```
DATE(DATCOL)
```

Example 3: DB2 recognizes '1989-03-02' as the ISO representation of 2 March 1989. So, the following statement returns the internal representation of 2 March 1989.

```
DATE('1989-03-02')
```

DAY

The DAY function returns the day part of a value.

►►—DAY(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or any numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.
- If *expression* is a number, it must be a date duration or a timestamp duration. For the valid formats of datetime durations, see “Datetime operands” on page 121.

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 for the function depend on the data type of the argument:

If the argument is a date, timestamp, or string representation of either, the result is the day part of the value, which is an integer between 1 and 31.

If the argument is a date duration or timestamp duration, the result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example 1: Set the INTEGER host variable DAYVAR to the day of the month on which employee 140 in the sample table DSN8910.EMP was hired.

```
EXEC SQL SELECT DAY(HIREDATE)
        INTO :DAYVAR
        FROM DSN8910.EMP
        WHERE EMPNO = '000140';
```

Example 2: Assume that DATE1 and DATE2 are DATE columns in the same table. Assume also that for a given row in this table, DATE1 and DATE2 represent the dates 15 January 2000 and 31 December 1999, respectively. Then, for the given row:

```
DAY(DATE1 - DATE2)
```

returns the value 15.

DAYOFMONTH

The DAYOFMONTH function returns the day part of a value. The function is similar to the DAY function, except DAYOFMONTH does not support a date or timestamp duration as an argument.

►►—DAYOFMONTH(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of a date, a timestamp, a character string, or a graphic string built-in data type.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

The result of the function is a large integer between 1 and 31, which represents the day part of the value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example 1: Set the INTEGER variable DAYVAR to the day of the month on which employee 140 in sample table DSN8910.EMP was hired.

```
SELECT DAYOFMONTH(HIREDATE)
      INTO :DAYVAR
      FROM DSN8910.EMP
      WHERE EMPNO = '000140';
```

DAYOFWEEK

The DAYOFWEEK function returns an integer, in the range of 1 to 7, that represents the day of the week, where 1 is Sunday and 7 is Saturday. The DAYOFWEEK function is similar to the DAYOFWEEK_ISO function.

►►—DAYOFWEEK(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 1: Using sample table DSN8910.EMP, set the integer host variable DAY_OF_WEEK to the day of the week that Christine Haas (EMPNO = '000010') was hired (HIREDATE).

```
SELECT DAYOFWEEK(HIREDATE)
       INTO :DAY_OF_WEEK
FROM   DSN8910.EMP
WHERE  EMPNO = '000010';
```

The result is that DAY_OF_WEEK is set to 6, which represents Friday.

Example 2: The following query returns four values: 1, 2, 1, and 2.

```
SELECT DAYOFWEEK(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK(TIMESTAMP('10/12/1998', '01.02')),
       DAYOFWEEK(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK(CAST(TIMESTAMP('10/12/1998', '01.02') AS CHAR(20)))
FROM   SYSIBM.SYSDUMMY1;
```

DAYOFWEEK_ISO

The DAYOFWEEK_ISO function returns an integer, in the range of 1 to 7, that represents the day of the week, where 1 is Monday and 7 is Sunday. The DAYOFWEEK_ISO function is similar to the DAYOFWEEK function.

►►—DAYOFWEEK_ISO(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 1: Using sample table DSN8910.EMP, set the integer host variable DAY_OF_WEEK to the day of the week that Christine Haas (EMPNO = '000010') was hired (HIREDATE).

```
SELECT DAYOFWEEK_ISO(HIREDATE)
      INTO :DAY_OF_WEEK
      FROM DSN8910.EMP
      WHERE EMPNO = '000010';
```

The result is that DAY_OF_WEEK is set to 5, which represents Friday.

Example 2: The following query returns four values: 7, 1, 7, and 1.

```
SELECT DAYOFWEEK_ISO(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK_ISO(TIMESTAMP('10/12/1998', '01.02')),
       DAYOFWEEK_ISO(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK_ISO(CAST(TIMESTAMP('10/12/1998', '01.02') AS CHAR(20)))
      FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following list shows what is returned by the DAYOFWEEK_ISO function for various dates.

DATE:	DAYOFWEEK_ISO returns:
2003-12-28	'7'
2003-12-31	'3'
2004-01-01	'4'
2004-01-10	'6'
2005-01-04	'2'
2005-12-31	'7'
2006-01-01	'7'
2006-01-03	'2'

DAYOFYEAR

The DAYOFYEAR function returns an integer, in the range of 1 to 366, that represents the day of the year, where 1 is January 1.

►►—DAYOFYEAR(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 1: Using sample table DSN8910.EMP, set the integer host variable AVG_DAY_OF_YEAR to the average of the day of the year on which employees were hired (HIREDATE):

```
SELECT AVG(DAYOFYEAR(HIREDATE))  
  INTO :AVG_DAY_OF_YEAR  
FROM DSN8910.EMP;
```

The result is that AVG_DAY_OF_YEAR is set to 202.

DAYS

The DAYS function returns an integer representation of a date.

►►—DAYS(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string.

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 result is 1 more than the number of days from January 1, 0001 to *D*, where *D* is the date that would occur if the DATE function were applied to the argument.

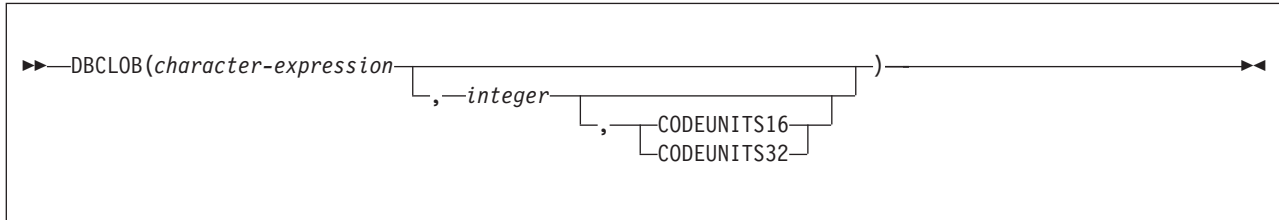
Example1: Set the INTEGER host variable DAYSVAR to the number of days that employee 140 had been with the company on the last day of 1997.

```
EXEC SQL SELECT DAYS('1997-12-31') - DAYS(HIREDATE) + 1
          INTO :DAYSVAR
          FROM DSN8910.EMP
          WHERE EMPNO = '000140';
```

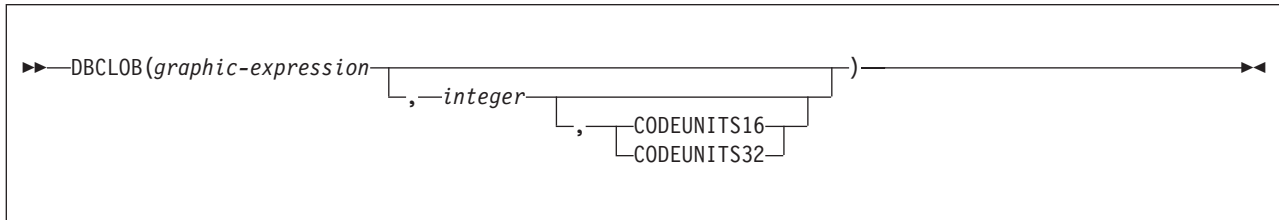
DBCLOB

The DBCLOB function returns a DBCLOB representation of a character string value (with the single-byte characters converted to double-byte characters) or a graphic string value.

Character to DBCLOB:



Graphic to DBCLOB:



The schema is SYSIBM.

Character to DBCLOB

character-expression

An expression that returns a value that is an EBCDIC-encoded or Unicode-encoded character string. It cannot be BIT data. The argument does not need to be mixed data, but any occurrences of X'0E' and X'0F' in the string must conform to the rules for EBCDIC mixed data. (See “Character strings” on page 73 for these rules.)

integer

The length attribute of the resulting DBCLOB. The value of *integer* must be between 1 and the maximum length of a DBCLOB, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string. If CODEUNITS32 is specified, the value of *integer* must be between 1 and the maximum length of a DBCLOB divided by two (to allow for an intermediate result string that is long enough to evaluate the function).

If *integer* is not specified and *character-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *character-expression*.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression*, as measured in single-byte characters, is greater than the specified length of the result, as measured in double-byte characters, the result is truncated. Unless all the truncated characters are blanks appropriate for *character-expression*, a warning is returned.

The CCSID of the result is the graphic CCSID that corresponds to the character CCSID of *character-expression*.

For EBCDIC input data, each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data and its CCSID is *M*.
- The argument is SBCS data and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*. If there is no system CCSID for mixed data, conversion is to the coded character set that the system CCSID for SBCS data identifies.

The result is derived from *S* using the following steps:

- Each shift character (X'0E' or X'0F') is removed.
- Each double-byte character remains as is.
- Each single-byte character is replaced by a double-byte character.

The replacement for a single-byte character is the equivalent DBCS character if an equivalent exists. Otherwise, the replacement is X'FEFE'. The existence of an equivalent character depends on *M*. If there is no system CCSID for mixed data, the DBCS equivalent of X'xx' for EBCDIC is X'42xx', except for X'40', whose DBCS equivalent is X'4040'.

For Unicode input data, each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data, and its CCSID is *M*.
- The argument is SBCS data, and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*.

The result is derived from *S* using the following steps:

- Each non-supplementary character is replaced by a Unicode double-byte character (a UTF-16 code point). A non-supplementary character in UTF-8 is between 1 and 3 bytes.
- Each supplementary character is replaced by a pair of Unicode double-byte characters (a pair of UTF-16 code points).

The replacement for a single-byte character is the Unicode equivalent character if an equivalent exists. Otherwise, the replacement is X'FFFD'.

Graphic to DBCLOB

graphic-expression

An expression that returns a value that is an EBCDIC-encoded or Unicode-encoded graphic string.

integer

The length attribute for the resulting varying-length graphic string. The value must be an integer between 1 and the maximum length of a DBCLOB, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If *integer* is not specified and *graphic-expression* is an empty string constant, the length attribute of the result is 1, and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of *graphic-expression*.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of *graphic-expression* is greater than the length attribute of the result, truncation is performed. Unless all of the truncated characters are double-byte blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *graphic-expression*.

The result of the function is a DBCLOB.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The length attribute and actual length of the result are measured in double-byte characters because the result is a graphic string.

Example 1: Assume that the application encoding scheme is Unicode. The following statement returns a graphic (UTF-16) host variable.

```
VALUES DBCLOB('123')
      INTO :GHV1;
```

Example 2: FIRSTNAME is a VARCHAR(12) column (Unicode UTF-8 data) in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

Function ...	Returns ...
DBCLOB(FIRSTNAME,3,CODEUNITS32)	'Jür' -- x'004A00FC0072'
DBCLOB(FIRSTNAME,3,CODEUNITS16)	'Jür' -- x'004A00FC0072'

DECFLOAT

The DECFLOAT function returns a decimal floating-point representation of either a number or a character string representation of a number, a decimal number, an integer, a floating-point number, or a decimal floating-point number.

Numeric to DECFLOAT:

►► DECFLOAT(*numeric-expression* ,34
,16) ◀◀

String to DECFLOAT:

►► DECFLOAT(*string-expression* ,34
,16) ◀◀

The schema is SYSIBM.

Numeric to DECFLOAT

numeric-expression

An expression that returns a value of any built-in numeric data type.

34 or 16

Specifies the number of digits of precision for the result. The default is 34.

String to DECFLOAT

string-expression

An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

Use the *string-expression* syntax variation to specify a negative zero as a constant, or to preserve the precision of a floating point constant.

34 or 16

Specifies the number of digits of precision for the result. The default is 34.

The result is the same number that would result from CAST(*string-expression* AS DECFLOAT(*n*)) or CAST(*numeric-expression* AS DECFLOAT(*n*)). Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming a string representation of an SQL decimal-floating point constant.

If necessary, the source is rounded to the precision of the target.

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

| For dynamic SQL statements (and static CREATE VIEW statements), the special
| register CURRENT DECFLOAT ROUNDING MODE determines the rounding
| mode.

| The result of the function is a DECFLOAT with the implicitly or explicitly specified
| number of digits of precision. If the first argument can be null, the result can be
| null; if the first argument is null, the result is the null value.

| **Note:** To increase the portability of applications, use the CAST specification. For
| more information, see “CAST specification” on page 202.
|

DECFLOAT_SORTKEY

The DECFLOAT_SORTKEY function returns a binary value that can be used when sorting DECFLOAT values. The sorting occurs in a manner that is consistent with the IEEE 754R specification on total ordering.

►►—DECFLOAT_SORTKEY(*decfloat-expression*)—◄◄

The schema is SYSIBM.

decfloat-expression

An expression that returns a DECFLOAT value.

The result is a fixed length binary string with a length attribute of 9 if *decfloat-expression* is a DECFLOAT(16) value or 17 if *decfloat-expression* is a DECFLOAT(34) value.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Assume that the following CREATE TABLE statement is used to create a table with a column that contains DECFLOAT values and the INSERT statements are used to populate the table:

```
CREATE TABLE T1(D1 DECFLOAT(16));
INSERT INTO T1 VALUES (2.100);
INSERT INTO T1 VALUES (2.10);
INSERT INTO T1 VALUES (2.1000);
INSERT INTO T1 VALUES (2.1);
```

Then the following SELECT statement is used to return the values from D1:

```
SELECT D1 FROM T1 ORDER BY D1;
```

The SELECT statement returns the following values, but because all numbers in the column have the same value, the ORDER BY clause has no effect and the values are returned in an arbitrary order:

```
D1
-----
2.1
2.1000
2.10
2.100
```

The following SELECT statement, which includes the DECFLOAT_SORTKEY function in the ORDER BY clause, returns the properly ordered values:

```
SELECT D1
FROM T1
ORDER BY (DECFLOAT_SORTKEY(D1));

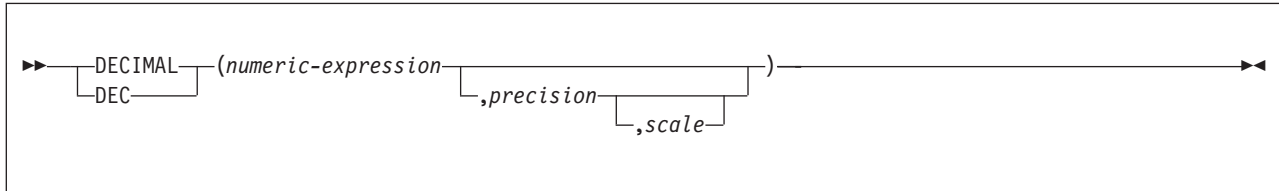
D1
-----
```


	2.1000
	2.100
	2.10
	2.1

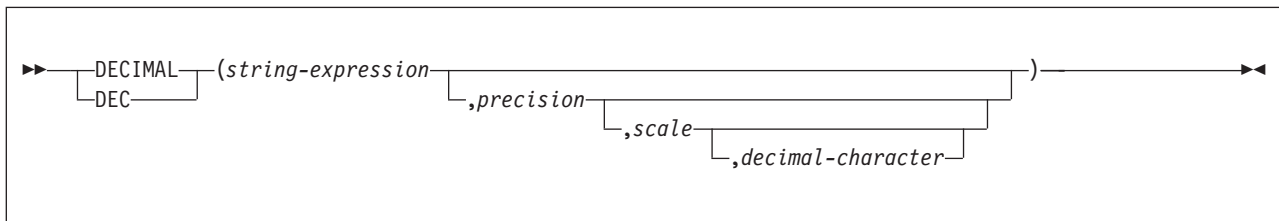
DECIMAL or DEC

The DECIMAL function returns a decimal representation of either a number or a character-string or graphic-string representation of a number, an integer, or a decimal number.

Numeric to Decimal:



String to Decimal:



The schema is SYSIBM.

Numeric to decimal

numeric-expression

An expression that returns a value of any built-in numeric data type.

precision

An integer constant with a value in the range of 1 to 31. The value of this second argument specifies the precision of the result.

The default value depends on the data type of the first argument as follows:

- 5 if the first argument is a small integer
- 11 if the first argument is a large integer
- 19 if the first argument is a big integer
- 31 if the first argument is a DECFLOAT value
- 15 in all other cases

scale

An integer constant that is greater than or equal to zero and less than or equal to *precision*. The value specifies the scale of the result. The default value is 0.

The result of the function is the same number that would occur if the argument were assigned to a decimal column or variable with precision *p* and scale *s*, where *p* and *s* are specified by the second and third arguments. An error occurs if the number of significant digits required to represent the whole part of the number is greater than *p-s*.

String to decimal

string-expression

An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes.

The string must contain a valid string representation of a number. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming a valid string representation of an SQL integer or decimal constant.

precision

An integer constant with a value in the range of 1 to 31. The value of this second argument specifies the precision of the result.

The default value depends on the data type of the first argument as follows:

- 5 if the first argument is a small integer
- 11 if the first argument is a large integer
- 15 in all other cases

scale

An integer constant that is greater than or equal to zero and less than or equal to *precision*. The value specifies the scale of the result. The default value is 0.

decimal-character

A single-byte character constant used to delimit the decimal digits in *string-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank. The default value is period (.) or comma (,); the default value cannot be used in *string-expression* if a different value for *decimal-character* is specified.

The result is the same number that would result from `CAST(string-expression AS DECIMAL(p,s))`. Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *s*. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in *string-expression* is greater than *p-s*.

The result of the function is a decimal number with precision of *p* and scale of *s*, where *p* and *s* are the second and third arguments. If the first argument can be null, the result can be null; if the first argument is null, the result is null.

Note: To increase the portability of applications when the precision is specified, use the CAST specification. For more information, see “CAST specification” on page 202.

Example 1: Represent the average salary of the employees in DSN8910.EMP as an 8-digit decimal number with two of these digits to the right of the decimal point.

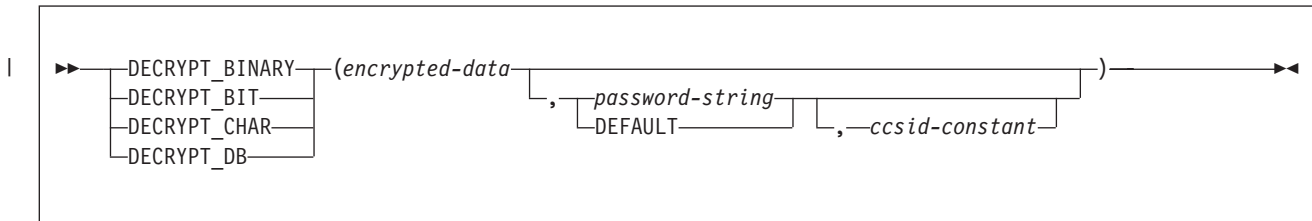
```
SELECT DECIMAL(AVG(SALARY),8,2)
FROM DSN8910.EMP;
```

Example 2: Assume that updates to the SALARY column are input as a character string that uses comma as the decimal character. For example, the user inputs 21400,50. The input value is assigned to the host variable NEWSALARY that is defined as CHAR(10), and the host variable is used in the following UPDATE statement:

```
UPDATE DSN8910.EMP
SET SALARY = DECIMAL (:NEWSALARY,9,2,',')
WHERE EMPNO = :EMPID;
```

DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, and DECRYPT_DB

The decryption functions return a value that is the result of decrypting encrypted data. The decryption functions can decrypt only values that are encrypted by using the ENCRYPT_TDES function.



The schema is SYSIBM.

The password used for decryption is either the *password-string* value or the ENCRYPTION PASSWORD value, which is assigned by the SET ENCRYPTION PASSWORD statement.

encrypted-data

An expression that returns a complete, encrypted data value of a CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY data type. The data string must have been encrypted using the ENCRYPT_TDES function. The length attribute must be greater than or equal to 0 (zero) and less than or equal to 32672.

password-string

An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes. This expression must be the same password that was used to encrypt the data or decryption will result in a different value than was originally encrypted. For enhanced security, *password-string* should be specified using a host variable rather than a string constant. If the value of the password argument is null or not provided, the data will be decrypted using the ENCRYPTION PASSWORD value, which must have been assigned by the SET ENCRYPTION PASSWORD statement.

For a static SQL statement, it is recommended that the password be specified with a host variable rather than with a string constant.

DEFAULT

The data is decrypted using the ENCRYPTION PASSWORD value, which must have been assigned by the SET ENCRYPTION PASSWORD statement.

ccsid-constant

A integer constant that specifies the CCSID in which the data should be returned by the decryption function. If DECRYPT_BIT or DECRYPT_BINARY is specified, *ccsid-constant* must not be specified. The default is

- The ENCODING bind option of the plan or package or the APPLICATION ENCODING SCHEMA option of the CREATE PROCEDURE or ALTER PROCEDURE statement for native SQL procedures that contain the static SQL statements
- The value of the APPLICATION ENCODING special register for dynamic SQL statements

The data type of the result of the function is determined by the function that is specified and the data type of the first argument, as shown in the following table. If the cast from the actual type of the encrypted data to the result of the function is not supported, a warning or error is returned.

Table 55. Result of the decryption function

Function	Type of first argument	Actual type of encrypted data	Result
DECRYPT_BINARY	FOR BIT DATA ¹ , BINARY, VARBINARY	Any string (except for LOBs)	VARBINARY
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	CHAR, VARCHAR	VARCHAR FOR BIT DATA
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (UTF16)	Warning or error If a warning is returned, the result is VARCHAR FOR BIT DATA
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (not UTF16)	Warning or error If a warning is returned, the result is VARCHAR FOR BIT DATA
DECRYPT_BIT	FOR BIT DATA, BINARY, VARBINARY	BINARY, VARBINARY	Warning or error If a warning is returned, the result is VARCHAR FOR BIT DATA
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	CHAR, VARCHAR	VARCHAR(3)
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (UTF16)	VARCHAR(3)
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	GRAPHIC, VARGRAPHIC (not UTF16)	Warning or error If a warning is returned, the result is VARCHAR(3)
DECRYPT_CHAR	FOR BIT DATA, BINARY, VARBINARY	BINARY, VARBINARY	Warning or error If a warning is returned, the result is VARCHAR(3)
DECRYPT_DB	FOR BIT DATA, BINARY, VARBINARY	CHAR, VARCHAR, GRAPHIC, VARGRAPHIC	VARGRAPHIC
DECRYPT_DB	FOR BIT DATA, BINARY, VARBINARY	BINARY, VARBINARY	Warning or error If a warning is returned, the result is VARGRAPHIC

Table 55. Result of the decryption function (continued)

Function	Type of first argument	Actual type of encrypted data	Result
Note: ¹ FOR BIT DATA means CHAR or VARCHAR FOR BIT DATA			

If *encrypted-data* included a hint, the hint is not returned by the function. The length attribute of the result is the length attribute of *encrypted-data* minus 8 bytes. The actual length of the value that is returned by the function will match the length of the original string that was encrypted. If *encrypted-data* includes bytes beyond the encrypted string, these bytes are not returned by the function.

Administration of encrypted data: The decryption functions can only decrypt data that was encrypted using the Triple DES encryption algorithm. Therefore, columns with encrypted data can only be used after replication if they were encrypted using the Triple DES encryption algorithm.

If the data is decrypted using a different CCSID than the originally encrypted value, it is possible that expansion might occur when converting the decrypted value to this CCSID. In such situations, the *encrypted-data* value must first be cast to a VARCHAR string with a larger number of bytes before performing the decryption functions.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

For additional information about using the decryption functions, see "ENCRYPT_TDES" on page 363 and "GETHINT" on page 373.

Password protection: To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source statement. Instead, use the ENCRYPTION PASSWORD special register or specify the password using a host variable.

Example 1: Set the ENCRYPTION PASSWORD value to 'Ben123' and use it as the password to insert a decrypted social security number into the table. Decrypt the value of the added social security number, using the ENCRYPTION PASSWORD value.

```
SET ENCRYPTION PASSWORD ='Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES('289-46-8832');
SELECT DECRYPT_CHAR(SSN) FROM EMP;
```

This example returns the value '289-46-8832'.

Example 2: Decrypt the social security number that is inserted into the table. Instead of using the ENCRYPTION PASSWORD value, explicitly specify 'Ben123' as the encryption password.

```
SELECT DECRYPT_CHAR(SSN,'Ben123') FROM EMP;
```

This example returns the value '289-46-8832'.

Example 3: Insert a decrypted social security number into the table, explicitly specifying 'Ben123' as the password. Decrypt the data and have it converted to CCSID 1208.

```

SET ENCRYPTION PASSWORD = 'Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES('289-46-8832');
SELECT DECRYPT_CHAR(SSN) FROM EMP;

```

When a CCSID is specified, it might be necessary to explicitly cast the data to a longer value to ensure that there is room for expansion when the data is decrypted. The following example illustrates the technique:

```

SELECT DECRYPT_CHAR(CAST(SSN AS VARCHAR(57)),
                   'Ben123',1208)
FROM EMP;

```

In the first case, where the data is not cast to a longer value, the result is a VARCHAR(11) value. In the second case, to allow for expansion, SSN is cast as VARCHAR(57) ($11 * 3 + 24$). Casting the data to a longer value allows for three times expansion in the normal VARCHAR(11) result. Three times expansion is often associated with a worst case of ASCII or EBCDIC to Unicode UTF-8 conversion. In both cases in this example, the result is the VARCHAR(11) value '289-46-8832'.

DEGREES

The DEGREES function returns the number of degrees of the argument, which is an angle, expressed in radians.

►►—DEGREES(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HRAD is a DOUBLE with a value of 3.1415926536. The following statement returns a double precision floating-point number with an approximate value of 180.0.

```
SELECT DEGREES(:HRAD)
FROM SYSIBM.SYSDUMMY1;
```


DIFFERENCE

The DIFFERENCE function returns a value, from 0 to 4, that represents the difference between the sounds of two strings, based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.

►►—DIFFERENCE(*expression-1*,*expression-2*)—◄◄

The schema is SYSIBM.

expression-1 **or** *expression-2*

Each expression must return a value that is a built-in numeric, character string, or graphic string data type that is not a LOB. A numeric argument is cast to a character string before the function is evaluated. For more information on converting a numeric string to a character string, see “VARCHAR” on page 558.

The data type of the result is INTEGER. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Example 1: Find the DIFFERENCE and SOUNDEX values for 'CONSTRAINT' and 'CONSTANT':

```
SELECT DIFFERENCE('CONSTRAINT','CONSTANT'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONSTANT')
FROM SYSIBM.SYSDUMMY1;
```

This example returns the values 4, C523, and C523. Since the two strings return the same SOUNDEX value, the difference is 4 (the highest value possible).

Example 2: Find the DIFFERENCE and SOUNDEX values for 'CONSTRAINT' and 'CONTRITE':

```
SELECT DIFFERENCE('CONSTRAINT','CONTRITE'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONTRITE')
FROM SYSIBM.SYSDUMMY1;
```

This example returns the values 2, C523, and C536. In this case, the two strings return different SOUNDEX values, and hence, a lower difference value.

DIGITS

The DIGITS function returns a character string representation of the absolute value of a number.

►►—DIGITS(*numeric-expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value that is a SMALLINT, INTEGER, BIGINT, or DECIMAL built-in numeric data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal point. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- *p* if the argument is a decimal number with a precision of *p*

The CCSID of the result is determined from the context in which the function was invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Example 1: Assume that an INTEGER column called INTCOL containing a 10-digit number is in a table called TABLEX. INTCOL has the data type INTEGER instead of CHAR(10) to save space. the following query lists all combinations of the first four digits in column INTCOL.

```
SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
FROM TABLEX;
```

Example 2: Assume that COLUMNX has the data type DECIMAL(6,2), and that one of its values is -6.28. For this value, the following statement returns the value '000628'.

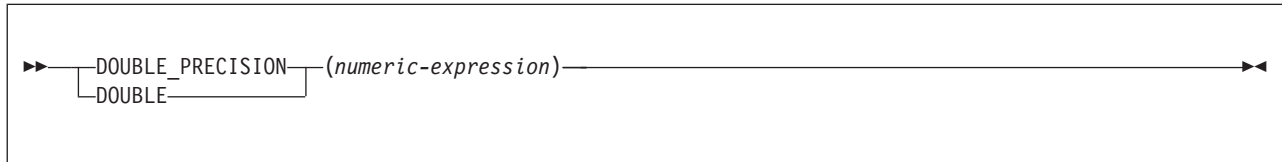
```
DIGITS(COLUMNX)
```

The result is a string of length six (the precision of the column) with leading zeros padding the string out to this length. Neither sign nor decimal point appear in the result.

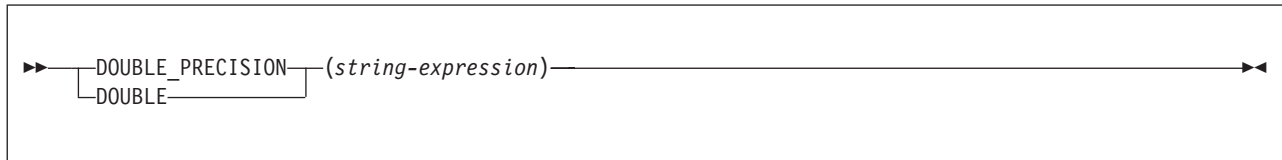
DOUBLE_PRECISION or DOUBLE

The DOUBLE_PRECISION and DOUBLE functions returns a floating-point representation of either a number or a character-string or graphic-string representation of a number, an integer, a decimal number, or a floating-point number.

Numeric to Double:



String to Double:



The schema is SYSIBM.

Numeric to Double

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the expression were assigned to a double precision floating-point column or variable.

String to Double

string-expression

An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from CAST(*string-expression* AS DOUBLE PRECISION). Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming a valid string representation of an SQL floating-point, integer, or decimal constant.

The result of the function is a double precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note: To increase the portability of applications, use the CAST specification. For more information, see “CAST specification” on page 202.

FLOAT can be specified as a synonym for DOUBLE or DOUBLE_PRECISION.

Example: Using sample table DSN8910.EMP, find the ratio of salary to commission for employees whose commission is not zero. The columns involved in the

calculation, SALARY and COMM, have decimal data types. To eliminate the possibility of out-of-range results, apply the DOUBLE function to SALARY so that the division is carried out in floating-point.

```
SELECT EMPNO, DOUBLE(SALARY)/COMM
FROM DSN8910.EMP
WHERE COMM > 0;
```

DSN_XMLVALIDATE

The DSN_XMLVALIDATE function returns an XML value that is the result of applying XML schema validation to the first argument of the function. DSN_XMLVALIDATE can validate XML data that has a maximum length of 2 GB - 1 byte.

```
DSN_XMLVALIDATE(string-expression, [schema-name-string | target-namespace-uri-string, schema-location-string])
```

The schema is SYSIBM.

string-expression

An expression that returns a built-in character, graphic, or binary string. The value must be a well-formed XML document that conforms to the XML Version 1.0 standard.

schema-name-string

An expression that returns a built-in varying length character string that is not a CLOB. The value specifies the name of the XML schema object that is used for validation. The value must not be an empty string or the null value, and the actual length must be less than or equal to 257. If the XML schema name is qualified, the qualifier must be SYSXSR (SYSXSR is the default qualifier). The value must identify a registered XML schema in the DB2 XML schema repository.

target-namespace-uri-string

An expression that returns a built-in varying length character string that is not a CLOB, with a length attribute that is not greater than 1000. The value specifies the target namespace name or universal resource identifier (URI) of the XML schema that is to be used for validation. If the value is an empty string of the null value, no namespace is used to locate the XML schema.

schema-location-string

An expression that returns a built-in varying length character string that is not a CLOB, with a length attribute that is not greater than 1000. The value specifies the XML schema location hint URI of the XML schema that is to be used for validation. If the value is an empty string of the null value, no schema location is used to locate the XML schema.

If *target-namespace-uri-string* and *schema-location-string* are specified, the combination must identify a registered XML schema in the DB2 XML schema repository, and there must be only one such registered XML schema.

A schema must be registered successfully in the DB2 XML schema repository before it can be used for DSN_XMLVALIDATE. If the validation fails, DB2 returns an error.

To use the version of DSN_XMLVALIDATE described here, you need to explicitly qualify DSN_XMLVALIDATE with SYSIBM. When DSN_XMLVALIDATE is unqualified or is qualified with SYSFUN, the version of the DSN_XMLVALIDATE function that is used as input to the DSN_XMLPARSE function is invoked.

| The result of the function is an XML value. The result can be null; if the first
| argument is null, the result is the null value.

| *Example 1:* The following example shows how the DSN_XMLVALIDATE function
| validates the XML data that is contained in the *value_host_var* host variable. The
| XML schema, SYSXSR.ORDERSCHEMA, was registered prior to this statement:

```
|      INSERT INTO T1(C1) VALUES(  
|          DSN_XMLVALIDATE(:value_host_var, 'SYSXSR.MYXMLSCHEMA'));
```

| *Example 2:* The following example is similar to the previous example but references
| the namespace and schema location:

```
|      INSERT INTO T1(C1) VALUES(  
|          DSN_XMLVALIDATE(:value_host_var,  
|                          'http://www.n1.com',  
|                          'http://www.n1.com/report.xsd'));
```

DSN_XMLVALIDATE

The result of DSN_XMLVALIDATE is a varying-length binary value of up to 50 MB. The result of this function is relevant only as input to the XMLPARSE function.

This version of DSN_XMLVALIDATE can be used, but has been deprecated in favor of a new version of DSN_XMLVALIDATE.

CLOB input with an XML schema name:

```
»»—DSN_XMLVALIDATE(clob-expression,varchar-expression)—————»«
```

CLOB input with namespace and location hint URIs:

```
»»—DSN_XMLVALIDATE(clob-expression,varchar-expression1,varchar-expression2)—————»«
```

BLOB input with an XML schema name:

```
»»—DSN_XMLVALIDATE(blob-expression,varchar-expression)—————»«
```

BLOB input with namespace and location hint URIs:

```
»»—DSN_XMLVALIDATE(blob-expression,varchar-expression1,varchar-expression2)—————»«
```

VARCHAR input with an XML schema name:

```
»»—DSN_XMLVALIDATE(varchar-expression1,varchar-expression2)—————»«
```

VARCHAR input with namespace and location hint URIs:

```
»»—DSN_XMLVALIDATE(varchar-expression1,varchar-expression2,varchar-expression3)—————»«
```

The schema is SYSFUN.

CLOB input with an XML schema name:

clob-expression

An expression that returns a value that is CLOB(250M). *clob-expression* contains the XML value that is to be validated. DB2 will convert this value from the application encoding scheme to UTF-8. *clob-expression* cannot be null.

varchar-expression

An expression that returns a value that is VARCHAR(257). *varchar-expression* contains the name of the XML schema object to use for validation. *varchar-expression* cannot be null.

If the name contains a '.', the characters before the '.' are the schema name, which is the qualifier for the name of the XML schema to use for validation of the VALUE parameter. It must be the value 'SYSXSR'. If the name does not contain '.', DB2 will use the value 'SYSXSR' for the relational schema qualifier. The number of characters after the '.' must not exceed 128.

Rules for valid characters and delimiters that apply to any SQL identifier also apply to *varchar-expression*. If an invalid character is used as an ordinary SQL identifier, an error will be returned.

CLOB input with namespace and location hint URIs:

clob-expression

An expression that returns a value that is CLOB(250M). *clob-expression* contains the XML value that is to be validated. *clob-expression* cannot be null.

varchar-expression1

An expression that returns a value that is VARCHAR(1000). *varchar-expression1* contains the target namespace URI.

varchar-expression2

An expression that returns a value that is VARCHAR(1000). *varchar-expression2* contains the XML schema location hint URI.

Both *varchar-expression1* and *varchar-expression2* can be null. If both values are null, DB2 will look for a single XML schema with null values for both the target namespace and the XML schema location hint. An XML schema with the combination of namespace URI and location hint URI must exist at the current server. The combination of namespace URI and location hint URI must be unique.

BLOB input with an XML schema name:

blob-expression

An expression that returns a value that is BLOB(250M). *blob-expression* contains the XML value that is to be validated. *blob-expression* cannot be null.

varchar-expression

An expression that returns a value that is VARCHAR(257). *varchar-expression* contains the name of the XML schema object to use for validation. *varchar-expression* cannot be null.

If the name contains a '.', the characters before the '.' are the schema name, which is the qualifier for the name of the XML schema to use for validation of the VALUE parameter. It must be the value 'SYSXSR'. If the name does not contain '.', DB2 will use the value 'SYSXSR' for the relational schema qualifier. The number of characters after the '.' must not exceed 128.

Rules for valid characters and delimiters that apply to any SQL identifier also apply to *varchar-expression*. If an invalid character is used as an ordinary SQL identifier, an error will be returned.

BLOB input with namespace and location hint URIs:

blob-expression

An expression that returns a value that is BLOB(250M). *blob-expression* contains the XML value that is to be validated. *blob-expression* cannot be null.

varchar-expression1

An expression that returns a value that is VARCHAR(1000). *varchar-expression1* contains the target namespace URI.

varchar-expression2

An expression that returns a value that is VARCHAR(1000). *varchar-expression2* contains the XML schema location hint URI.

Both *varchar-expression1* and *varchar-expression2* can be null. If both values are null, DB2 will look for a single XML schema with null values for both the target namespace and the XML schema location hint. An XML schema with the combination of namespace URI and location hint URI must exist at the current server. The combination of namespace URI and location hint URI must be unique.

VARCHAR input with an XML schema name:

varchar-expression1

An expression that returns a value that is VARCHAR(32K). *varchar-expression1* contains the XML value that is to be validated. DB2 will convert this value from the application encoding scheme to UTF-8. *varchar-expression1* cannot be null.

varchar-expression2

An expression that returns a value that is VARCHAR(257). *varchar-expression2* contains the name of the XML schema object to use for validation. *varchar-expression2* cannot be null.

If the name contains a '.', the characters before the '.' are the schema name, which is the qualifier for the name of the XML schema to use for validation of the VALUE parameter. It must be the value 'SYSXSR'. If the name does not contain '.', DB2 will use the value 'SYSXSR' for the relational schema qualifier. The number of characters after the '.' must not exceed 128.

Rules for valid characters and delimiters that apply to any SQL identifier also apply to *varchar-expression2*. If an invalid character is used as an ordinary SQL identifier, an error will be returned.

VARCHAR input with namespace and location hint URIs:

varchar-expression1

An expression that returns a value that is VARCHAR(32K). *varchar-expression1* contains the XML value that is to be validated. DB2 will convert this value from the application encoding scheme to UTF-8. *varchar-expression1* cannot be null.

varchar-expression2

An expression that returns a value that is VARCHAR(1000). *varchar-expression2* contains the target namespace URI.

varchar-expression3

An expression that returns a value that is VARCHAR(1000). *varchar-expression3* contains the XML schema location hint URI.

Both *varchar-expression1* and *varchar-expression2* can be null. If both values are null, DB2 will look for a single XML schema with null values for both the target namespace and the XML schema location hint. An XML schema with the combination of namespace URI and location hint URI must exist at the current server. The combination of namespace URI and location hint URI must be unique.

Example 1: The following example shows how the DSN_XMLVALIDATE function provides input to the XMLPARSE function. The XML schema, SYSXSR.ORDERSHEMA, was registered prior to this statement:

```
INSERT INTO T1(C1) VALUES
(XMLPARSE (DOCUMENT SYSFUN.DSN_XMLVALIDATE(:value_host_variable,
'SYSXSR.ORDERSHEMA')));
```

Example 2: The following example is similar to the previous example but references the namespace and schema location:

```
INSERT INTO T1(C1) VALUES
(XMLPARSE (DOCUMENT DSN_XMLVALIDATE(:value_host_variable,
'REPORT',
'http://www.n1.com/report.xsd')));
```

EBCDIC_CHR

The EBCDIC_CHR function returns the character that has the EBCDIC code value that is specified by the argument.

►►—EBCDIC_CHR(*expression*)—◄◄

The schema is SYSIBM.

expression

An expression that returns a BIGINT, INTEGER, or SMALLINT built-in data type value.

The result of the function is a CHAR(1) string encoded in the SBCS EBCDIC CCSID (regardless of the setting of the MIXED option in DSNHDECP). If the value of *expression* is not in the range of 0 to 255, the null value is returned.

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

Example: Set *hv* with the Euro symbol "€" in CCSID 1140:

```
SET :hv = EBCDIC_CHR(159);  -- x'9F'
```

Set *hv* with the Euro symbol "€" in CCSID 1142:

```
SET :hv = EBCDIC_CHR(90);   -- x'5A'
```

In both cases, the "€" is assigned to *hv*, but because the Euro symbol is located at different code points for the two CCSIDs, the input value is different.

EBCDIC_STR

The EBCDIC_STR function returns a string, in the system EBCDIC CCSID, that is an EBCDIC version of the string.

►—EBCDIC_STR(*string-expression*)—◄

The schema is SYSIBM.

The system EBCDIC CCSID is defined as the SBCS EBCDIC CCSID on a MIXED=NO system or the MIXED EBCDIC CCSID on a MIXED=YES system.

string-expression

An expression that returns a value of a built-in character or graphic string. If the string is a character string, it cannot be bit data. *string-expression* must be an ASCII, EBCDIC, or Unicode string. EBCDIC_STR returns an EBCDIC version of the string. Non-EBCDIC characters are converted to the form \xxxx, where xxxx represents a UTF-16 code unit.

The length attribute of the result is calculated using the formulas in Table 35 on page 126. The length attribute of the result will be $\text{MAX}(\text{length_of_input_string}, 32704)$. Where n is the result of applying the formulas in Table 35 on page 126 based on input and output data types.

The result of the function is an EBCDIC character string (in the system EBCDIC CCSID). If the actual length of the result string exceeds the maximum for the return type, an error occurs.

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

Example: The following example returns the EBCDIC string equivalent of the text string "Hi my name is Андрей (Andrei)"

```
SET :HV1 = EBCDIC_STR('Hi, my name is Андрей (Andrei)');
```

HV1 is assigned the value "Hi, my name is \0410\043D\0434\0440\0435\0439 (Andrei)"

ENCRYPT_TDES

The ENCRYPT_TDES function returns a value that is the result of encrypting the first argument by using the Triple DES encryption algorithm. The function can also set the password that is used for encryption.

The encryption password can also be set by using the ENCRYPTION PASSWORD value, which is assigned by using the SET ENCRYPTION PASSWORD statement.

```
►►—ENCRYPT_TDES(data-string—  
               |, —password-string—  
               |, —hint-string—)——►►
```

The schema is SYSIBM.

data-string

An expression that returns the string value to be encrypted. The string expression must return a built-in string data type that is not a LOB. The length attribute must be greater than or equal to 0 (zero). The length attribute is limited to 32640 if *hint-string* is specified and 32672 if *hint-string* is not specified.

password-string

An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes.

The value represents the password that is used to encrypt *data-string*. If the value of the password argument is null or not specified, the data is encrypted using the ENCRYPTION PASSWORD value, which must have been assigned by the SET ENCRYPTION PASSWORD statement.

hint-string

An expression that returns a CHAR or VARCHAR value up to 32 bytes that is to help data owners remember passwords (for example, 'Ocean' as a hint to remember 'Pacific').

If a hint value is specified, the hint is embedded into the result and can be retrieved using the GETHINT function. If this argument is null or not specified and no hint was specified when the ENCRYPTION PASSWORD was set, no hint is embedded in the result. If *password-string* is not specified, the hint can be specified using the SET ENCRYPTION PASSWORD statement.

The data type of the result is determined by the first argument as shown in the following table:

Table 56. Data type of the results of the ENCRYPT_TDES function

Data type of the first argument	Data type of the result
BINARY, VARBINARY	VARBINARY
CHAR, VARCHAR, GRAPHIC, VARGRAPHIC	VARCHAR FOR BIT DATA

The encoding scheme of the result is the same as the encoding scheme of *data-string*. If the result is character data, the result is bit data.

The length attribute of the result is different depending of whether *hint-string* is specified:

- If *hint-string* is specified, the length attribute of the result is the length attribute of the non-encrypted data + 24 bytes + number of bytes to the next 8 byte boundary + 32 bytes for the hint.
- If *hint-string* is not specified, the length attribute of the result is the length attribute of the non-encrypted data + 24 bytes + the number of bytes to the next 8 byte boundary.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The encrypted result is longer than the *data-string* value. Therefore, when assigning encrypted values, ensure that the target is declared with a length that can contain the entire encrypted value.

When encrypting data, be aware of the following points:

- **Password protection:** To prevent inadvertent access to the encryption password, do not specify *password-string* as a string constant in the source for a program, procedure, or function. Instead, use the SET ENCRYPTION PASSWORD statement or a variable.
- **Encryption algorithm:** The internal encryption algorithm used is Triple DES cipher block chaining (CBC) with padding. The 128-bit secret key is derived from the password using an MD5 hash.
- **Encryption passwords and data:** It is your responsibility to perform password management. After data is encrypted, only the password that is used to encrypt it can be used to decrypt it. If a different password is used to decrypt the data than was used to encrypt the data, the results of decryption will not match the original string. No error or warning is returned. CHAR variables might be padded with blanks if they are used to set password values. The encrypted result might contain null terminator and other non-printable characters.
- **Table column definitions:** When defining columns and types to contain encrypted data, always calculate the length attribute as follows:
 - For encrypted data with an embedded hint, the column length should be the length attribute of the non-encrypted data + 24 bytes + number of bytes to the next 8 byte boundary + 32 bytes for the hint.
 - For encrypted data without an embedded hint, the column length should be the length attribute of the non-encrypted data + 24 bytes + number of bytes to the next 8 byte boundary.

Here are some sample column length calculations, which assume that a hint is not embedded:

Maximum length of non-encrypted data	6 bytes
24 bytes for encryption key	24 bytes
Number of bytes to the next 8 byte boundary	2 bytes

Encrypted data column length	32 bytes
Maximum length of non-encrypted data	32 bytes
24 bytes for encryption key	24 bytes
Number of bytes to the next 8 byte boundary	0 bytes

Encrypted data column length	56 bytes

- **Administration of encrypted data:** Encrypted data can be decrypted only on servers that support the decryption of data that was encrypted using the Triple

| DES encryption algorithm. Hence, replication of columns with encrypted data
| should only be done to servers that support the decryption functions and the
| same encryption algorithms.

ENCRYPT can be specified as a synonym for ENCRYPT_TDES. DB2 supports this keyword to provide compatibility with other products in the DB2 family.

Example 1: Encrypt the social security number that is inserted into the table. Set the ENCRYPTION PASSWORD value to 'Ben123' and use it as the password.

```
SET ENCRYPTION PASSWORD ='Ben123';  
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES ('289-46-8832');
```

Example 2: Encrypt the social security number that is inserted into the table. Explicitly specify 'Ben123' as the encryption password.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES ('289-46-8832','Ben123');
```

Example 3: Encrypt the social security number that is inserted into the table. Specify 'Pacific' as the encryption password, and provide 'Ocean' as a hint to help the user remember the password of 'Pacific'.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT_TDES ('289-46-8832','Pacific','Ocean');
```

The preceding statement returns a double precision floating-point number with an approximate value of 31.62.

EXP

The EXP function returns a value that is the base of the natural logarithm (e), raised to a power that is specified by the argument. The EXP and LN functions are inverse operations.

►►—EXP(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

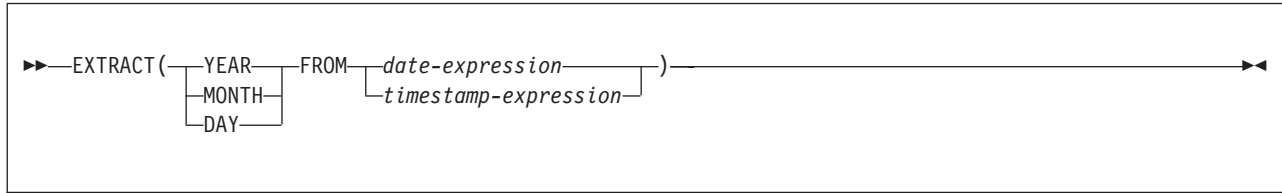
Example: Assume that host variable E is DECIMAL(10,9) with a value of 3.453789832. The following statement returns a double precision floating-point number with an approximate value of 31.62.

```
SELECT EXP(:E)
FROM SYSIBM.SYSDUMMY1;
```

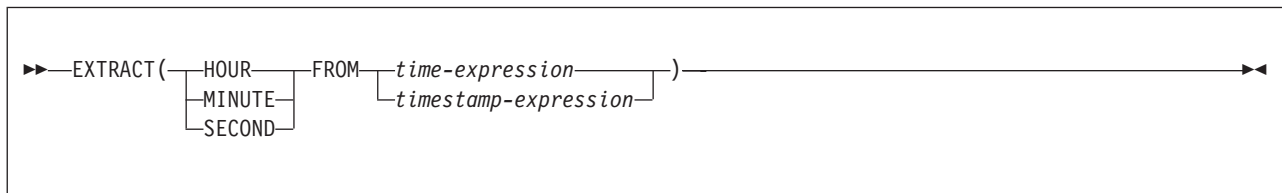

EXTRACT

The EXTRACT function returns a portion of a date or timestamp, based on its arguments.

Extract date values:



Extract time values:



The schema is SYSIBM.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Extract date values

YEAR

Specifies that the year portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the YEAR scalar function. For more information, see “YEAR” on page 600.

MONTH

Specifies that the month portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the MONTH scalar function. For more information, see “MONTH” on page 423.

DAY

Specifies that the day portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the DAY scalar function. For more information, see “DAY” on page 330.

date-expression

An expression that returns the value of either a built-in date or built-in character string data type.

If *date-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a date. For the valid formats of string representations of dates, see “String representations of datetime values” on page 89.

timestamp-expression

An expression that returns the value of either a built-in timestamp or built-in character string data type.

If *timestamp-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid character-string or graphic-string representation of a timestamp. For the valid formats of string representations of timestamps, see “String representations of datetime values” on page 89.

Extract time values

hour

Specifies that the hour portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the HOUR scalar function. For more information, see “HOUR” on page 381.

minute

Specifies that the minute portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the MINUTE scalar function. For more information, see “MINUTE” on page 420.

second

Specifies that the second portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the SECOND scalar function. For more information, see “SECOND” on page 501.

time-expression

An expression that returns the value of either a built-in time or built-in character string data type.

If *time-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a time. For the valid formats of string representations of times, see “String representations of datetime values” on page 89.

timestamp-expression

An expression that returns the value of either a built-in timestamp or built-in character string data type.

If *timestamp-expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a timestamp. For the valid formats of string representations of timestamps, see “String representations of datetime values” on page 89.

The data type of the result of the function depends on the part of the datetime value that is specified:

- The result is INTEGER, if one of the following is specified:
 - YEAR
 - MONTH
 - DAY
 - HOUR
 - MINUTE
- The result is DECIMAL(8,6), if SECOND is specified. The fractional digits contains microseconds.

Example 1:

Assume that the column PRSTDAT has an internal value that is equivalent to 2010-12-25. The following statement returns the value 12:

```
SELECT EXTRACT(MONTH FROM PRSTDAT)
FROM PROJECT;
```

FLOAT

The FLOAT function returns a floating-point representation of either a number or a string representation of a number. FLOAT is a synonym for the DOUBLE function.

►►—FLOAT(*numeric-expression*)—◄◄

The schema is SYSIBM.

FLOAT is a synonym for the DOUBLE function. See “DOUBLE_PRECISION or DOUBLE” on page 353 for details.

FLOOR

The FLOOR function returns the largest integer value that is less than or equal to the argument.

►►—FLOOR(*numeric-expression*)—◄◄

The schema is SYSIBM.

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

The result of the function has the same data type and length attribute as the argument. When the argument is DECIMAL, the scale of the result is 0 and not the scale of the input argument. For example, an argument with a data type of DECIMAL(5,5) results in DECIMAL(5,0).

The result can be null. If the argument is null, the result is the null value.

Example 1: Using sample table DSN8910.EMP, find the highest monthly salary, rounding the result down to the next integer. The SALARY column has a decimal data type.

```
SELECT FLOOR(MAX(SALARY)/12)
FROM DSN8910.EMP;
```

This example returns 04395 because the highest paid employee is Christine Haas who earns \$52750.00 per year. Her average monthly salary before applying the FLOOR function is 4395.83.

Example 2: This example demonstrates using FLOOR with both positive and negative numbers.

```
SELECT FLOOR( 3.5),
       FLOOR( 3.1),
       FLOOR(-3.1),
       FLOOR(-3.5)
FROM SYSIBM.SYSDUMMY1;
```

This example returns (leading zeros are shown to demonstrate the precision and scale of the result):

03. 03. -04. -04.

GENERATE_UNIQUE

The `GENERATE_UNIQUE` function returns a bit data character string that is unique, compared to any other execution of the same function.



►►—`GENERATE_UNIQUE()`—►►

The schema is `SYSIBM`.

The `GENERATE_UNIQUE` function returns a bit data character string 13 bytes long (`CHAR(13) FOR BIT DATA`) that is unique compared to any other execution of the same function. The function is defined as not deterministic. Although the function has no arguments, the empty parentheses must be specified when the function is invoked.

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and, if in a sysplex environment, the sysplex member where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The sequence is based on the time when the function was executed.

This function differs from using the special register `CURRENT_TIMESTAMP` in that a unique value is generated for each row of a multiple row insert statement, an insert statement with a fullselect, or an insert operation in a `MERGE` statement.

The timestamp value that is part of the result of this function can be determined using the `TIMESTAMP` function with the result of `GENERATE_UNIQUE` as an argument.

Example: Create a table that includes a column that is unique for each row. Populate this column using the `GENERATE_UNIQUE` function. Notice that the `UNIQUE_ID` column is defined as `FOR BIT DATA` to identify the column as a bit data character string.

```
CREATE TABLE EMP_UPDATE
  (UNIQUE_ID VARCHAR(13)FOR BIT DATA,
   EMPNO CHAR(6),
   TEXT VARCHAR(1000));
INSERT INTO EMP_UPDATE VALUES (GENERATE_UNIQUE(),'000020','Update entry 1...');
INSERT INTO EMP_UPDATE VALUES (GENERATE_UNIQUE(),'000050','Update entry 2...');
```

This table will have a unique identifier for each row if `GENERATE_UNIQUE` is always used to set the value the `UNIQUE_ID` column. You can create an insert trigger on the table to ensure that `GENERATE_UNIQUE` is used to set the value:

```
CREATE TRIGGER EMP_UPDATE_UNIQUE
  NO CASCADE BEFORE INSERT ON EMP_UPDATE
  REFERENCING NEW AS NEW_UPD
  FOR EACH ROW MODE DB2SQL
  SET NEW_UPD.UNIQUE_ID = GENERATE_UNIQUE();
```

With this trigger, the previous INSERT statements that were used to populate the table could be issued without specifying a value for the UNIQUE_ID column:

```
INSERT INTO EMP_UPDATE (EMPNO,TEXT) VALUES ('000020','Update entry 1...');
INSERT INTO EMP_UPDATE (EMPNO,TEXT) VALUES ('000050','Update entry 2...');
```

The timestamp (in UTC) for when a row was added to EMP_UPDATE can be returned using:

```
SELECT TIMESTAMP(UNIQUE_ID), EMPNO, TEXT FROM EMP_UPDATE;
```

Therefore, the table does not need a timestamp column to record when a row is inserted.

GETHINT

The GETHINT function returns a hint for the password if a hint was embedded in the encrypted data. A password hint is a phrase that helps you remember the password with which the data was encrypted. For example, 'Ocean' might be used as a hint to help remember the password 'Pacific'.

►►—GETHINT(*encrypted-data*)—◄◄

The schema is SYSIBM.

encrypted-data

An expression that returns a string that contains a complete, encrypted data string. *encrypted-data* must return a value that is a CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BINARY, or VARBINARY built-in data type. The string must have been encrypted using ENCRYPT_TDES function.

The result of the function is VARCHAR(32). The actual length of the result is the actual length of the hint that was provided when the data was encrypted.

The result can be null; if the argument is null or no hint was specified when the ENCRYPT_TDES function was used to encrypt the data, the result is the null value.

The encoding scheme of the result is the same as the encoding scheme of *encrypted-data*. If *encrypted-data* is bit data, the CCSID of the result is the default character CCSID for that encoding scheme. Otherwise, the CCSID of the result is the same as the CCSID of *encrypted-data*.

For additional information about this function, see “DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, and DECRYPT_DB” on page 346 and “ENCRYPT_TDES” on page 363.

Example: This example shows how to embed a hint for the password when encrypting data and how to later use the GETHINT function to retrieve the embedded hint. In this example, the hint 'Ocean' is used to help remember the encryption password 'Pacific'.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT_TDES ('289-46-8832','Pacific','Ocean');  
SELECT GETHINT (SSN) FROM EMP;
```

The value that is returned is 'Ocean'.

GETVARIABLE

The GETVARIABLE function returns a varying-length character-string representation of the current value of the session variable that is identified by the argument.

```
►► GETVARIABLE(string-constant [ , default-value ] [ , CAST(—NULL AS VARCHAR(1)—) ] ) ►►
```

The schema is SYSIBM.

string-constant

Specifies a string constant that contains the name of the session variable whose value is to be returned. The string constant:

- Must have a length that does not exceed 142 bytes.
- Must contain the fully qualified name of the variable, with no embedded blanks. Delimited identifiers must not be specified.
- Must not contain lowercase letters or characters that cannot be specified in an ordinary identifier.

The schema qualifier for the variable must be:

- SESSION for user-defined session variables. User-defined session variables are established via the connection or signon exit routines.
- SYSIBM for built-in session variables. For a list of the built-in session variables, see “References to built-in session variables” on page 168.

default-value

Specifies a string constant that contains the value to be returned if the specified variable does not exist or is not supported by DB2. *default-value* must be a string constant that does not exceed 255 bytes.

If *default-value* is not specified and the specified user-defined session variable does not exist or the built-in session variable is not supported by DB2, an error is returned.

CAST(NULL AS VARCHAR(1))

Specifies that a null value is to be returned if the specified variable does not exist or is not supported by DB2.

The data type of the result is VARCHAR(255). The result can be null.

The CCSID of the result is the CCSID for Unicode mixed data.

Example 1: Use the GETVARIABLE function to set the value of host variable :hv1 to the name of the plan that is currently being executed. The name of the built-in session variable that contains the name of the plan is SYSIBM.PLAN_NAME.

```
SET :hv1 = GETVARIABLE('SYSIBM.PLAN_NAME');
```

If DB2 does not support the name of the session variable, an error is returned. For example, the following statement returns an error because DB2 does not support a built-in session variable that is named SYSIBM.XYZ.

```
SET :hv1 = GETVARIABLE('SYSIBM.XYZ');
```


Example 2: Use the GETVARIABLE function to set the value of host variable :hv2 to the value for the user that is defined in user-defined session variable TEST. If the session variable has not been set or cannot be found, have the function return the value 'TEST FAILED'.

```
SET :hv2 = GETVARIABLE('SESSION.TEST','TEST FAILED');
```

Example 3: Use the GETVARIABLE function to set the value of host variable :hv3 to a string representation of the SYSTEM EBCDIC CCSIDs. The name of the built-in session variable that contains the system EBCDIC CCSIDs is SYSIBM.SYSTEM_EBCDIC_CCSID.

```
SET :hv3 = GETVARIABLE('SYSIBM.SYSTEM_EBCDIC_CCSID');
```

Regardless of the setting of the field MIXED DATA on the installation panel (YES or NO), the function returns three comma-delimited values that correspond to the SBCS, MIXED, and GRAPHIC CCSIDs for the encoding scheme.

For example, if the statement were issued on a system with the field MIXED DATA on the installation panel equal to NO and the default system CCSID of 37, this string would be returned:

```
'37,65534,65534'
```

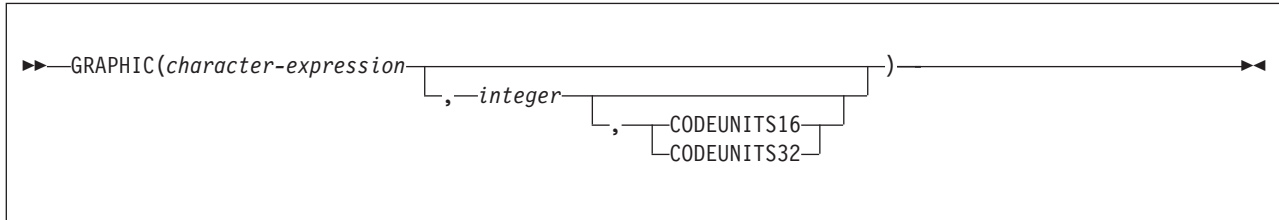
If the statement were issued on a system with the field MIXED DATA on the installation panel equal to YES and a default system CCSID of 930 (the mixed CCSID for the system), this string would be returned:

```
'290,930,300'
```

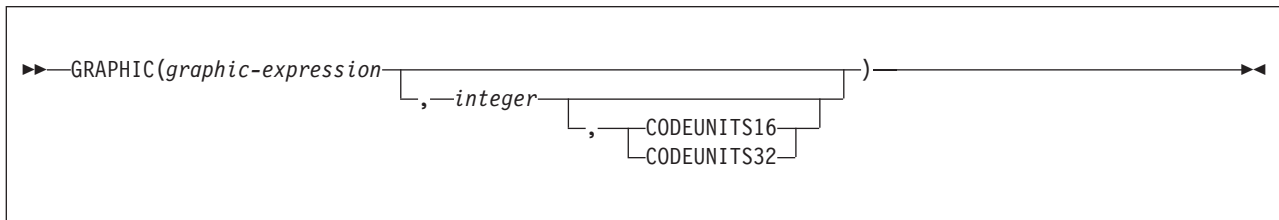
GRAPHIC

The GRAPHIC function returns a fixed-length graphic-string representation of a character string or a graphic string value, depending on the type of the first argument.

Character to Graphic:



Graphic to Graphic:



The schema is SYSIBM.

The result of the function is a fixed-length graphic string (GRAPHIC).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length attribute of the result is measured in double-byte characters because it is a graphic string.

Character to Graphic

character-expression

An expression that returns a value that is an EBCDIC-encoded or Unicode-encoded character string. It cannot be BIT data. The argument does not need to be mixed data, but any occurrences of X'0E' and X'0F' in the string must conform to the rules for EBCDIC mixed data. (See “Character strings” on page 73 for these rules.)

The value of the expression must not be an empty string if *integer* is not specified or have the value X'0E0F' if the string is an EBCDIC string.

integer

The length of the resulting fixed-length graphic string in the units that are either implicitly or explicitly specified. The value must be an integer constant between 1 and 127. If the length of *character-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If *integer* is not specified, the length of the result for an EBCDIC string is the minimum of 127 and the length attribute of *character-expression*, excluding shift characters. For a Unicode (UTF-8) string, the length is data dependent, but does not exceed 127.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

The CCSID of the result is the graphic CCSID that corresponds to the character CCSID of *character-expression*. If the input is EBCDIC and there is no system CCSID for EBCDIC GRAPHIC data, the CCSID of the result is X'FFFE'.

For EBCDIC data, each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* be the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data and its CCSID is *M*.
- The argument is SBCS data and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*. If there is no system CCSID for EBCDIC mixed data, conversion is to the coded character set that the system CCSID for SBCS data identifies.

The result is derived from *S* using the following steps:

- Each shift character (X'0E' or X'0F') is removed.
- Each double-byte character remains as is.
- Each single-byte character is replaced by a double-byte character.

The replacement for an SBCS character is the equivalent DBCS character if an equivalent exists. Otherwise, the replacement is X'FEFE'. The existence of an equivalent character depends on *M*. If there is no system CCSID for mixed data, the DBCS equivalent of X'xxxx' for EBCDIC is X'42xx', except for X'40', whose DBCS equivalent is X'4040'.

For Unicode data:

Each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the

result is derived. Let *M* be the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data, and its CCSID is *M*.
- The argument is SBCS data, and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*.

The result is derived from *S* by using the following steps:

- Each non-supplementary character is replaced by a Unicode double-byte character (a UTF-16 code point). A non-supplementary character in UTF-8 is between 1 and 3 bytes.
- Each supplementary character is replaced by a pair of Unicode double-byte characters (a pair of UTF-16 code points).

The replacement for a single-byte character is the Unicode equivalent character if an equivalent exists. Otherwise, the replacement is X'FEFE'.

Graphic to Graphic

graphic-expression

An expression that returns a value that is a graphic string. The graphic string must not be an empty string if *integer* is not specified.

integer

The length of the resulting fixed-length graphic string in the units that are either implicitly or explicitly specified. The value must be an integer constant between 1 and 127. If the length of *graphic-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If *integer* is not specified, the length of the result is the minimum of 127 and the length attribute of *graphic-expression*.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no system CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

If the length of the *graphic-expression* is greater than the specified length of the result, the result is truncated. Unless all the truncated characters are blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *graphic-expression*.

Example: Assume that MYCOL is a VARCHAR column in TABLEY. The following function returns the string in MYCOL as a fixed-length graphic string.

```
SELECT GRAPHIC(MYCOL)
FROM TABLEY;
```

HEX

The HEX function returns a hexadecimal representation of a value.

►►—HEX(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of any built-in data type that is not XML. A character or binary string must not have a maximum length greater than 16382. A graphic string must not have a maximum length greater than 8192.

The result of the function is a character string. 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 a string of hexadecimal digits. The first two represent the first byte of the argument, the next two represent the second byte of the argument, and so forth. If the argument is a datetime value, the result is the hexadecimal representation of the internal form of the argument.

If the argument is a fixed-length string and the length of the result is less than 255, the result is a fixed-length string. Otherwise, the result is a varying-length string with a length attribute that depends on the following considerations:

If the argument is not a varying-length string, the length attribute of the result string is the same as the length of the argument.

If the argument is a varying-length character or binary string, the length attribute of the result string is twice the length attribute of the argument.

If the argument is a varying-length graphic string, the length attribute of the result string is four times the length attribute of the argument.

If *expression* returns string data, the CCSID of the result is the SBCS CCSID that corresponds to the CCSID of *expression*. Otherwise, the CCSID of the result is determined from the context in which the function was invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

If the argument is a graphic string, the length of the result is four times the maximum length of the argument. Otherwise, the length of the result is twice the (maximum) length of the argument.

Example: Return the hexadecimal representation of START_RBA in the SYSIBM.SYSCOPY catalog table.

```
SELECT HEX(START_RBA) FROM SYSIBM.SYSCOPY;
```

HOUR

The HOUR function returns the hour part of a value.

►►—HOUR(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String representations of datetime values” on page 89.
- If *expression* is a number, it must be a time or timestamp duration. For the valid formats of time and timestamp durations, see “Datetime operands” on page 121.

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:

If the argument is a time, timestamp, or string representation of either, the result is the hour part of the value, which is an integer between 1 and 24.

If the argument is a time duration or timestamp duration, the result is the hour part of the value, which is an integer between -99 and +99. A nonzero result has the same sign as the argument.

Example 1: Assume that a table named CLASSES contains a row for each scheduled class. Also assume that the class starting times are in a TIME column named STARTTM. Select those rows in CLASSES that represent classes that start after the noon hour.

```
SELECT *
FROM CLASSES
WHERE HOUR(STARTTM) > 12;
```

IDENTITY_VAL_LOCAL

The IDENTITY_VAL_LOCAL function returns the most recently assigned value for an identity column.

►►—IDENTITY_VAL_LOCAL()—◄◄

The schema is SYSIBM.

The IDENTITY_VAL_LOCAL function is not deterministic.²¹ Although the function has no input parameters, the empty parentheses must be specified when the function is invoked.

The result is DECIMAL(31,0), regardless of the actual data type of the identity column to which the result value corresponds.

A *qualifying data change statement* refers to an insert operation (specified in either an INSERT statement or a MERGE statement).

The value that is returned is the value that was assigned to the identity column of the table identified in the most recent qualifying data change statement or LOAD utility operation for a table with an identity column. The insert operation has to be issued at the same level; that is, the value has to be available locally within the level at which it was assigned until replaced by the next assigned value. A new level is initiated when a trigger, function, or stored procedure is invoked. A trigger condition is at the same level as the associated triggered action.

The assigned value can be a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT) or an identity value that was generated by DB2.

Note: Use a SELECT FROM data change statement to obtain the assigned value for an identity column. See data-change-table-reference for more information.

The result can be null. The result is null in the following situations:

- When a qualifying data change statement has not been issued for a table containing an identity column at the current processing level
- When a COMMIT or ROLLBACK of a unit of work occurred since the most recent qualifying data change statement that assigned a value

The result of the function is not affected by a ROLLBACK TO SAVEPOINT statement.

Invoking the function within a qualifying data change statement: Expressions in a qualifying data change statement are evaluated before values are assigned to the target columns of the qualifying data change statement. Thus, when you invoke IDENTITY_VAL_LOCAL in a qualifying data change statement, the value that is

21. Being not deterministic affects what optimization (such as view processing and parallel processing) can be done when this function is used and in what contexts the function can be invoked. For example, the RAND function is another built-in scalar function that is not deterministic. Using functions that are not deterministic within a predicate can cause unpredictable results.

used is the most recently assigned value for an identity column from a previous qualifying data change statement. The function returns the null value if no such qualifying data change statement had been executed within the same level as the invocation of the `IDENTITY_VAL_LOCAL` function. Each qualifying data change statement that involves an `IDENTITY` column causes the identity value to be copied into connection-specific storage in DB2. Thus, the most recent identity value is used for a connection, regardless of what is happening with other concurrent user connections.

Invoking the function following a failed insert operation: The function returns an unpredictable result when it is invoked after the unsuccessful execution of a qualifying data change statement for a table with an identity column. The value might be the value that would have been returned from the function had it been invoked before the failed qualifying data change statement or the value that would have been assigned had the qualifying data change statement succeeded. The actual value returned depends on the point of failure and is therefore unpredictable.

Invoking the function within the `SELECT` statement of a cursor: Because the results of the `IDENTITY_VAL_LOCAL` function are not deterministic, the result of an invocation of the `IDENTITY_VAL_LOCAL` function from within the `SELECT` statement of a cursor can vary for each `FETCH` statement.

Invoking the function within the trigger condition of an insert trigger: The result of invoking the `IDENTITY_VAL_LOCAL` function from within the condition of an insert trigger is the null value.

Invoking the function within a triggered action of an insert trigger: Multiple before or after insert triggers can exist for a table. In such cases, each trigger is processed separately, and identity values generated by SQL statements issued within a triggered action are not available to other triggered actions using the `IDENTITY_VAL_LOCAL` function. This is the case even though the multiple triggered actions are conceptually defined at the same level.

Do not use the `IDENTITY_VAL_LOCAL` function in the triggered action of a before insert trigger. The result of invoking the `IDENTITY_VAL_LOCAL` function from within the triggered action of a before insert trigger is the null value.

The value for the identity column of the table for which the trigger is defined cannot be obtained by invoking the `IDENTITY_VAL_LOCAL` function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.

The result of invoking the `IDENTITY_VAL_LOCAL` function in the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent qualifying data change statement. That statement is the one invoked in the same triggered action that had a qualifying data change statement for a table containing an identity column. If a qualifying data change statement for a table containing an identity column was not executed within the same triggered action before invoking the `IDENTITY_VAL_LOCAL` function, then the function returns a null value.

Invoking the function following an insert operation with triggered actions: The result of invoking the function after an insert that activates triggers is the value actually assigned to the identity column (that is, the value that would be returned

on a subsequent SELECT statement). This value is not necessarily the value provided in the qualifying data change statement or a value generated by DB2. The assigned value could be a value that was specified in a SET transition variable statement within the triggered action of a before insert trigger for a trigger transition variable associated with the identity column.

Scope of IDENTITY_VAL_LOCAL: The IDENTITY_VAL_LOCAL value persists until the next insert in the current session into a table that has an identity column defined on it, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements for local applications. The IDENTITY_VAL_LOCAL value cannot be directly set and is a result of inserting a row into a table. Client applications or middleware products that save the state of a session and then restore the state of a session for subsequent processing are not able to restore the IDENTITY_VAL_LOCAL value. In these situations, the availability of the IDENTITY_VAL_LOCAL value should only be relied on until the end of the transaction. Examples of where this type of situation can occur include applications that do the following:

- use XA protocols
- use connection pooling
- use the connection concentrator
- use Sysplex workload balancing
- connect to a z/OS server that uses DDF inactive threads

When there is a need to preserve the value associated with IDENTITY_VAL_LOCAL across transaction boundaries for distributed applications, define the cursors as WITH HOLD, or specify the bind option KEEP DYNAMIC(YES) to prevent the server thread from being pooled.

Example 1: Set the variable IVAR to the value assigned to the identity column in the EMPLOYEE table. The value returned from the function in the VALUES statement should be 1.

```
CREATE TABLE EMPLOYEE
(EMPNO      INTEGER GENERATED ALWAYS AS IDENTITY,
 NAME       CHAR(30),
 SALARY     DECIMAL(5,2),
 DEPTNO     SMALLINT);
INSERT INTO EMPLOYEE
(NAME, SALARY, DEPTNO)
VALUES ('Rupert', 989.99, 50);
VALUES IDENTITY_VAL_LOCAL() INTO :IVAR;
```

Example 2: Assume two tables, T1 and T2, have an identity column named C1. DB2 generates values 1, 2, 3, . . . for the C1 column in table T1, and values 10, 11, 12, . . . for the C1 column in table T2.

```
CREATE TABLE T1 (C1 SMALLINT GENERATED ALWAYS AS IDENTITY,
                  C2 SMALLINT );
CREATE TABLE T2 (C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY
                  (START WITH 10),
                  C2 SMALLINT );
INSERT INTO T1 (C2) VALUES (5);
INSERT INTO T1 (C2) VALUES (5);
SELECT * FROM T1;
      C1          C2
-----
          1          5
          2          5
VALUES IDENTITY_VAL_LOCAL() INTO :IVAR;
```

At this point, the `IDENTITY_VAL_LOCAL` function would return a value of 2 in `IVAR`. The following `INSERT` statement inserts a single row into `T2` where column `C2` gets a value of 2 from the `IDENTITY_VAL_LOCAL` function

```
INSERT INTO T2 (C2) VALUES (IDENTITY_VAL_LOCAL());
SELECT * FROM T2
WHERE C1 = DECIMAL(IDENTITY_VAL_LOCAL(),15,0);
      C1                                C2
-----
10                                2
```

Invoking the `IDENTITY_VAL_LOCAL` function after this insert would result in a value of 10, which is the value generated by `DB2` for column `C1` of `T2`. Assume another single row is inserted into `T2`. For the following `INSERT` statement, `DB2` assigns a value of 13 to identity column `C1` and gives `C2` a value of 10 from `IDENTITY_VAL_LOCAL`. Thus, `C2` is given the last identity value that was inserted into `T2`.

```
INSERT INTO T2 (C2, C1) VALUES (IDENTITY_VAL_LOCAL(), 13);
```

Example 3: The `IDENTITY_VAL_LOCAL` function can also be invoked in an `INSERT` statement that both invokes the `IDENTITY_VAL_LOCAL` function and causes a new value for an identity column to be assigned. The next value to be returned is thus established when the `IDENTITY_VAL_LOCAL` function is invoked after the `INSERT` statement completes. For example, consider the following table definition:

```
CREATE TABLE T1 (C1 SMALLINT GENERATED BY DEFAULT AS IDENTITY,
                  C2 SMALLINT);
```

For the following `INSERT` statement, specify a value of 25 for the `C2` column, and `DB2` generates a value of 1 for `C1`, the identity column. This establishes 1 as the value that will be returned on the next invocation of the `IDENTITY_VAL_LOCAL` function.

```
INSERT INTO T1 (C2) VALUES (25);
```

In the following `INSERT` statement, the `IDENTITY_VAL_LOCAL` function is invoked to provide a value for the `C2` column. A value of 1 (the identity value assigned to the `C1` column of the first row) is assigned to the `C2` column, and `DB2` generates a value of 2 for `C1`, the identity column. This establishes 2 as the value that will be returned on the next invocation of the `IDENTITY_VAL_LOCAL` function.

```
INSERT INTO T1 (C2) VALUES (IDENTITY_VAL_LOCAL());
```

In the following `INSERT` statement, the `IDENTITY_VAL_LOCAL` function is again invoked to provide a value for the `C2` column, and the user provides a value of 11 for `C1`, the identity column. A value of 2 (the identity value assigned to the `C1` column of the second row) is assigned to the `C2` column. The assignment of 11 to `C1` establishes 11 as the value that will be returned on the next invocation of the `IDENTITY_VAL_LOCAL` function.

```
INSERT INTO T1 (C2, C1) VALUES (IDENTITY_VAL_LOCAL(), 11);
```

After the 3 `INSERT` statements have been processed, table `T1` contains the following:

```
SELECT * FROM T1;
C1      C2
-----
```

1	25
2	1
11	2

The contents of T1 illustrate that the expressions in the VALUES clause are evaluated before the assignments for the columns of the INSERT statement. Thus, an invocation of an IDENTITY_VAL_LOCAL function invoked from a VALUES clause of an INSERT statement uses the most recently assigned value for an identity column in a previous INSERT statement.

IFNULL

The IFNULL function returns the first nonnull expression.

►►—IFNULL(*expression*,*expression*)—◄◄

The schema is SYSIBM.

IFNULL is identical to the COALESCE scalar function except that IFNULL is limited to two arguments instead of multiple arguments. For a description, see “COALESCE” on page 315.

Example: For all the rows in sample table DSN8910.EMP, select the employee number and salary. If the salary is missing (is null), have the value 0 returned.

```
SELECT EMPNO, IFNULL(SALARY,0)
FROM DSN8910.EMP;
```

INSERT

The INSERT function returns a string where, beginning at *start* in *source-string*, *length* characters have been deleted and *insert-string* has been inserted.

```
►►—INSERT—(—source-string—,—start—,—length—,—insert-string—,—)————►
                                     |
                                     | CODEUNITS16
                                     | CODEUNITS32
                                     | OCTETS
```

The schema is SYSIBM.

The INSERT function returns a string where *length* characters have been deleted from *source-string*, beginning at *start*, and where *insert-string* has been inserted into *source-string*, beginning at *start*.

source-string

An expression that specifies the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB. The actual length of the string must be greater than or equal to 1 and less than or equal to 32704 bytes.

start

An expression that returns an integer. The integer specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The value of the integer must be in the range of 1 to the length of *source-string* plus one. If OCTETS is specified and the result is graphic data, the value must be an odd value between 1 and twice the length of *source-string* plus one.

length

An expression that specifies the length of the string to replace in *source-string* starting at *start*. *length* must be an expression that returns a value of the built-in INTEGER data type. *length* is expressed in the string unit specified, and the value must be in the range of 0 to the length of *source-string*. If OCTETS is specified and the result is graphic data, *length* must be even and be between 0 and twice the length of *source-string*. Not specifying *length* is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying *length* is equivalent to specifying a value of 2.

insert-string

An expression that specifies the string to be inserted into the source string, starting at the position identified by *start*. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB.

source-string and *insert-string* must have compatible data types.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the units that are used to express *start* and *length*. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used:

- If the result is a graphic string, a string unit is two bytes. For ASCII and EBCDIC data, this corresponds to a double byte character. For Unicode, this corresponds to a UTF-16 code point.
- Otherwise, a string unit is a byte.

CODEUNITS16

Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *start* and *length* are expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

If *source-string* and *insert-string* have different CCSID sets, *insert-string* (the string to be inserted) is converted to the CCSID of *source-string* (the source string).

The encoding scheme of the result is the same as *source-string*. The data type of the result of the function depends on the data type of *source-string* and *insert-string*:

- VARCHAR if *source-string* is a character string. The CCSID of the result depends on the arguments:
 - If either *source-string* or *insert-string* is character bit data, the result is bit data.
 - If *source-string* is SBCS Unicode data and *insert-string* is not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
 - If both *source-string* and *insert-string* are SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.
 - Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *source-string*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *source-string*.
- VARGRAPHIC if *source-string* is a graphic. The CCSID of the result is the same as the CCSID of *source-string*.
- VARBINARY if *source-string* and *insert-string* are both binary strings.

The length attribute of the result depends on the arguments:

- If *start* and *length* are constants, the length attribute of the result is:

$$L1 - \text{MIN}((L1 - V2 + 1), V3) + L4$$

where:

L1 is the length attribute of *source-string*

V2 is the value of *start*

V3 is the value of *length*

L4 is the length attribute of *insert-string*

- Otherwise, the length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*. In this case, the length attribute of *source-string* plus the length attribute of *insert-string* must not exceed 32704 for a VARCHAR result or 16352 for a VARGRAPHIC result.

If CODEUNITS16 or CODEUNITS32 is specified, the insert operation is performed on a Unicode version of the data. If needed, the data is converted to an intermediate form in order to evaluate the function. If an intermediate form is used, the actual length of the result depends on the original data (*source-string* and *insert-string*), and the representation of that data in Unicode. See “Determining the length attribute of the final result” on page 79 for more information on how to calculate the length attribute of the result string.

If CODEUNITS16 or CODEUNITS32 are not specified, the actual length of the result is:

$$A1 - \text{MIN}((A1 - V2 + 1), V3) + A4$$

where:

A1 is the actual length of *source-string*

V2 is the value of *start*

V3 is the value of *length*

A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error occurs.

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

Example 1: The following example shows how the string 'INSERTING' can be changed into other strings. The use of the CHAR function limits the length of the resulting string to 10 bytes.

```
SELECT CHAR(INSERT('INSERTING',4,2,'IS'),10),
       CHAR(INSERT('INSERTING',4,0,'IS'),10),
       CHAR(INSERT('INSERTING',4,2,''),10)
FROM SYSIBM.SYSDUMMY1;
```

This example returns 'INSISTING ', 'INSISERTIN', and 'INSTING '.

Example 2: The previous example demonstrated how to insert text into the middle of some text. This example shows how to insert text before some text by using 1 as the starting point (*start*).

```
SELECT CHAR(INSERT('INSERTING',1,0,'XX'),10),
       CHAR(INSERT('INSERTING',1,1,'XX'),10),
       CHAR(INSERT('INSERTING',1,2,'XX'),10),
       CHAR(INSERT('INSERTING',1,3,'XX'),10)
FROM SYSIBM.SYSDUMMY1;
```

This example returns 'XXINSERTIN', 'XXNSERTING', 'XXSERTING ', and 'XXERTING '.

Example 3: The following example shows how to insert text after some text. Add 'XX' at the end of string 'ABCABC'. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT CHAR(INSERT('ABCABC',7,0,'XX'),10)
FROM SYSIBM.SYSDUMMY1;
```

This example returns 'ABCABCXX '.

Example 4: The following example shows how the string 'Hegelstraße' can be changed to 'Hegelstrasse'.


```
SELECT VARCHAR(INSERT('Hegelstraße',10,1,'ss'),15)
FROM SYSIBM.SYSDUMMY1;
```

This example returns 'Hegelstrasse'.

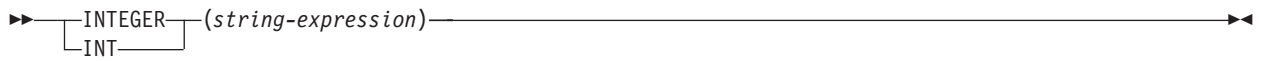
INTEGER or INT

The INTEGER function returns an integer representation of either a number or a character string or graphic string representation of an integer.

Numeric to Integer:



String to Integer:



The schema is SYSIBM.

Numeric to Integer

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of large integers, an error occurs. The fractional part of the argument is truncated.

String to Integer

string-expression

An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from `CAST(string-expression AS INTEGER)`. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer constant. If the whole part of the argument is not within the range of large integers, an error is returned.

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

Recommendation: To increase the portability of applications, use the CAST specification. For more information, see “CAST specification” on page 202.

Example 1: Using sample table DSN8910.EMP, find the average salary of the employees in department A00, rounding the result to the nearest dollar.

```
SELECT INTEGER(AVG(SALARY)+.5)
FROM DSN8910.EMP
WHERE WORKDEPT = 'A00';
```

Example 2: Using sample table DSN8910.EMP, select the EMPNO column, which is defined as CHAR(6), in integer form.

```
SELECT INTEGER(EMPNO)
FROM DSN8910.EMP;
```

JULIAN_DAY

The JULIAN_DAY function returns an integer value that represents a number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date that is specified in the argument.

►►—JULIAN_DAY(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns one of the following data types: a date, a timestamp, or a valid string representation of a date or timestamp. An argument with a character string data type must not be a CLOB. An argument with a graphic string data type must not be a DBCLOB. A string argument must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 1: Using sample table DSN8910.EMP, set the integer host variable JDAY to the Julian day of the day that Christine Haas (EMPNO = '000010') was employed (HIREDATE = '1965-01-01').

```
SELECT JULIAN_DAY(HIREDATE)
      INTO :JDAY
FROM DSN8910.EMP
WHERE EMPNO = '000010';
```

The result is that JDAY is set to 2438762.

Example 2: Set integer host variable JDAY to the Julian day for January 1, 1998.

```
SELECT JULIAN_DAY('1998-01-01')
      INTO :JDAY
FROM SYSIBM.SYSDUMMY1;
```

The result is that JDAY is set to 2450815.

LAST_DAY

The LAST_DAY scalar function returns a date that represents the last day of the month of the date argument.

►►—LAST_DAY(*expression*)—◄◄

The schema is SYSIBM.

expression

An expression that specifies the starting date. The expression must return a value of one of the following data types:

- a date
- a timestamp
- a valid string representation of a date or timestamp

An argument with a character string data type must not be a CLOB. An argument with a graphic string data type must not be a DBCLOB. A string argument must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

The result of the function is a DATE.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

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

Example 1: Set the host variable *END_OF_MONTH* with the last day of the current month.

```
SET :END_OF_MONTH = LAST_DAY(CURRENT_DATE);
```

The host variable *END_OF_MONTH* is set with the value representing the end of the current month. If the current day is 2000-02-10, *END_OF_MONTH* is set to 2000-02-29.

Example 2: Set the host variable *END_OF_MONTH* with the last day of the month in EUR format for the given date.

```
SET :END_OF_MONTH = CHAR(LAST_DAY('1965-07-07'), EUR);
```

The host variable *END_OF_MONTH* is set with the value '31.07.1965'.

LCASE

The LCASE function returns a string in which all the characters are converted to lowercase characters.

I ►►—LCASE(*string-expression* [,—*locale-name*] [,—*integer*])—►◄

The schema is SYSIBM.

The LCASE function is identical to the LOWER function. For more information, see “LOWER” on page 410.

LEFT

The LEFT function returns a string that consists of the specified number of leftmost bytes of the specified string units.

Character string:

►►—LEFT(*character-expression*,*length*—, —CODEUNITS16—
—CODEUNITS32—
—OCTETS—)——►►

Graphic string:

►►—LEFT(*graphic-expression*,*length*—, —CODEUNITS16—
—CODEUNITS32—)——►►

Binary string:

►►—LEFT(*binary-expression*,*length*)——►►

The schema is SYSIBM.

The LEFT function returns the leftmost string of *character-expression*, *graphic-expression*, or *binary-expression* consisting of length of the string units that are specified implicitly or explicitly.

Character string:

character-expression

An expression that specifies the string from which the result is derived. The string must be a character string. A substring of *character-expression* is zero or more contiguous code points of *character-expression*.

The string can contain mixed data. Depending on the units that are specified to evaluate the function, the result is not necessarily a properly formed mixed data character string.

length

An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *character-expression*, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the unit that is used to express *length*. If *character-expression* is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified.

CODEUNITS16

Specifies that *length* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *length* is expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *length* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

Graphic string:

graphic-expression

An expression that specifies the string from which the result is derived. The string must be a graphic string. A substring of *graphic-expression* is zero or more contiguous code points of *graphic-expression*.

length

An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *graphic-expression*, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *length*.

CODEUNITS16

Specifies that *length* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *length* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

Binary string:

binary-expression

An expression that specifies the string from which the result is derived. The string must be a binary string. A substring of *binary-expression* is zero or more contiguous code points of *binary-expression*.

length

An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *binary-expression*, expressed in the units that are either implicitly or explicitly specified.

The *character-expression*, *graphic-expression*, or *binary-expression* is effectively padded on the right with the necessary number of padding characters so that the specified substring of the expression always exists. The encoding scheme of the data determines the padding character:

- For ASCII SBCS data or ASCII mixed data, the padding character is X'20'.

- For ASCII DBCS data, the padding character depends on the CCSID; for example, for Japanese (CCSID 301) the padding character is X'8140', while for simplified Chinese it is X'A1A1'.
- For EBCDIC SBCS data or EBCDIC mixed data, the padding character is X'40'.
- For EBCDIC DBCS data, the padding character is X'4040'.
- For Unicode SBCS data or UTF-8 (Unicode mixed data), the padding character is X'20'.
- For UTF-16 (Unicode DBCS) data, the padding character is X'0020'.
- For binary data, the padding character is X'00'.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of the first expression and a data type that depends on the data type of the expression:

- VARCHAR if *character-expression* is CHAR or VARCHAR
- CLOB if *character-expression* is CLOB
- VARGRAPHIC if *graphic-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *graphic-expression* is DBCLOB
- VARBINARY if *binary-expression* is BINARY or VARBINARY
- BLOB if *binary-expression* is BLOB

The actual length of the result is determined from *length*.

If any argument of the function can be null, the result can be null; if any argument is null, the result is the null value.

The CCSID of the result is the same as that of the first expression.

Example 1: Assume that host variable *ALPHA* has a value of 'ABCDEF'. The following statement returns 'ABC', which are the three leftmost characters in *ALPHA*:

```
SELECT LEFT(:ALPHA,3)
FROM SYSIBM.SYSDUMMY1;
```

Example 2: Assume that host variable *NAME*, which is defined as VARCHAR(50), has a value of 'KATIE AUSTIN' and the integer host variable *FIRSTNAME_LEN* has a value of 5. The following statement returns the value 'KATIE':

```
SELECT LEFT(:NAME, :FIRSTNAME_LEN)
FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following statement returns a zero length string.

```
SELECT LEFT('ABCABC',0)
FROM SYSIBM.SYSDUMMY1;
```

Example 4: The *FIRSTNAME* column in sample EMP table is defined as VARCHAR(12). Find the first name for an employee whose last name is 'BROWN' and return the first name in a 10-byte string.

```
SELECT LEFT(FIRSTNAME,10)
FROM DSN8910.EMP
WHERE LASTNAME='BROWN';
```

This function returns a VARCHAR(10) string that has the value of 'DAVID' followed by 5 blank characters.

Example 5: *FIRSTNAME* is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When *FIRSTNAME* has this value:

	Function ...	Returns ...
	LEFT(FIRSTNAME,2,CODEUNITS32)	'Jü' -- x'4AC3BC'
	LEFT(FIRSTNAME,2,CODEUNITS16)	'Jü' -- x'4AC3BC'
	LEFT(FIRSTNAME,2,OCTETS)	'J ' -- x'4A20' a truncated string

LENGTH

The LENGTH function returns the length of a value.

►►—LENGTH(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of any built-in data type that is not XML.

The result of the function is a large integer. 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 length of the argument. The length does not include the null indicator byte of column arguments that allow null values. The length of strings includes blanks. The length of a varying-length string is the actual length, not the maximum length.

The length of a graphic string is the number of double-byte characters. Unicode UTF-16 data is treated as graphic data; a UTF-16 supplementary character takes two DBCS characters to represent and as such is counted as two DBCS characters.

The length of all other values is the number of bytes used to represent the value:

- 2 for small integer
- 4 for large integer
- 8 for big integer
- The integer part of $(p/2)+1$ for decimal numbers with precision p
- 16 for DECFLOAT(34)
- 8 for DECFLOAT(16)
- 4 for single precision floating-point
- 8 for double precision floating-point
- The length of the string for strings
- 4 for DATE
- 3 for TIME
- 10 for TIMESTAMP
- The length of the row ID

Example 1: Assume that FIRSTNME is a VARCHAR(12) column that contains 'ETHEL' for employee 280. The following query returns the value 5:

```
SELECT LENGTH(FIRSTNME)
FROM DSN8910.EMP
WHERE EMPNO = '000280';
```

Example 2: Assume that HIREDATE is a column of data type DATE. Then, regardless of value the following statement returns the value 4:

```
LENGTH(HIREDATE)
```

And the following function returns the value 10:

```
LENGTH(CHAR(HIREDATE, EUR))
```

LN

The LN function returns the natural logarithm of the argument. The LN and EXP functions are inverse operations.

►►—LN(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

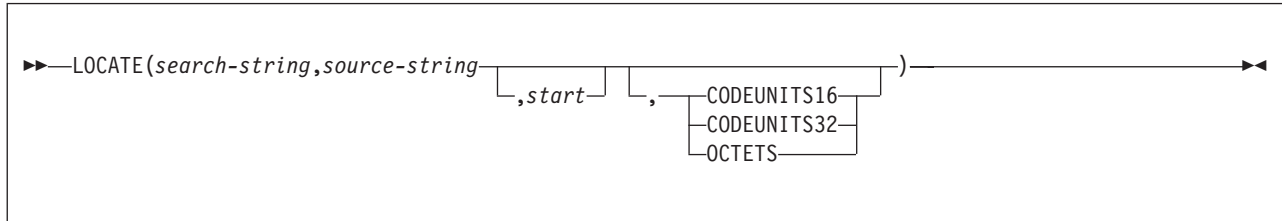
LOG is a synonym for LN.

Example: Assume that host variable NATLOG is DECIMAL(4,2) with a value of 31.62. The following statement returns a double precision floating-point number with an approximate value of 3.45:

```
SELECT LN(:NATLOG)
FROM SYSIBM.SYSDUMMY1;
```

LOCATE

The LOCATE function returns the position at which the first occurrence of an argument starts within another argument.



The schema is SYSIBM.

The LOCATE function returns the starting position of *search-string* within *source-string*. If *search-string* is not found and neither argument is null, the result is zero. If *search-string* is found, the result is a number from 1 to the actual length of *source-string*. If the optional *start* is specified, it indicates the character position in *source-string* at which the search is to begin. An optional string unit can be specified to indicate in what units the start and result of the function are expressed.

search-string

An expression that specifies the string that is to be searched for. *search-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

source-string

An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

start

An expression that specifies the position within *search-string* where the search is to start. *start* is expressed in the specified string unit and must return an integer value that is greater than zero.

If *start* is specified, the LOCATE function is similar to the following POSITION function, where *string-units* is CODEUNITS16, CODEUNITS32, or OCTETS:

```
POSITION(search-string,  
        SUBSTRING(source-string, start, string-units)) + start - 1
```

If *start* is not specified, the search begins at the first position of *source-string* and the LOCATE function is similar to the following POSITION function, where *string-units* is CODEUNITS16, CODEUNITS32, or OCTETS:

```
POSITION(search-string, source-string, string-units)
```

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit that is used to express *start* and the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

CODEUNITS16

Specifies that *start* and the result are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *start* and the result are expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *start* and the result are expressed in terms of bytes.

If a string unit is not explicitly specified, the data type of the result determines the string unit that is used. If the result is graphic data, *start* and the returned position are expressed in two-byte units; otherwise, they are expressed in bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

The first and second arguments must have compatible string types. For more information on compatibility, see “Conversion rules for operations that combine strings” on page 122.

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

For more information about LOCATE, see the description of “POSITION” on page 466.

Example 1: Find the location of the first occurrence of the character 'N' in the string 'DINING'.

```
SELECT LOCATE('N', 'DINING')  
FROM SYSIBM.SYSDUMMY1;
```

The result is the value 3.

Example 2: For all the rows in the table named IN_TRAY, select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD' within the NOTE_TEXT column.

```
SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
FROM IN_TRAY
WHERE LOCATE('GOOD', NOTE_TEXT) <> 0;
```

Example 3: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = LOCATE('ß','Jürgen lives on Hegelstraße',1,CODEUNITS32);
```

The value of host variable LOCATION is set to 26.

Example 4: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS16 units, within the string.

```
SET :LOCATION = LOCATE('ß','Jürgen lives on Hegelstraße',1,CODEUNITS16);
```

The value of host variable LOCATION is set to 26.

Example 5: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = LOCATE('ß','Jürgen lives on Hegelstraße',1,OCTETS);
```

The value of host variable LOCATION is set to 27.

Related reference

“LOCATE_IN_STRING” on page 406

“POSITION” on page 466

“POSSTR” on page 469

LOCATE_IN_STRING

The LOCATE_IN_STRING function returns the position at which an argument starts within a specified string.

►►—LOCATE_IN_STRING(*source-string*,*search-string* [,*start* [,*instance* [,*CODEUNITS16* [,*CODEUNITS32* [,*OCTETS*]]]]]])—►►

The schema is SYSIBM.

The LOCATE_IN_STRING function returns the starting position of a string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*.

If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin. If *start* is specified, an instance number can also be specified. The instance argument determines the position of a specific occurrence of *search-string* within *source-string*. An optional string unit can be specified to indicate in what units the start and result of the function are expressed. Each unique instance can include any of the characters in a previous instance, but not all characters in a previous instance.

If *search-string* has a length of zero, the result returned by the function is 1. If *source-string* has a length of zero, the result returned by the function is 0. If neither condition exists, and if the value of *search-string* is equal to an identical length of a substring of contiguous positions within the value of *source-string*, the result returned by the function is the starting position of that substring within the *source-string* value. Otherwise, the result returned by the function is 0.

source-string

An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

search-string

An expression that specifies the string that is the object of the search. *search-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes. The expression can be specified by any of the following:

- A constant

- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

start

An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value that is a built-in INTEGER or SMALLINT data type.

If the value of the integer is greater than zero, the search begins at *start* and continues for each position to the end of the string. If the value of the integer is less than zero, the search begins at the LENGTH(*source-string*) + *start* + 1 and continues for each position to the beginning of the string.

If *start* is not specified, the default is 1. If the value of the integer is zero, an error is returned.

instance

An expression that specifies which instance of *search-string* to search for within *source-string*. The expression must return a value that is a built-in INTEGER or SMALLINT data type. If *instance* is not specified, the default is 1. The value of the integer must be greater than or equal to one.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit that is used to express *start* and the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

CODEUNITS16

Specifies that *start* and the result are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *start* and the result are expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *start* and the result are expressed in terms of bytes.

If a string unit is not explicitly specified, the data type of the result determines the string unit that is used. If the result is graphic data, *start* and the returned position are expressed in two-byte units; otherwise, they are expressed in bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

The first and second arguments must have compatible string types. For more information on compatibility, see “Conversion rules for operations that combine strings” on page 122.

The result of the function is a large integer. The result is the starting position of the instance of *search-string* within *source-string*. The value is relative to the beginning of the string (regardless of the specification of *start*). If any argument can be null, the result can be null; if any argument is null, the result is the null value.

INSTR can be used as a synonym for LOCATE_IN_STRING.

Example 1: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable *POSITION* with the position, as measured in CODEUNITS32 units, within the string.

```
SET :POSITION = LOCATE_IN_STRING('Jürgen lives on Hegelstraße',
                                  'ß',1,CODEUNITS32);
```

The value of host variable *POSITION* is set to 26.

Example 2: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße' by searching from the end of the string, and set the host variable *POSITION* with the position, as measured in CODEUNITS32 units, within the string.

```
SET :POSITION = LOCATE_IN_STRING('Jürgen lives on Hegelstraße',
                                  'ß',-1,CODEUNITS32);
```

The value of host variable *POSITION* is set to 26.

Example 3: Find the position of an occurrence of the character 'N' in the string 'WINNING' by searching from the start of the string as measured in bytes, within the string.

```
SELECT LOCATE_IN_STRING('WINNING','N',1,3,OCTETS),
       LOCATE_IN_STRING('WINNING','N',3,2,OCTETS),
       LOCATE_IN_STRING('WINNING','N',3,3,OCTETS),
       LOCATE_IN_STRING('WINNING','N',-1,3,OCTETS),
       LOCATE_IN_STRING('WINNING','N',-3,2,OCTETS),
       LOCATE_IN_STRING('WINNING','N',-3,3,OCTETS)
FROM SYSIBM.SYSDUMMY1;
```

Returns the values:

```
6 4 6 3 3 0
```

Related reference

“LOCATE” on page 403

“POSITION” on page 466

“POSSTR” on page 469

LOG10

The LOG10 function returns the common logarithm (base 10) of a number.

►►—LOG10(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument is an expression that returns the value of any built-in numeric data
| type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HLOG is an INTEGER with a value of 100. The following statement returns a double precision floating-point number with an approximate value of 2:

```
SELECT LOG10(:HLOG)
FROM SYSIBM.SYSDUMMY1;
```

LOWER

The LOWER function returns a string in which all the characters are converted to lowercase characters.

```
►►—LOWER(string-expression [ ,—locale-name ] [ ,—integer ] )—►►
```

The schema is SYSIBM.

string-expression

An expression that specifies the string to be converted. The string must be a character or graphic string. A character string argument must not be a CLOB, and a graphic string argument must not be a DBCLOB.

locale-name

A string constant or a string host variable other than a CLOB or DBCLOB that specifies a valid locale name. If *locale-name* is not in EBCDIC, it is converted to EBCDIC. The length of *locale-name* must be between 1 and 255 bytes of the EBCDIC representation. The value of *locale-name* is not case sensitive and must be a valid locale. For information on locales and their naming conventions, see *z/OS C/C++ Programming Guide*. Some examples of locales include:

```
Fr_BE
Fr_FR@EURO
En_US
Ja_JP
```

For EBCDIC and ASCII data, there are two options for *local-name*:

- blank — *string-expression* must not specify a graphic string. For a character string, characters A-Z are converted to a-z and characters with diacritical marks are not converted. If the string contains MIXED or DBCS characters, full-width Latin uppercase letters A-Z are converted to full-width lowercase letters a-z. This is the default value specified in the LOCALE LC_CTYPE on the installation panel DSNTIPF.
- a local — local specific casing will be done using the "LOCAL" casing capabilities, as specified during installation

For optimal performance, use blank for *locale-name* unless your data must be interpreted using the rules provided for specific locales.

For Unicode data, there are three options for *locale-name*:

- blank — simple casting on A-Z, a-z, and full-width Latin lowercase letters a-z and full-width Latin uppercase letters A-Z. Characters with diacritics are not affected.
- "UNI" — If the value "UNI" is specified, casting will use both the "NORMAL" and "SPECIAL" casing capabilities as described in *z/OS Support for Unicode: Using Conversion Services*.
- a locale — In this case, locale specific casing will be performed using the "LOCALE" casing capabilities as described in *z/OS Support for Unicode: Using Conversion Services*.

The value of the host variable must not be null. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value. The locale name must be:

- left justified within the host variable
- padded on the right with blanks if its length is less than that of the host variable and the host variable is in fixed length character or graphic data type

If *locale-name* is not specified, the locale is determined by special register CURRENT LOCALE LC_CTYPE. For information about the special register, see “CURRENT LOCALE LC_CTYPE” on page 140. However, if an index references the LOWER function, the local is determined as follows (in order) to determine if the index can be used:

- At prepare time — using the value in the CURRENT LOCALE LC_CTYPE special register
- At bind time — using the value in the LOCALE LC_CTYPE field on installation panel DSNTPF

If the index is chosen in the access path, the locale in the CURRENT LOCALE LC_CTYPE special register must remain the same at run time, and prepare or bind time. To avoid this dependence, don't omit *locale-name*.

If the LOWER function is referenced in an index that is based on an expression, *locale-name* must be specified. See the examples section for an example of how the index can be utilized in a query.

integer

An integer value that specifies the length attribute of the result. If specified, *integer* must be an integer constant between 1 and 32704 bytes in the representation of the encoding scheme of *string-expression*.

If *integer* is not specified, the length attribute of the result is the same as the length of *string-expression*.

For Unicode data, usage of the LOWER function can result in expansion if certain characters are processed. For example, LOWER ('İ') —UX'00CC'— will result in UX'006903070300' (if the LT_LT locale is in effect at the time). You should ensure that the result length is large enough to contain the result of the expression.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

LCASE is a synonym for LOWER.

Example 1: Return the characters in the value of host variable NAME in lowercase. NAME has a data type of VARCHAR(30) and a value of 'Christine Smith'. Assume that the locale in effect is blank.

```
SELECT LCASE(:NAME)
FROM SYSIBM.SYSDUMMY1;
```

The result is the value 'christine smith'.

Example 2: Return the lowercase of 'İ'. Assume that the locale in effect is LT_LT.

```
SELECT LOWER('İ')
FROM SYSIBM.SYSDUMMY1;
```

This would result in an error because of the expansion that occurs when certain Unicode characters are processed. To avoid the error, you would need to use the following statement instead:

```
SELECT LOWER(VARCHAR('İ', 3))
FROM SYSIBM.SYSDUMMYU;
```

The result of the preceding statement is the value UX'006903070300'.

Example 3: Create an index EMPLOYEE_NAME_LOWER for table EMPLOYEE based on built-in function LOWER with locale name 'LT_LT'.

```
CREATE INDEX EMPLOYEE_NAME_LOWER
ON EMPLOYEE (LOWER(LASTNAME, 'LT_LT', 60),
             LOWER(FIRSTNAME, 'LT_LT', 60),
             ID);
```

Example 4: Create an index LNAME for table T1 based on the LOWER function with the default local value, ' '. Then specify the same expression in a query.

```
CREATE INDEX LNAME
ON T1 (LOWER(LASTNAME, ' '));

SELECT LOWER(LASTNAME, ' ')
FROM T1
WHERE LOWER(LASTNAME, ' ') = 'smith';
```

Example 5: Create an index LNAME that is based on the LOWER function with a locale name 'FR_CA' for the table T1. Then specify the same expression in a query except *locale-name* is omitted.

```
CREATE INDEX LNAME
ON T1 (LOWER(LASTNAME, 'FR_CA'));
```

If the query is a dynamic statement and the CURRENT LOCALE LC_CTYPE special register contains 'FR_CA':

```
SELECT LASTNAME
FROM T1
WHERE LOWER(LASTNAME)='smith';
```

At prepare time, locale 'FR_CA' in CURRENT LOCALE LC_CTYPE is used for LOWER(LASTNAME) in the predicate to determine whether index LNAME can be used for index access. If index LNAME is used in access path selection, at run time, the locale in CURRENT LOCALE LC_CTYPE must remain the same.

If the query is a static statement and locale 'FR_CA' has been set on the LOCALE LC_CTYPE field of installation panel DSNTIPF:

```
SELECT LASTNAME
FROM T1
WHERE LOWER(LASTNAME)='smith';
```

At bind time, local 'FR_CA' in the LOCALE LC_CTYPE file of installation panel DSNTIPF is used for LOWER(LASTNAME) in the predicate to determine whether index LNAME is used for index access. If index LNAME is chosen in access path selection, the locale in the CURRENT LOCALE LC_CTYPE special register must contain 'FR_CA'.

LPAD

The LPAD function returns a string that is composed of *string-expression* that is padded on the left, with *pad* or blanks. The LPAD function treats leading or trailing blanks in *string-expression* as significant.

The diagram shows the function syntax LPAD(string-expression, integer [, pad]) enclosed in a box. A horizontal line with arrowheads at both ends passes through the function name and the first two arguments. A bracket is positioned below the third argument, which is enclosed in square brackets.

Padding occurs only if the actual length of *string-expression* is less than *integer*, and if *pad* is not an empty string.

The schema is SYSIBM.

string-expression

An expression that specifies the source string. The expression must return a value that is a built-in string data type that is not a LOB.

integer

An integer constant that specifies the length of the result. The value must be zero or a positive integer that is less than or equal to *n*, where *n* is 32704 if *string-expression* is a character or binary string, or where *n* is 16352 if *string-expression* is a graphic string.

pad

An expression that specifies the string with which to pad. The expression must return a value that is a built-in string data type that is not a LOB. If *pad* is not specified, the pad character is determined as follows:

- SBCS blank character if *string-expression* is a character string.
- DBCS blank character if *string-expression* is a graphic string.
- Hexadecimal zero (X'00'), if *string-expression* is a binary string.

The result of the function is a varying length string that has the same CCSID of *string-expression*. *string-expression* and *pad* must have compatible data types. If the string expressions have different CCSID sets, then *pad* is converted to the CCSID set of *string-expression*. If either *string-expression* or *pad* is FOR BIT DATA, no character conversion occurs.

The length attribute of the result depends on *integer*. If *integer* is greater than 0, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the result is 1.

The actual length of the result is determined from *integer*. If *integer* is 0, the actual length is 0, and the result is the empty result string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

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

Example 1: Assume that NAME is a VARCHAR(15) column that contains the values 'Chris', 'Meg', and 'Jeff'. The following query will pad a value on the left with periods.

```
| SELECT LPAD(NAME,15,'.' ) AS NAME
| FROM T1;
```

The results are similar to the following:

```
| NAME
| -----
| .....Chris
| .....Meg
| .....Jeff
```

Example 2: Similar to Example 1, the following query will only pad each value to a length of 5:

```
| SELECT LPAD(NAME,5,'.' ) AS NAME
| FROM T1;
```

The results are similar to the following:

```
| NAME
| -----
| Chris
| ..Meg
| .Jeff
```

Example 3: Assume that NAME is a CHAR(15) column containing the values 'Chris', 'Meg', and 'Jeff'. Note that the LPAD function does not pad because NAME is a fixed length character field and is blank padded already. However, since the length of the result is 5, the columns are truncated:

```
| SELECT LPAD(NAME,5,'.' ) AS NAME
| FROM T1;
```

The results are similar to the following:

```
| NAME
| -----
| Chris
| Meg
| Jeff
```

Example 4: Assume that NAME is a VARCHAR(15) column containing the values 'Chris', 'Meg', and 'Jeff'. Note that in some cases, a partial instance of the pad specification is returned.

```
| SELECT LPAD(NAME,15,'123') AS NAME
| FROM T1
```

The results are similar to the following:

```
| NAME
| -----
| 1231231231Chris
| 123123123123Meg
| 12312312312Jeff
```


LTRIM

The LTRIM function removes blanks or hexadecimal zeros from the beginning of a string expression.

►►—LTRIM(*string-expression*)—◄◄

The schema is SYSIBM.

The LTRIM function returns the same results as the STRIP function with LEADING specified:

STRIP(*string-expression*, LEADING)

The argument must be an expression that returns a value that is a character string, graphic string, or binary string built-in data type. The argument must not be a LOB.

- If the argument is a binary string, the leading hexadecimal zeros (X'00') are removed.
- If the argument is a DBCS graphic string, the leading DBCS blanks are removed.
- If the argument is a Unicode graphic string, the leading UTF-16 or UCS-2 blanks are removed.
- If the argument is a UTF-8 character string, the leading UTF-8 blanks are removed.
- Otherwise, leading SBCS blanks are removed.

The result of the function depends on the data type of its argument:

- VARBINARY if the argument is a binary string
- VARCHAR if the argument is a character string
- VARGRAPHIC if the argument is a graphic string

The length attribute of the result is the same as the length attribute of *string-expression*. The actual length of the result is the length of the expression minus the number of characters removed. If all of the bytes are removed, the result is an empty string.

If the argument can be null, the result can be null; if the argument is null, the result is the null value. The CCSID of the result is the same as that of *string-expression*.

Example 1: Assume that host variable *HELLO* is defined as CHAR(9) and has a value of ' Hello'.

```
SELECT LTRIM(:HELLO)
FROM SYSIBM.SYSDUMMY1;
```

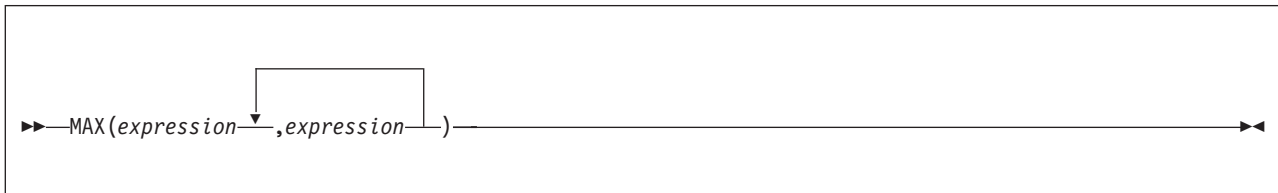
The result is 'Hello'.

Example 2: The following function returns a VARBINARY string with a length attribute 5, actual length 1 and a value BX'C1':

```
SELECT LTRIM(BINARY(X'00000000C1'))
FROM SYSIBM.SYSDUMMY1;
```

MAX

The MAX scalar function returns the maximum value in a set of values.



The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 20 on page 102. All but the first argument can be parameter markers. There must be two or more arguments.

Each argument must be an expression that returns a value of any built-in data type other than a CLOB, DBCLOB, BLOB, ROWID, or XML. Character and binary string arguments cannot have an actual length greater than 255, and graphic string arguments cannot have an actual length greater than 127.

The arguments are evaluated in the order in which they are specified. The result of the function is the maximum argument value. The result can be null if at least one argument is null; the result is the null value if one of the arguments is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined using the “Rules for result data types” on page 119. If the MAX function has more than two arguments, the rules are applied to the first two arguments to determine a candidate result type. The rules are then applied to that candidate result type and the third argument to determine another candidate result type. This process continues until all arguments are analyzed and the final result type and CCSID is determined.

GREATEST can be specified as a synonym for MAX.

Example 1: Assume the host variable *M1* is a DECIMAL(2,1) host variable with a value of 5.5, host variable *M2* is a DECIMAL(3,1) host variable with a value of 4.5, and host variable *M3* is a DECIMAL(3,2) host variable with a value of 6.25. The following function returns the value 6.25.

```
MAX(:M1, :M2, :M3)
```

Example 2: Assume the host variable *M1* is a CHAR(2) host variable with a value of 'AA', host variable *M2* is a CHAR(3) host variable with a value of 'AA ', and host variable *M3* is a CHAR(4) host variable with a value of 'AA A'. The following function returns the value 'AA A'.

```
MAX(:M1, :M2, :M3)
```

MICROSECOND

The MICROSECOND function returns the microsecond part of a value.

►►—MICROSECOND(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String representations of datetime values” on page 89.
- If *expression* is a number, it must be a timestamp duration. For the valid formats of timestamp durations, see “Datetime operands” on page 121.

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:

If the argument is a timestamp or string representation of a timestamp, the result is the microsecond part of the value, which is an integer between 0 and 999999.

If the argument is a duration, the result is the microsecond part of the value, which is an integer between -999999 and 999999. A nonzero result has the same sign as the argument.

Example 1: Assume that table TABLEX contains a TIMESTAMP column named TSTMPCOL and a SMALLINT column named INTCOL. Select the microseconds part of the TSTMPCOL column of the rows where the INTCOL value is 1234:

```
SELECT MICROSECOND(TSTMPCOL) FROM TABLEX
WHERE INTCOL = 1234;
```

MIDNIGHT_SECONDS

The MIDNIGHT_SECONDS function returns an integer, in the range of 0 to 86400, that represents the number of seconds between midnight and the time that is specified in the argument.

►►—MIDNIGHT_SECONDS(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String representations of datetime values” on page 89.

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 1: Find the number of seconds between midnight and 00:01:00, and midnight and 13:10:10. Assume that host variable *XTIME1* has a value of '00:01:00', and that *XTIME2* has a value of '13:10:10'.

```
SELECT MIDNIGHT_SECONDS(:XTIME1), MIDNIGHT_SECONDS(:XTIME2)
FROM SYSIBM.SYSDUMMY1;
```

This example returns 60 and 47410. Because there are 60 seconds in a minute and 3600 seconds in an hour, 00:01:00 is 60 seconds after midnight $((60 * 1) + 0)$, and 13:10:10 is 47410 seconds $((3600 * 13) + (60 * 10) + 10)$.

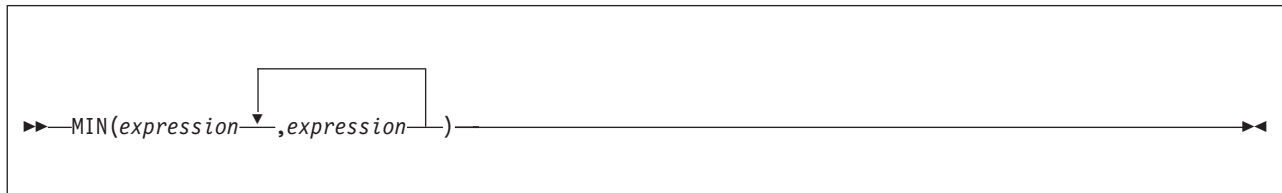
Example 2: Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
SELECT MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00')
FROM SYSIBM.SYSDUMMY1;
```

This example returns 86400 and 0. Although these two values represent the same point in time, different values are returned.

MIN

The MIN scalar function returns the minimum value in a set of values.



The schema is SYSIBM.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 20 on page 102. All but the first argument can be parameter markers. There must be two or more arguments.

Each argument must be an expression that returns a value of any built-in data type other than a CLOB, DBCLOB, BLOB, ROWID, or XML. Character string and binary string arguments cannot have an actual length greater than 255, and graphic string arguments cannot have an actual length greater than 127.

The arguments are evaluated in the order in which they are specified. The result of the function is the minimum argument value. The result can be null if at least one argument is null; the result is the null value if one of the arguments is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined using the “Rules for result data types” on page 119. If the MIN function has more than two arguments, the rules are applied to the first two arguments to determine a candidate result type. The rules are then applied to that candidate result type and the third argument to determine another candidate result type. This process continues until all arguments are analyzed and the final result type and CCSID is determined.

LEAST can be specified as a synonym for MIN.

Example 1: Assume the host variable *M1* is a DECIMAL(2,1) host variable with a value of 5.5, host variable *M2* is a DECIMAL(3,1) host variable with a value of 4.5, and host variable *M3* is a DECIMAL(3,2) host variable with a value of 6.25. The following function returns the value 4.5.

```
MIN(:M1, :M2, :M3)
```

Example 2: Assume the host variable *M1* is a CHAR(2) host variable with a value of 'AA', host variable *M2* is a CHAR(3) host variable with a value of 'AAA', and host variable *M3* is a CHAR(4) host variable with a value of 'AAAA'. The following function returns the value 'AA'.

```
MIN(:M1, :M2, :M3)
```

MINUTE

The MINUTE function returns the minute part of a value.

►►—MINUTE(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String representations of datetime values” on page 89.
- If *expression* is a number, it must be a time or timestamp duration. For the valid formats of time and timestamp durations, see “Datetime operands” on page 121.

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:

If the argument is a time, timestamp, or string representation of either, the result is the minute part of the value, which is an integer between 0 and 59.

If the argument is a time duration or timestamp duration, the result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example 1: Assume that a table named CLASSES contains one row for each scheduled class. Assume also that the class starting times are in the TIME column named STARTTM. Using these assumptions, select those rows in CLASSES that represent classes that start on the hour.

```
SELECT * FROM CLASSES
WHERE MINUTE(STARTTM) = 0;
```

MOD

The MOD function divides the first argument by the second argument and returns the remainder.

►►—MOD(*numeric-expression-1*,*numeric-expression-2*)——►►

The schema is SYSIBM.

The formula used to calculate the remainder is:

$$\text{MOD}(x,y) = x - (x/y) * y$$

Where x/y is the truncated integer result of the division. The result is negative only if the first argument is negative.

Each argument must be an expression that returns a value of any built-in numeric data type. The second argument cannot be zero.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The attributes of the result are based on the arguments as follows:

- If both arguments are large or small integers, the data type of the result is large integer.
- If both arguments are integers and at least one argument is a big integer, the data type of the result is big integer.
- If one argument is an integer and the other is a decimal, the data type of the result is decimal with the same precision and scale as the decimal argument.
- If both arguments are decimal, the data type of the result is decimal. The precision of the result is $\min(p-s, p'-s') + \max(s, s')$, and the scale of the result is $\max(s, s')$, where the symbols p and s denote the precision and scale of the first argument, and the symbols p' and s' denote the precision and scale of the second argument.

- If one argument is a floating-point number, and the other is not a DECFLOAT, or both argument is a floating-point number, the data type of the result is double precision floating-point.

The operation is performed in floating-point. If necessary, the operands are first converted to double precision floating-point numbers. For example, an operation that involves a floating-point number and either an integer or a decimal number is performed with a temporary copy of the integer or decimal number that has been converted to double precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

- If either argument is a DECFLOAT, the data type of the result is DECFLOAT(34). If either argument is a special decimal floating point value, the general rules for arithmetic operations apply. See “General Arithmetic Operation Rules for DECFLOAT” on page 186 for more information.

If one argument is a DECFLOAT and the second argument is zero, the result is NaN and an invalid operation condition is returned.

Example: Assume that *M1* and *M2* are two host variables. Find the remainder of dividing *M1* by *M2*.

```
SELECT MOD(:M1,:M2)
FROM SYSIBM.SYSDUMMY1;
```

The following table shows the result for this function for various values of *M1* and *M2*.

<i>M1</i> data type	<i>M1</i> value	<i>M2</i> data type	<i>M2</i> value	Result of MOD(:M1,:M2)
INTEGER	5	INTEGER	2	1
INTEGER	5	DECIMAL(3,1)	2.2	0.6
INTEGER	5	DECIMAL(3,2)	2.20	0.60
DECIMAL(4,2)	5.50	DECIMAL(4,1)	2.0	1.50
DECFLOAT	1	DECFLOAT	-INFINITY	1
DECFLOAT	-0	DECFLOAT	INFINITY	-0
DECFLOAT	-0	DECFLOAT	-INFINITY	-0

MONTH

The MONTH function returns the month part of a value.

►►—MONTH(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.
- If *expression* is a number, it must be a date or timestamp duration. For the valid formats of date and timestamp durations, see “Datetime operands” on page 121.

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:

If the argument is a date, timestamp, or string representation of either, the result is the month part of the value, which is an integer between 1 and 12.

If the argument is a date duration or timestamp duration, the result is the month part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example 1: Select all rows in the sample table DSN8910.EMP for employees who were born in May:

```
SELECT * FROM DSN8910.EMP
WHERE MONTH(BIRTHDATE) = 5;
```

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns an estimate of the number of months between two arguments.

►►—MONTHS_BETWEEN(*expression1*,*expression2*)—◄◄

The schema is SYSIBM.

expression1 **or** *expression2*

Expressions that return a value of any of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If either expression is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

If *expression1* represents a date that is later than *expression2*, the result is positive. If *expression1* represents a date that is earlier than *expression2*, the result is negative.

- If *expression1* and *expression2* represent dates or timestamps with the same day of the month, or both arguments represent the last day of their respective months, the result is a the whole number difference based on the year and month values, ignoring any time portions of timestamp arguments.
- Otherwise, the whole number part of the result is the difference based on the year and month values. The fractional part of the result is calculated from the remainder based on an assumption that every month has 31 days. If either argument represents a timestamp, the arguments are effectively processed as timestamps with maximum precision, and the time portions of these values are also considered when determining the result.

The result of the function is a DECIMAL(31,15). If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples 1: The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN ('2008-01-17','2008-02-17')
       AS MONTHS_BETWEEN
FROM SYSIBM.SYSDUMMY1;
```

The results of this statement are similar to the following results:

```
MONTHS_BETWEEN
-----
-1.000000000000000
```

Examples 2: The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN ('2008-02-20','2008-01-17')
       AS MONTHS_BETWEEN
FROM SYSIBM.SYSDUMMY1;
```

The results of this statement are similar to the following results:

MONTHS_BETWEEN

1.096774193548387

| *Example 3:* Calculate the number of months that project AD3100 will take. Assume
| that the start date is 1982-01-01 and the end date is 1983-02-01:

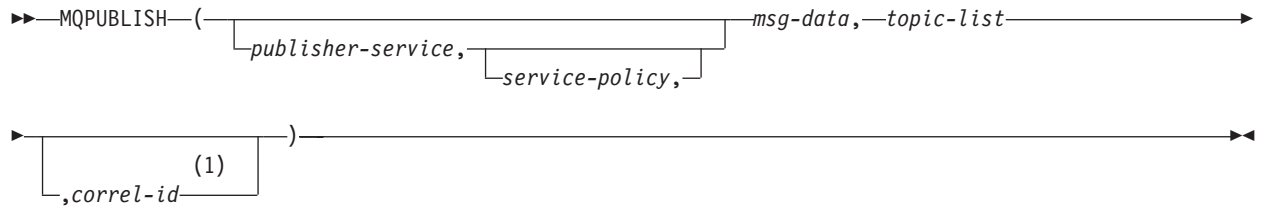
| SELECT MONTHS_BETWEEN (PRENDATE, PRSDATE)
| FROM PROJECT
| WHERE PROJNO='AD3100';

| The result is 13.000000000000000.

| *Example 4:* The following table illustrates the use of the MONTHS_BETWEEN
| function in certain situations:

MQPUBLISH

The MQPUBLISH function publishes a message to the specified MQSeries publisher, and returns a varying-length character string that indicates whether the function was successful or unsuccessful.



Notes:

- 1 *correl-id* can be specified only if a publisher service and a service policy have been defined.

The schema is DB2MQ1N or DB2MQ2N.

The MQPUBLISH function publishes the data that is contained in *msg-data* to the MQSeries publisher that is specified in *publisher-service*, using the quality-of-service policy that is defined by *service-policy*. A list of topics for the message must be specified, and an optional user-defined message correlation identifier can also be specified. The MQPUBLISH function requires the installation and configuration of an MQSeries-based publish and subscribe system, such as Websphere MQSeries Integrator. See www.ibm.com/software/MQSeries for more details. The function returns a value of '1' if successful or '0' if unsuccessful.

publisher-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination to which the message is sent. A service point is defined in the DSNAMT repository file, and it specifies a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *WebSphere MQ application messaging interface* for more details.

If *publisher-service* is not specified, DB2.DEFAULT.PUBLISHER is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to the MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *WebSphere MQ application messaging interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

msg-data

An expression that returns a value that is a built-in character string or graphic string data type. If the expression is a CLOB, the value must not be longer than 1 MB. Otherwise, the value must not be longer than 4000 bytes. The value of the expression specifies the data to be sent via MQSeries. A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values.

topic-list

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 40 bytes. The value of the expression specifies the topics for the message publication. One or more topics can be specified, where the topics are separated with a colon. For example, 't1:t2:the third topic' indicates that the message is associated with all three topics: t1, t2, and 'the third topic'.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression specifies the correlation identifier to be associated with this message. The *correl-id* is often specified in request-and-reply scenarios to associate requests with replies.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQPUBLISH, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVE will not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQPUBLISH request.

If *correl-id* is not specified, a correlation identifier is not used, and a correlation identifier is not added to the message.

The result of the function is a varying-length string with a length attribute of 1. The result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Publish the string 'Testing 123' to the default publisher service (DB2.DEFAULT.PUBLISHER), using the default service policy (DB2.DEFAULT.POLICY). Do not specify a topic or correlation identifier for the message.

```
SELECT DB2MQ2C.MQPUBLISH('Testing 123')
FROM SYSIBM.SYSDUMMY1;
```

Example 2: Publish the string 'Testing 345' to the publisher service 'MYPUBLISHER' under the topic 'TESTS', using the default service policy (DB2.DEFAULT.POLICY). Do not specify a correlation identifier for the message.

```
SELECT DB2MQ2C.MQPUBLISH('MYPUBLISHER','Testing 345', 'TESTS')
FROM SYSIBM.SYSDUMMY1;
```

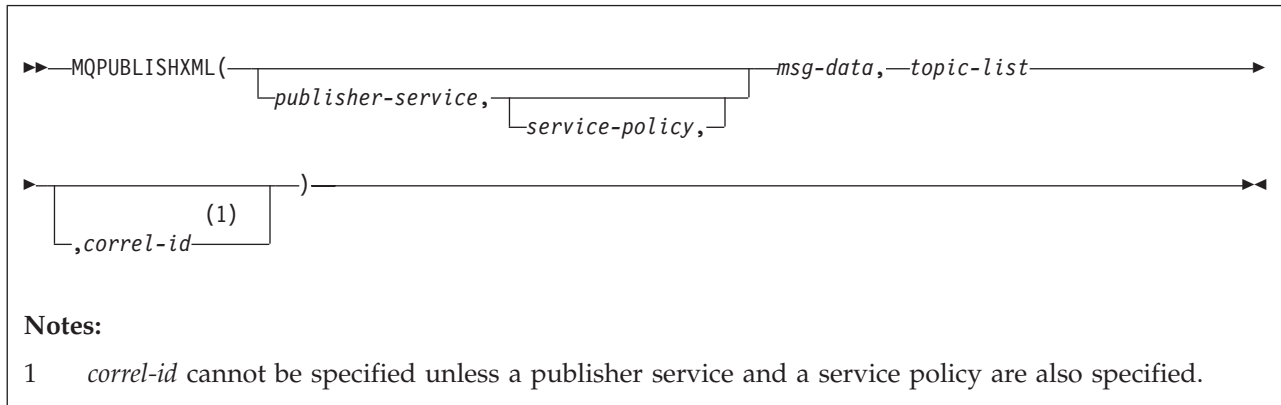
Example 3: Publishes the string 'Testing 678' to the publisher service 'MYPUBLISHER' under the topic 'TESTS', using the service policy 'MYPOLICY'. Specify a correlation identifier of 'TEST1' for the message.

```
SELECT DB2MQ2C.MQPUBLISH('MYPUBLISHER','MYPOLICY','Testing 678', 'TESTS', 'TEST1')
FROM SYSIBM.SYSDUMMY1;
```

All of the examples return a value of '1' if they are successful.

MQPUBLISHXML

The MQPUBLISHXML function publishes the XML data in a message to the specified MQSeries publisher.



The schema is DMQXML1C or DMQXML2C.

The MQPUBLISHXML function publishes the XML data that is contained in *msg-data* to the MQSeries publisher that is specified in *publisher-service*, using the quality-of-service policy that is defined by *service-policy*. A list of topics for the message must be specified, and an optional user-defined message correlation identifier can also be specified. The MQPUBLISHXML function requires the installation and configuration of an MQSeries-based publish and subscribe system, such as Websphere MQSeries Integrator. See www.ibm.com/software/MQSeries for more details. The function returns a value of '1' if it is successful or '0' if it is unsuccessful.

publisher-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to service point that is the logical MQSeries destination to which the message is sent. A service point is defined in the DSNAMT repository file, and it specifies a logical end point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See for more details.

If *publisher-service* is not specified, DB2.DEFAULT.PUBLISHER is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to the MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

msg-data

An expression that returns a value with a user-defined data type of DB2XML.XMLVARCHAR (for messages up to 3K in size) or DB2XML.XMLCLOB (for messages up to 1M in size). The value of the expression specifies the message data to be sent via MQSeries.

topic-list

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 40 bytes. The value of the expression specifies the topics for the message publication. One or more topics can be specified, where the topics are separated with a colon. For example, 't1:t2:the third topic' indicates that the message is associated with all three topics: t1, t2, and 'the third topic'.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier to be associated with this message. The *correl-id* is often specified in request-and-reply scenarios to associate requests with replies.

A null value, an empty string, and a fixed-length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request, such as MQPUBLISHXML, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEXML will not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQPUBLISHXML request.

If *correl-id* is not specified, a correlation identifier is not used, and a correlation identifier is not added to the message.

The result of the function is a varying-length string with a length attribute of 1. The result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed.

Example 1: The ORDER column in table ORDER_TABLE contains XML documents. Publish these XML documents to the default publisher service (DB2.DEFAULT.PUBLISHER), using the default service policy (DB2.DEFAULT.POLICY). Do not specify a topic or correlation identifier for the message.

```
SELECT MQPUBLISHXML(ORDER, 't1')
FROM ORDER_TABLE;
```

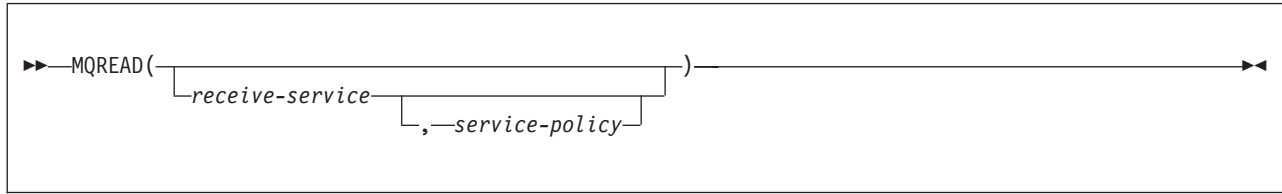
Example 2: The CUSTOMER column in table CUSTOMER_TABLE contains XML documents. Publish the XML documents for the Midwestern customers to the publisher service 'MYPUBLISHER' under the topic '/MIDWEST/CUSTOMERS', using the default service policy (DB2.DEFAULT.POLICY). Do not specify a correlation identifier for the message.

```
SELECT MQPUBLISHXML('MYPUBLISHER', CUSTOMER, '/MIDWEST/CUSTOMERS')
FROM CUSTOMER_TABLE
WHERE TERRITORY = 'MIDWEST';
```

All of the examples return a value of '1' if they are successful.

MQREAD

The MQREAD function returns a message from a specified MQSeries location without removing the message from the queue.



The schema is DB2MQ1N or DB2MQ2N.

The MQREAD function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the message from the queue that is associated with *receive-service*, but instead returns the message at the beginning of the queue.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is received. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

The result of the function is a varying-length string with a length attribute of 4000. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve the message at the beginning of the queue that is specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREAD()  
FROM TABLE;
```

The message at the beginning of the queue specified by the default server and using the default policy is returned as VARCHAR(4000).

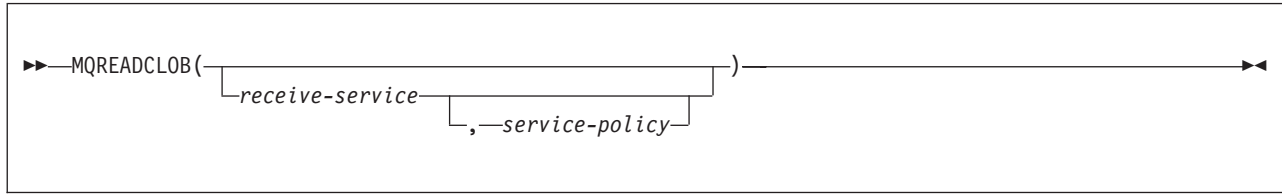
Example 2: Read the message from the beginning of the queue specified by the service MYSERVICE, using the default policy MYPOLICY.

```
SELECT MQREAD('MYSERVICE','MYPOLICY')  
FROM TABLE;
```

The message at the beginning of the queue specified by MYSERVICE and using MYPOLICY is returned as VARCHAR(4000).

MQREADCLOB

The MQREADCLOB function returns a message from a specified MQSeries location without removing the message from the queue.



The schema is DB2MQ1N or DB2MQ2N.

The MQREADCLOB function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the message from the queue that is associated with *receive-service*, but instead returns the message at the beginning of the queue.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is received. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

The result of the function is a CLOB with a length attribute of 1 MB. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Read the message from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREADCLOB(0)
FROM TABLE;
```

The message at the beginning of the queue specified by the default service and using the default policy is returned as a CLOB.

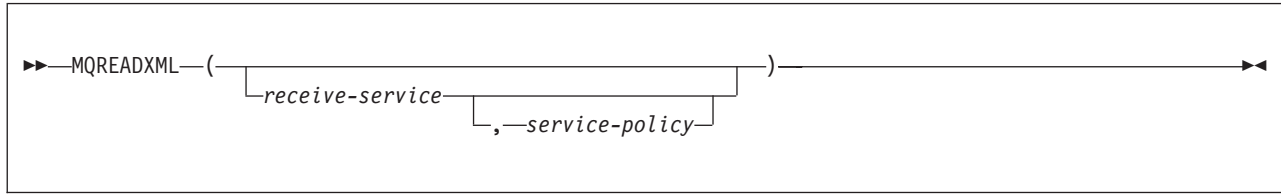
Example 2: Read the message from the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREADCLOB('MYSERVICE')  
FROM TABLE;
```

The message at the beginning of the queue specified by MYSERVICE and using the default policy is returned as a CLOB.

MQREADXML

The MQREADXML function returns an XML message from a specified MQSeries location without removing the message from the queue.



The schema is DMQXML1C or DMQXML2C

The MQREADXML function returns an XML message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the message from the queue that is associated with *receive-service*, but instead returns the message at the beginning of the queue.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is read. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

The result of the function is a value with the user-defined data type DB2XML.XMLVARCHAR that contains the XML messages. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

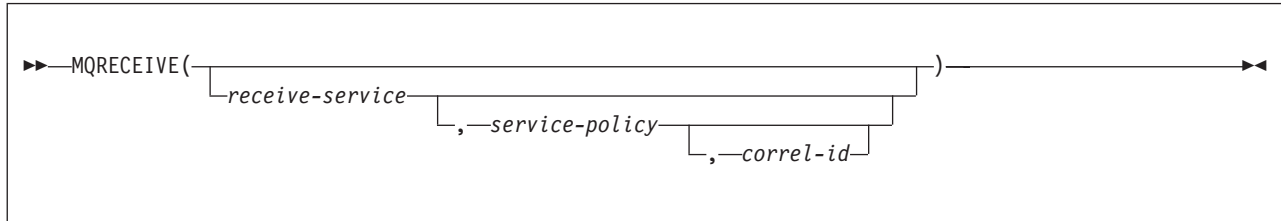
Example: Read the message from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQREADXML()  
FROM SYSIBM.SYSDUMMY;
```

The message at the beginning of the queue specified by the default service and using the default policy is returned as a XMLVARCHAR.

MQRECEIVE

The MQRECEIVE function returns a message from a specified MQSeries location and removes the message from the queue.



The schema is DB2MQ1N or DB2MQ2N.

The MQRECEIVE function returns a message from the MQSeries location specified by *receive-service*, using the quality-of-service policy defined in *service-policy*. Performing this operation removes the message from the queue that is associated with *receive-service*.

receive-service

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to a service point that is the logical MQSeries destination
| from which the message is received. A service point is defined in the DSNAMT
| repository file, and it represents a logical end-point from which a message is
| sent or received. A service point definition includes the name of the MQSeries
| queue manager and the name of the queue. See *MQSeries Application Messaging
| Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to an MQSeries AMI service policy that is used in handling
| this message. A service policy is defined in the DSNAMT repository file, and it
| specifies a set of quality-of-service options that are to be applied to this
| messaging operation. These options include message priority and message
| persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

correl-id

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The expression must have an actual length
| that is no greater than 24 bytes. The value of the expression specifies the
| correlation identifier that is associated with this message. A correlation
| identifier is often specified in request-and-reply scenarios to associate requests
| with replies. The first message with a matching correlation identifier is
| returned.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another

request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVE does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a varying-length string of length attribute of 4000. The result can be null. The result is null if no messages are available to return.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve the message from beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVE()  
FROM TABLE;
```

The message at the beginning of the queue specified by the default service and using the default policy is returned as VARCHAR(4000) and is deleted from the queue.

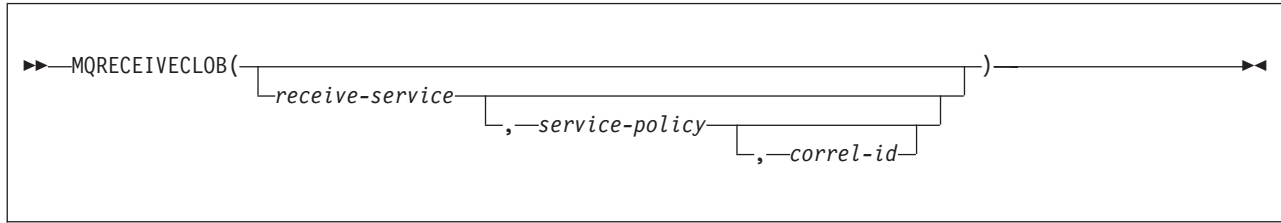
Example 2: Retrieve the first message with a correlation identifier that matches '1234' from the beginning of the queue specified by the service MYSERVICE, using the policy MYPOLICY.

```
SELECT MQRECEIVE('MYSERVICE', 'MYPOLICY', '1234')  
FROM TABLE;
```

The message with CORRELID of '1234' from the beginning of the queue specified by MYSERVICE and using MYPOLICY is returned as VARCHAR(4000) and is deleted from the queue.

MQRECEIVECLOB

The MQRECEIVECLOB function returns a message from a specified MQSeries location and removes the message from the queue.



The schema is DB2MQ1N or DB2MQ2N.

The MQRECEIVECLOB function returns a message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the message from the queue that is associated with *receive-service*.

receive-service

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to a service point that is the logical MQSeries destination
| from which the message is received. A service point is defined in the DSNAMT
| repository file, and it represents a logical end-point from which a message is
| sent or received. A service point definition includes the name of the MQSeries
| queue manager and the name of the queue. See *MQSeries Application Messaging
| Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression specifies an MQSeries AMI service policy that is used in handling
| this message. A service policy is defined in the DSNAMT repository file, and it
| specifies a set of quality-of-service options that are to be applied to this
| messaging operation. These options include message priority and message
| persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

correl-id

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The expression must have an actual length
| that is no greater than 24 bytes. The value of the expression specifies the
| correlation identifier that is associated with this message. A correlation
| identifier is often specified in request-and-reply scenarios to associate requests
| with replies. The first message with a matching correlation identifier is
| returned.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another

request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVECLOB does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a CLOB with a length attribute of 1 MB. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve the message from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MORERECEIVECLOB()  
FROM TABLE;
```

The message at the beginning of the queue specified by the default service and using the default policy is returned as a CLOB and is deleted from the queue.

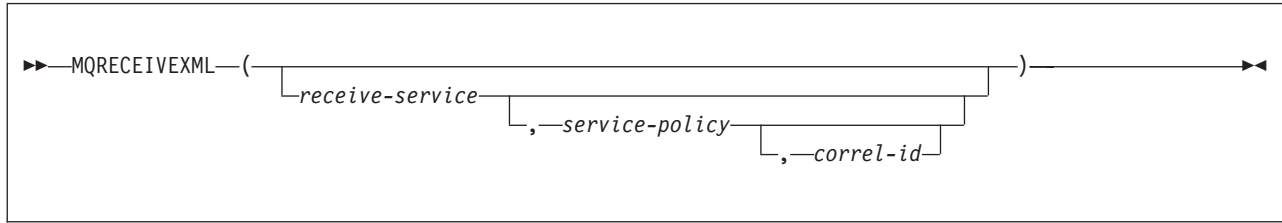
Example 2: Retrieve the message from the beginning of the queue specified by the service MYSERVICE, using the policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVECLOB('MYSERVICE')  
FROM TABLE;
```

The message at the beginning of the queue specified by MYSERVICE and using the default policy is returned as a CLOB and is deleted from the queue.

MQRECEIVEXML

The MQRECEIVEXML function returns a message from a specified MQSeries location and removes the message from the queue.



The schema is DMQXML1C or DMQXML2C.

The MQRECEIVEXML function returns an XML message from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the message from the queue that is associated with *receive-service*.

receive-service

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to a service point that is the logical MQSeries destination
| from which the message is read. A service point is defined in the DSNAMT
| repository file, and it represents a logical end-point from which a message is
| sent or received. A service point definition includes the name of the MQSeries
| queue manager and the name of the queue. See *MQSeries Application Messaging
| Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to an MQSeries AMI service policy that is used in handling
| this message. A service policy is defined in the DSNAMT repository file, and it
| specifies a set of quality-of-service options that are to be applied to this
| messaging operation. These options include message priority and message
| persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

correl-id

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The expression must have an actual length
| that is no greater than 24 bytes. The value of the expression specifies the
| correlation identifier to be associated with this message. The *correl-id* is often
| specified in request-and-reply scenarios to associate requests with replies. The
| first message with a matching correlation identifier is returned.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQSENDXML, the *correl-id* must be specified the same to be

recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEXML does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSENDXML request.

If *correl-id* is not specified, a correlation identifier is not used, and the message at the beginning of the queue is returned.

The result of the function is a value with user-defined data type DB2XML.XMLVARCHAR that contains the XML messages. The result can be null. If no messages are available to be returned, the result is the null value.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

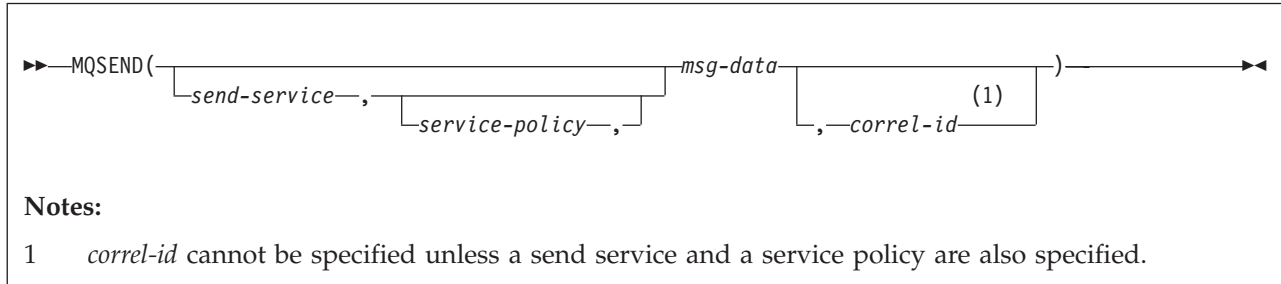
Example: Retrieve the message from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT MQRECEIVEXML()  
FROM SYSIBM.SYSDUMMY;
```

The message at the beginning of the queue specified by the default service and using the default policy is returned. The message is deleted from the queue.

MQSEND

The MQSEND function sends data to a specified MQSeries location, and returns a varying-length character string that indicates whether the function was successful or unsuccessful.



Notes:

- 1 *correl-id* cannot be specified unless a send service and a service policy are also specified.

The schema is DB2MQ1N or DB2MQ2N.

The MQSEND function sends the data that is contained in *msg-data* to the MQSeries location that is specified by *send-service*, using the quality-of-service policy that is defined in *service-policy*. The returned value is '1' if the function was successful or '0' if unsuccessful.

send-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination to which the message is sent. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *send-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression specifies an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

msg-data

An expression that returns a value that is a built-in character string data type. If the expression is a CLOB, the value must not be longer than 1 MB. Otherwise, the value must not be longer than 4000 bytes. The value of the expression is the message data that is to be sent via MQSeries. A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. *correl-id* must not be specified unless *send-service* and *service-policy* are also specified.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQRECEIVE, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQSEND does not match a *correl-id* value of 'test ' (with trailing blanks) specified subsequently on an MQRECEIVE request.

If *correl-id* is not specified, a correlation identifier is not sent.

The result of the function is a varying-length string with a length attribute of 1. The result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Send the string "Testing msg" to the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY) and no correlation identifier.

```
SELECT MQSEND('Testing msg')
FROM TABLE;
```

The message is sent to the default service, using the default policy.

Example 2: Send the string "Testing 123" to the service MYSERVICE, using the policy MYPOLICY and the correlation identifier "TEST3".

```
SELECT MQSEND('MYSERVICE','MYPOLICY','Testing 123','TEST3')
FROM TABLE;
```

The string "TESTING 123" is sent to MYSERVICE, using MYPOLICY and the correlation identifier "TEST3".

MQSENDXML

The MQSENDXML function sends XML data to a specified MQSeries location.

```
MQSENDXML ( ( send-service , service-policy , msg-data , correl-id (1) ) )
```

Notes:

- 1 *correl-id* cannot be specified unless a send service and a service policy are also specified.

The schema is DMQXML1C or DMQXML2C.

The MQSENDXML function sends the XML data that is contained in *msg-data* to the MQSeries location that is specified by *send-service*, using the quality-of-service policy that is defined in *service-policy*. The function returns a value of '1' if it is successful or a '0' if it is unsuccessful.

send-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination to which the message is sent. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *send-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression specifies an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

msg-data

An expression that returns a value with a user-defined data type of DB2XML.XMLVARCHAR (for data up to 3 KB in size) or DB2XML.XMLCLOB (for data up to 1 MB in size). The value of the expression specifies the message data to be sent via MQSeries.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the

correlation identifier to be associated with this message. The *correl-id* is often specified in request-and-reply scenarios to associate requests with replies.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQSENDXML, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEXML does not match a *correl-id* value of 'test ' (with trailing blanks) specified subsequently on an MQSENDXML request.

If *correl-id* is not specified, a correlation identifier is not sent.

The result of the function is a varying-length string with a length attribute of 1. The data type of the result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: The ORDER column in table ORDER_TABLE contains XML documents. Send the XML documents in this column to the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). Do not specify a correlation identifier.

```
SELECT MQSENDXML(ORDER)
FROM ORDER_TABLE;
```

The message is sent to the default service, using the default policy.

Example 2: The CUSTOMER column in table CUSTOMER_TABLE contains XML documents. Send the XML documents for the Midwestern customers to the service 'MYSERVICE', using the default service policy (DB2.DEFAULT.POLICY) and a correlation identifier of 'MIDWESTERN'.

```
SELECT MQSENDXML('MYSERVICE', 'MYPOLICY', CUSTOMER, 'MIDWESTERN')
FROM CUSTOMER_TABLE
WHERE TERRITORY = 'MIDWESTERN';
```

All of the examples return a value of '1' if they are successful.

MQSENDXMLFILE

The MQSENDXMLFILE function sends up to 3 KB of data that is contained in the specified XML file to a specified MQSeries location.

►► MQSENDXMLFILE (*send-service* , *service-policy* , *xml-file* , *correl-id* (1)) ►►

Notes:

- 1 *correl-id* cannot be specified unless a send service and a service policy are also specified.

The schema is DMQXML1C or DMQXML2C.

The MQSENDXMLFILE function sends the XML data that is contained in *xml-file* to the MQSeries location that is specified by *send-service*, using the quality-of-service policy that is defined in *service-policy*. The function returns a value of '1' if it is successful or a '0' if it is unsuccessful.

send-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination to which the message is sent. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *send-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression specifies an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

xml-file

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 48 bytes. The value of the expression specifies the name of the file that contains the XML data that is to be sent via MQSeries.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length

that is no greater than 24 bytes. The value of the expression specifies the correlation identifier to be associated with this message. The *correl-id* is often specified in request-and-reply scenarios to associate requests with replies.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQSENDXML, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVXML does not match a *correl-id* value of 'test ' (with trailing blanks) specified subsequently on an MQSENDXML request.

If *correl-id* is not specified, a correlation identifier is not sent.

The result of the function is a varying-length string with a length attribute of 1. The data type of the result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Send the XML documents that are in file '/tmp/test1.xml' to the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). Do not specify a correlation identifier.

```
SELECT MQSENDXMLFILE('/tmp/test1.xml')
FROM   SYSIBM.SYSDUMMY1;
```

Example 2: Send the XML documents that are in file '/tmp/test2.xml' to the service 'MYSERVICE', using the service policy 'MYPOLICY' and the correlation identifier 'Test 2').

```
SELECT MQSENDXMLFILE('MYSERVICE', 'MYPOLICY', CUSTOMER, '/tmp/test2.xml', 'Test2')
FROM   SYSIBM.SYSDUMMY1;
```

All of the examples return a value of '1' if they are successful.

MQSENDXMLFILECLOB

The MQSENDXMLFILECLOB function sends up to 1 MB of data that is contained in an XML file to a specified MQSeries location.

►►—MQSENDXMLFILECLOB—(—
 —*send-service*—,—
 —*service-policy*—,—
 xml-file—
 —*correl-id*—⁽¹⁾—
—)

Notes:

- 1 *correl-id* cannot be specified unless a send service and a service policy are also specified.

The schema is DMQXML1C or DMQXML2C.

The MQSENDXMLFILECLOB function sends the XML data that is contained in *xml-file* to the MQSeries location that is specified by *send-service*, using the quality-of-service policy that is defined in *service-policy*. The function returns a value of '1' if it is successful or a '0' if it is unsuccessful.

send-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination to which the message is sent. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *send-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression specifies an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

xml-file

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 48 bytes. The value of the expression specifies the name of the file that contains the XML data that is to be sent via MQSeries.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length

that is no greater than 24 bytes. The value of the expression specifies the correlation identifier to be associated with this message. The *correl-id* is often specified in request-and-reply scenarios to associate requests with replies.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQSENDXML, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEXML does not match a *correl-id* value of 'test ' (with trailing blanks) specified subsequently on an MQSENDXML request.

If *correl-id* is not specified, a correlation identifier is not sent.

The result of the function is a varying-length string with a length attribute of 1. The data type of the result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

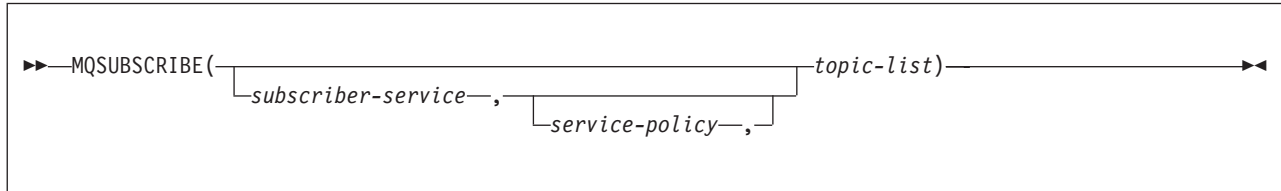
Example: Send the XML documents that are in file '/tmp/test1.xml' to the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY). Do not specify a correlation identifier.

```
SELECT MQSENDXMLFILECLOB('/tmp/test1.xml')
FROM   SYSIBM.SYSDUMMY1;
```

The example returns a value of '1' if it is successful.

MQSUBSCRIBE

The MQSUBSCRIBE function registers a subscription to MQSeries messages that are published on a specified topic, and returns a varying-length character string that indicates if the function was successful.



The schema is DB2MQ1N or DB2MQ2N.

The MQSUBSCRIBE function is used to register interest in MQSeries messages published on specified topics. The *subscriber-service* specifies a logical destination for messages that match the specified topics. Messages that match the specified topics are placed on the queue defined by *subscriber-service*. The messages can then be read or received through a subsequent invocation of the MQREAD, MQRECEIVE, MQREADALL, or MQRECEIVEALL functions. The MQSUBSCRIBE function requires the installation and configuration of an MQSeries-based publish and subscribe system, such as Websphere MQSeries Integrator. See www.ibm.com/software/MQSeries for more details.

The function returns a value of '1' if successful or '0' if unsuccessful. A successful execution of this function causes the publish and subscribe server to forward messages that match the specified topics to the service point defined by *subscriber-service*.

subscriber-service

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
null value, an empty string, or a string with trailing blanks. The expression
must have an actual length that is no greater than 48 bytes. The value of the
expression refers to a service point that is the logical MQSeries subscription
point to which messages that match the specified topics are sent. A subscriber's
service point is defined in the DSNAMT repository file, and it specifies a
logical subscription point to which a message is sent. A service point definition
includes the name of the MQSeries queue manager and the name of the queue.
See *MQSeries Application Messaging Interface* for more details.

If *subscriber-service* is not specified, DB2.DEFAULT.SUBSCRIBER is used.

service-policy

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
null value, an empty string, or a string with trailing blanks. The expression
must have an actual length that is no greater than 48 bytes. The value of the
expression refers to the MQSeries AMI service policy that is used in handling
this message. A service policy is defined in the DSNAMT repository file, and it
specifies a set of quality-of-service options that are to be applied to this
messaging operation. These options include message priority and message
persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

topic-list

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 40 bytes. The value of the expression specifies the types of messages to receive. Only messages published with the specified topics are received by this subscription. Multiple subscriptions can coexist. One or more topics can be specified, where the topics are separated with a colon. For example, 't1:t2:the third topic' indicates that the message is associated with all three topics: t1, t2, and 'the third topic'.

The result of the function is a varying-length string with a length attribute of 1. The result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Register an interest in messages that contain the topic 'Weather'. Allow the message to be sent to the default subscriber service point (DB2.DEFAULT.SUBSCRIBER), using the default service policy.

```
SELECT DB2MQ2C.MQSUBSCRIBE('Weather')
FROM SYSIBM.SYSDUMMY1;
```

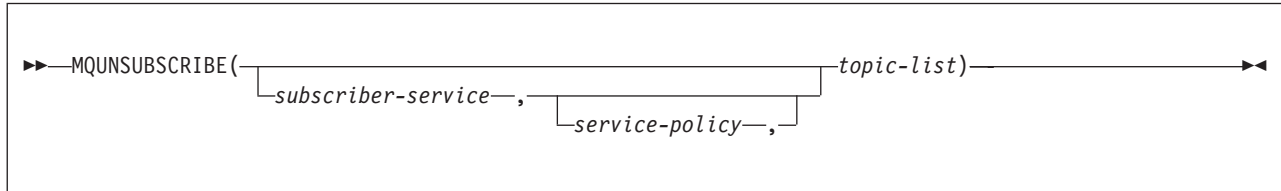
Example 2: Registering an interest in messages that contain the topic 'Stocks'. Specify 'PORTFOLIO-UPDATES' as the subscriber service point to which the message is to be sent, using service policy 'BASIC-POLICY'.

```
SELECT DB2MQ2C.MQSUBSCRIBE ('PORTFOLIO-UPDATES', 'BASIC-POLICY', 'Stocks')
FROM SYSIBM.SYSDUMMY1;
```

All of the examples return a value of '1' if they are successful.

MQUNSUBSCRIBE

The MQUNSUBSCRIBE function unsubscribes to MQSeries messages that are published on a specified topic, and returns a varying-length character string that indicates if the function was successful.



The schema is DB2MQ1N or DB2MQ2N.

The MQUNSUBSCRIBE function is used to unregister an existing message subscription. The *subscriber-service*, *service-policy*, and *topic-list* identify which subscription is canceled. The MQUNSUBSCRIBE function requires the installation and configuration of an MQSeries-based publish and subscribe system, such as Websphere MQSeries Integrator. See www.ibm.com/software/MQSeries for more details.

The function returns a value of '1' if successful or '0' if unsuccessful. A successful execution of this function causes the publish and subscribe server to remove the subscription that is identified by the given parameters. Messages that match the specified topics are no longer sent to the service point defined by *subscriber-service*.

subscriber-service

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to a service point that is the logical MQSeries subscription
| point to which messages that match the specified topics are sent. A subscriber's
| service point is defined in the DSNAMT repository file, and it specifies a
| logical subscription point to which a message is sent. A service point definition
| includes the name of the MQSeries queue manager and the name of the queue.
| See *MQSeries Application Messaging Interface* for more details.

If *subscriber-service* is not specified, DB2.DEFAULT.SUBSCRIBER is used.

service-policy

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to the MQSeries AMI service policy that is used in handling
| this message. A service policy is defined in the DSNAMT repository file, and it
| specifies a set of quality-of-service options that are to be applied to this
| messaging operation. These options include message priority and message
| persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

topic-list

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the

null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 40 bytes. The value of the expression specifies the types of messages to receive. Only messages published with the specified topics are received by this subscription. Multiple subscriptions can coexist. One or more topics can be specified, where the topics are separated with a colon. For example, 't1:t2:the third topic' indicates that the message is associated with all three topics: t1, t2, and 'the third topic'.

The result of the function is a varying-length string with a length attribute of 1. The result is nullable, even though a null value is never returned.

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Cancel the subscription for messages that contain the topic 'Weather'. The subscriber is registered with the default subscriber service point (DB2.DEFAULT.SERVICE) with the default service policy (DB2.DEFAULT.SERVICE).

```
SELECT DB2MQ2C.UNMQSUBSCRIBE('Weather')
FROM SYSIBM.SYSDUMMY1;
```

Example 2: Cancel the subscription for messages that contain the topic 'Stocks'. The subscriber is registered with the subscriber service point 'PORTFOLIO-UPDATES' with the service policy 'BASIC-POLICY'.

```
SELECT DB2MQ2C.MQUNSUBSCRIBE ('PORTFOLIO-UPDATES', 'BASIC-POLICY', 'Stocks')
FROM SYSIBM.SYSDUMMY1;
```

All of the examples return a value of '1' if they are successful.

MULTIPLY_ALT

The MULTIPLY_ALT scalar function returns the product of the two arguments. This function is an alternative to the multiplication operator and is especially useful when the sum of the precisions of the arguments exceeds 31.

►►MULTIPLY_ALT(*exact-numeric-expression-1*,*exact-numeric-expression-2*)◄◄

The schema is SYSIBM.

Each argument must be an expression that returns the value of one of the following built-in numeric data types: DECIMAL, BIGINT, INTEGER, or SMALLINT.

The result of the function is a DECIMAL. The precision and scale of the result are determined as follows, using the symbols p and s to denote the precision and scale of the first argument, and the symbols p' and s' to denote the precision and scale of the second argument.

- The precision is $\text{MIN}(31, p+p')$
- The scale is:
 - 0 if the scale of both arguments is 0
 - $\text{MIN}(31, s+s')$ if $p+p'$ is less than or equal to 31
 - $\text{MAX}(\text{MIN}(3, s+s'), 31-(p-s+p'-s'))$ if $p+p'$ is greater than 31.

The result can be null if at least one argument can be null; the result is the null value if one of the arguments is null.

The MULTIPLY_ALT function is a better choice than the multiplication operator when performing decimal arithmetic where a scale of at least 3 is desired and the sum of the precisions exceeds 31. In these cases, the internal computation is performed so that overflows are avoided and then assigned to the result type value using truncation for any loss of scale in the final result. Note that the possibility of overflow of the final result is still possible when the scale is 3.

The following table compares the result data types from the MULTIPLY_ALT function with the result data type of the multiplication operator when decimal data is used:

Type of Argument1	Type of Argument2	Result using MULTIPLY_ALT	Result using multiplication operator
DECIMAL(31,3)	DECIMAL(15,8)	DECIMAL(31,3)	DECIMAL(31,11)
DECIMAL(26,23)	DECIMAL(10,1)	DECIMAL(31,19)	DECIMAL(31,24)
DECIMAL(18,17)	DECIMAL(20,19)	DECIMAL(31,29)	DECIMAL(31,31)
DECIMAL(16,3)	DECIMAL(17,8)	DECIMAL(31,9)	DECIMAL(31,11)
DECIMAL(26,5)	DECIMAL(11,0)	DECIMAL(31,3)	DECIMAL(31,5)
DECIMAL(21,1)	DECIMAL(15,1)	DECIMAL(31,2)	DECIMAL(31,2)

NEXT_DAY

The NEXT_DAY function returns a timestamp that represents the first occurrence of the specified weekday that is after the date argument.

►►—NEXT_DAY(*expression*,*string-expression*)—►►

The schema is SYSIBM.

If *expression* is a timestamp or valid string representation of a timestamp, the timestamp value has the same hours, minutes, seconds, and partial seconds as *expression*. If *expression* is a date, or a valid string representation of a date, then the hours, minutes, seconds, and partial seconds value of the result is 0.

expression

An expression that returns one of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

string-expression

An expression that returns a built-in character or graphic string data type that is not a LOB. For portability across the platforms, the value should compare equal to the full name of a day of the week or should compare equal to the abbreviation of a day of the week. For example:

Day of week	Abbreviation
MONDAY	MON
TUESDAY	TUE
WEDNESDAY	WED
THURSDAY	THU
FRIDAY	FRI
SATURDAY	SAT
SUNDAY	SUN

The minimum length of the input value is the length of the abbreviation. Leading blanks must not be specified in *string-expression*. Trailing blanks are trimmed from *string-expression*. The resulting value is folded to uppercase. Any characters other than blank that immediately follow a valid abbreviation are ignored.

The result of the function is a timestamp. The result can be null; if any argument is null, the result is the null value.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example 1: Set the host variable NEXTDAY with a timestamp for the date of the Tuesday that follows April 24, 2007.

```
|          SET :NEXTDAY = NEXT_DAY(TIMESTAMP '2007-04-24-00.00.00.000000', 'TUESDAY');
```

| The host variable NEXTDAY is set with the value of '2007-05-01-00.00.00.000000',
| since April, 24, 200 is itself a Tuesday'.

| *Example 2:* Set the host variable vNEXTDAY with the date of the first Monday in
| May, 2007. Assume the host variable vDAYOFWEEK = 'MON':

```
|          SET :vNEXTDAY = NEXT_DAY(LAST_DAY(CURRENT_DATE), :vDAYOFWEEK);
```

| The host variable vNEXTDAY is set with the value of '2007-05-07', assuming that
| the value of the CURRENT_DATE special register is '2007-04-24'.

NORMALIZE_DECFLOAT

The NORMALIZE_DECFLOAT function returns a DECFLOAT value that is the result of the argument, set to its simplest form. That is, a non-zero number that has any trailing zeros in the coefficient has those zeros removed by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly. A zero has its exponent set to 0.

►►—NORMALIZE_DECFLOAT(*decfloat-expression*)—◄◄

The schema is SYSIBM.

decfloat-expression

The argument must be an expression that returns a DECFLOAT value.

If the argument is a special decimal floating point value then the general rules for arithmetic operations apply. See “General Arithmetic Operation Rules for DECFLOAT” on page 186 for more information.

The result of the function is a DECFLOAT(16) value if the data type of *decfloat-expression* is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples: The following examples show the result of using the NORMALIZE_DECFLOAT function on various DECFLOAT values:

NORMALIZE_DECFLOAT(DECFLOAT(2.1))	= 2.1
NORMALIZE_DECFLOAT(DECFLOAT(-2.0))	= -2
NORMALIZE_DECFLOAT(DECFLOAT(1.200))	= 1.2
NORMALIZE_DECFLOAT(DECFLOAT(-120))	= -1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(120.00))	= 1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(0.00))	= 0
NORMALIZE_DECFLOAT(-NAN)	= -NAN
NORMALIZE_DECFLOAT(-INFINITY)	= -INFINITY

NORMALIZE_STRING

The `NORMALIZE_STRING` function takes a Unicode string argument and returns a normalized string that can be used for comparison.

The `NORMALIZE_STRING` function can convert two strings that look the same (such as Å, which can be encoded in UTF-16 as X'00C5' and as X'00410307') but might not be encoded using the same Unicode code point, to a normalized form that can be compared.

►► `NORMALIZE_STRING(unicode-string,

NFC
NFD
NFKC
NFKD

, —integer)` ►►

The schema is SYSIBM.

unicode_string

An expression that returns a value of a built-in character string or graphic string data type that is either Unicode UTF-8 or Unicode UTF-16, and is not a LOB. The CAST specification can be used to convert ASCII or EBCDIC data to Unicode for use with this function.

NFC, NFD, NFKC, or NFKD

Specifies the normalized form:

NFC Canonical Decomposition followed by Canonical Composition

NFD Canonical Decomposition

NFKC Compatibility Decomposition followed by Canonical Composition

NFKD Compatibility Decomposition

integer

The length attribute, in bytes if the string is a character string, or in double byte code points if the string is a graphic string, for the resulting variable length string. The value must be an integer between 1 and 32704 if the source string is character, or 16352 if the source string is graphic.

The result of the function is a varying length string with a data type that depends on the data type of *unicode-string*:

- VARCHAR if *unicode-string* is CHAR or VARCHAR
- VARGRAPHIC if *unicode-string* is GRAPHIC or VARGRAPHIC

The CCSID of the result is the same as the CCSID of *unicode-string*.

The length attribute of the result depends on whether *integer* is specified. If *integer* is specified, the length attribute of the result is *integer* bytes or double byte code points. If *integer* is not specified, the length attribute of the result is MIN(3*n,32704) for character strings, or MIN(3*n,16352) for graphic strings, where *n* is the length attribute of the source.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Example 1: In the following example, "ábc" is normalized to normalization form NFC:

```
SET :hv1 = NORMALIZE_STRING('ábc',NFC) -- x'0061030100620063'
```

hv1 is set to 'ábc' -- X'00E100620063'. Using normalization form NFC, the two code-point sequence X'00610301', which represents the character 'á', is normalized to X'00E1' which is also the pre-composed equivalent of X'00610301'.

Example 2: In the following example, "ábc" is normalized to normalization form NFD.

```
SET :hv1 = NORMALIZE_STRING('ábc',NFD) -- x'00E100620063'
```

hv1 is set to 'ábc' -- X'0061030100620063'. Using normalization form NFD, the code point X'00E1' is decomposed into the two code-point sequence X'00610301', which consists of the Latin lower case letter A and the combining acute accent character.

NULLIF

The NULLIF function returns the null value if the two arguments are equal; otherwise, it returns the value of the first argument.

►►—NULLIF(*expression*,*expression*)—◄◄

The schema is SYSIBM.

The two arguments must be compatible and comparable. (See the compatibility matrix in Table 20 on page 102.) Neither argument can be a BLOB, CLOB, DBCLOB, XML, or distinct type. Character-string and graphic-string arguments are compatible and comparable with datetime values.

The attributes of the result are the attributes of the first argument.

The result of using NULLIF(*e1*,*e2*) is the same as using the CASE expression:

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

When *e1*=*e2* evaluates to unknown because one or both arguments is null, CASE expressions consider the evaluation not true. In this case, NULLIF returns the value of the first argument.

Example: Assume that host variables *PROFIT*, *CASH*, and *LOSSES* have decimal data types with the values of 4500.00, 500.00, and 5000.00 respectively. The following function returns a null value:

```
NULLIF (:PROFIT + :CASH , :LOSSES)
```

OVERLAY

The OVERLAY function returns a string that is composed of one argument that is inserted into another argument at the same position where some number of bytes have been deleted.

►► OVERLAY (—*source-string*—, —*insert-string*—, —*start*—, —*length*—, —*CODEUNITS16*—, —*CODEUNITS32*—, —*OCTETS*—) ►►

The schema is SYSIBM.

The OVERLAY function returns a string where a substring of *length*, beginning at *start* has been deleted from *source-string*, and where *insert-string* has been inserted into *source-string* beginning at *start*. If the value of *start* plus *length* is greater than the length of *source-string*, the substring that is deleted is from *start* to the end of *source-string*.

If the length of the result string exceeds the maximum for the return type, an error is returned.

The OVERLAY function is identical to the INSERT function, except that the length argument is optional.

source-string

An expression that specifies the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB. The actual length of the string must be greater than or equal to 1 byte and less than or equal to 32704 bytes.

insert-string

An expression that specifies the string that is inserted into *source-string*, starting at the position that is identified by *start*. *insert-string* must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB. *source-string* and *insert-string* must have compatible data types.

start

An expression that returns an integer. The integer specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The value of the integer must be in the range of 1 to the length of *source-string* plus one. If OCTETS is specified and the result is graphic data, the value must be an odd value between 1 and twice the length of *source-string* plus one.

length

An expression that specifies the length of the string to replace in *source-string* starting at *start*. *length* must be an expression that returns a value of the built-in INTEGER data type. *length* is expressed in the string unit specified, and the value must be in the range of 0 to the length of *source-string*. If OCTETS is specified and the result is graphic data, *length* must be even and be between 0 and twice the length of *source-string*. Not specifying *length* is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying *length* is equivalent to specifying a value of 2.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the units that are used to express *start* and *length* in the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified. If *source-string* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is a graphic string, a string unit is two bytes. For ASCII and EBCDIC data, this corresponds to a double byte character. For Unicode, this corresponds to a UTF-16 code point. Otherwise, a string unit is a byte.

CODEUNITS16

Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *start* and *length* are expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76. *length* must be an even number if *source-string* is graphic data and OCTETS is specified

If *source-string* and *insert-string* have different CCSID sets, *insert-string* (the string to be inserted) is converted to the CCSID of *source-string* (the source string).

The encoding scheme of the result is the same as *source-string*. The data type of the result of the function depends on the data type of *source-string* and *insert-string*:

- VARCHAR if *source-string* is a character string. The CCSID of the result depends on the arguments:
 - If either *source-string* or *insert-string* is character bit data, the result is bit data.
 - If both *source-string* and *insert-string* are SBCS:
 - If both *source-string* and *insert-string* are SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.
 - If *source-string* is SBCS Unicode data and *insert-string* is not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
 - Otherwise, the CCSID of the result is the same as the CCSID of *source-string*.
 - Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *source-string*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *source-string*.
- VARGRAPHIC if *source-string* is a graphic. The CCSID of the result is the same as the CCSID of *source-string*.
- VARBINARY if *source-string* and *insert-string* are both binary strings.

The length attribute of the result depends on the arguments:

- If *start* and *length* are constants, the length attribute of the result is:
$$L1 - \text{MIN}((L1 - V2 + 1), V3) + L4$$

where:

L1 is the length attribute of *source-string*

V2 is the value of *start*

V3 is the value of *length*

L4 is the length attribute of *insert-string*

- Otherwise, the length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*. In this case, the length attribute of *source-string* plus the length attribute of *insert-string* must not exceed 32704 for a VARCHAR result or 16352 for a VARGRAPHIC result.

If CODEUNITS16 or CODEUNITS32 is specified, the insert operation is performed on a Unicode version of the data. If needed, the data is converted to an intermediate form in order to evaluate the function. If an intermediate form is used, the actual length of the result depends on the original data (*source-string* and *insert-string*), and the representation of that data in Unicode. See “Determining the length attribute of the final result” on page 79 for more information on how to calculate the length attribute of the result string.

If CODEUNITS16 or CODEUNITS32 are not specified, the actual length of the result is:

$$A1 - \text{MIN}((A1 - V2 + 1), V3) + A4$$

where:

A1 is the actual length of *source-string*

V2 is the value of *start*

V3 is the value of *length*

A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error occurs.

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

Example 1: The following example shows how the string 'INSERTING' can be changed into other strings. The use of the CHAR function limits the length of the resulting string to 10 bytes.

```
SELECT CHAR(OVERLAY('INSERTING','IS',4,2,OCTETS),10),  
       CHAR(OVERLAY('INSERTING','IS',4,0,OCTETS),10),  
       CHAR(OVERLAY('INSERTING','',4,2,OCTETS),10)  
FROM SYSIBM.SYSDUMMY1;
```

This example returns 'INSISTING ', 'INSISERTIN', and 'INSTING '.

Example 2: Use the OVERLAY function to insert the character 'C' into the Unicode string '&N~AB', where '&' is the character for the musical symbol, G CLEF, and '~' is the character for combining tilde. The following table shows the Unicode string in different Unicode encoding forms:

Unicode format	&	N	~	A	B
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume the host variable *UTF8_VAR* contains the UTF-8 representation of

'&N~AB', and *UTF16_VAR* contains the UTF-16 representation of '&N~AB'. Then the following SELECT statement is run:

```
SELECT OVERLAY (:UTF8_VAR, 'C', 1, CODEUNITS16),
       OVERLAY (:UTF8_VAR, 'C', 1, CODEUNITS32),
       OVERLAY (:UTF8_VAR, 'C', 1, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

This statement returns the following values:

```
C N~AB
CN~AB
C?N~AB -- ? is the invalid UTF-8 sequence X'9D849E'
```

Assume that the previous SELECT statement was not run, but the following SELECT statement is run:

```
SELECT OVERLAY (:UTF8_VAR, 'C', 5, CODEUNITS16),
       OVERLAY (:UTF8_VAR, 'C', 5, CODEUNITS32),
       OVERLAY (:UTF8_VAR, 'C', 5, OCTETS)
FROM SYSIBM.SYSDUMMY1;
```

This statement returns the values:

```
&N~CB
&N~AC
&C~AB
```

Assume that the previous SELECT statement was not run, but the following SELECT statement is run:

```
SELECT OVERLAY (:UTF16_VAR, 'C', 1, CODEUNITS16),
       OVERLAY (:UTF16_VAR, 'C', 1, CODEUNITS32)
FROM SYSIBM.SYSDUMMY1;
```

This statement returns the values:

```
C?N~AB
CN~AB
```

Assume that the previous SELECT statement was not run, but the following SELECT statement is run:

```
SELECT OVERLAY (:UTF16_VAR, 'C', 5, CODEUNITS16),
       OVERLAY (:UTF16_VAR, 'C', 5, CODEUNITS32),
FROM SYSIBM.SYSDUMMY1;
```

This statement returns the values:

```
&N~CB
&N~AC
```

POSITION

The POSITION function returns the position of the first occurrence of an argument within another argument, where the position is expressed in terms of the string units that are specified.

►►—POSITION—(—*search-string*—,—*source-string*—,—CODEUNITS16
CODEUNITS32
OCTETS)—►►

The schema is SYSIBM.

If *search-string* is not found and neither argument is null, the result is 0. If *search-string* is found, the result is a number from 1 to the actual length of *source-string*, expressed in the units that are explicitly specified.

search-string

An expression that specifies the string for which to search. *search-string* must return a value that is any built-in string data type with an actual length that is no greater than 4000 bytes. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- An expression that concatenates (using CONCAT or ||) any of the above
- CAST specification
- A column name

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

source-string

An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is any built-in string data type. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A column name
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit that is used to express the result. If *source-string* is a character string that is defined as bit data, CODEUNITS16, or CODEUNITS32 cannot be specified. If *source-string* is a graphic string, OCTETS cannot be specified.

CODEUNITS16

Specifies that the result is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that the result is expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that the result is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

The first and second arguments must have compatible string types. For more information on compatibility, see “Conversion rules for operations that combine strings”.

If the search string and source string have different CCSID sets, then the *search-string* is converted to the CCSID set of the source string. If either CODEUNITS16 or CODEUNITS32 is specified, the function might be evaluated on a temporary copy of the data in Unicode.

The strings can contain mixed data. If OCTETS is specified:

- For ASCII data, if the search string or source string contains mixed data, the search string is found only if the same combination of single-byte and double-byte characters are found in the source string in exactly the same positions.
- For EBCDIC data, if the search string or source string contains mixed data, the search string is found only if any shift-in or shift-out characters are found in the source string in exactly the same positions, ignoring any redundant shift characters.
- For UTF-8 data, if the search string or source string contains mixed data, the search string is found only if the same combination of single-byte and multi-byte characters are found in the source string in exactly the same position.

The result of the function is a large integer. If either of the arguments can be null, the result can be null. If either of the arguments is null, the result is the null value. The POSITION function accepts mixed data strings.

When the POSITION function is invoked with OCTETS, the function operates on a strict byte-count basis without regard to single-byte or double-byte characters.

If the CCSID of the search string is different than the CCSID of the source string, it is converted to the CCSID of the source string.

The value of the result is determined by applying these rules in the order in which they appear:

- If *search-string* has a length of zero, the result is 1.
- If *source-string* has a length of zero, the result is 0.
- If the value of *search-string* is equal to an identical length of substring of contiguous positions within the value of *source-string*, the result is the starting position of the first such substring within the source string value.
- Otherwise, the result is 0. This includes the case where *search-string* is longer than *source-string*.

Example1: Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD BEER' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

```

SELECT RECEIVED, SUBJECT, POSITION('GOOD BEER', NOTE_TEXT, OCTETS)
FROM IN_TRAY
WHERE POSITION('GOOD BEER', NOTE_TEXT, OCTETS) <> 0;

```

Example 2: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable *LOCATION* with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = POSITION('ß','Jürgen lives on Hegelstraße',CODEUNITS32);
```

The value of host variable *LOCATION* is set to 27.

Example 3: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable *LOCATION* with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = POSITION('ß','Jürgen lives on Hegelstraße',OCTETS);
```

The value of host variable *LOCATION* is set to 28.

Related reference

“LOCATE” on page 403

“LOCATE_IN_STRING” on page 406

“POSSTR” on page 469

POSSTR

The POSSTR function returns the position of the first occurrence of an argument within another argument.

►►—POSSTR(*source-string*,*search-string*)—►►

The schema is SYSIBM.

If *search-string* is not found and neither argument is null, the result is 0. If *search-string* is found, the result is a number from 1 to the actual length of *source-string*.

source-string

An expression that specifies the source string in which the search is to take place. *source-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A column name
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above

search-string

An expression that specifies the string for which to search. *search-string* must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable or a file reference variable)
- A scalar function whose arguments are any of the above (though nested function invocations cannot be used)
- A CAST specification whose arguments are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

The first and second arguments must have compatible string types. For more information on compatibility, see “Conversion rules for operations that combine strings” on page 122.

If the *search-string* and *source-string* have different CCSID sets, then the *search-string* is converted to the CCSID set of the *source-string*.

Both *search-string* and *source-string* have zero or more contiguous positions. For character strings and binary strings, a position is a byte. For graphic strings, a

position is a DBCS character. Graphic Unicode data is treated as UTF-16 data; a UTF-16 supplementary character takes two DBCS characters to represent and as such is counted as two DBCS characters.

The strings can contain mixed data.

- For ASCII data, if *search-string* or *source-string* contains mixed data, *search-string* is found only if the same combination of single-byte and double-byte characters are found in *source-string* in exactly the same positions.
- For EBCDIC data, if *search-string* or *source-string* contains mixed data, *search-string* is found only if any shift-in or shift-out characters are found in *source-string* in exactly the same positions, ignoring any redundant shift characters.
- For UTF-8 data, if *search-string* or *source-string* contains mixed data, *search-string* is found only if the same combination of single-byte and multi-byte characters are found in *source-string* in exactly the same position.

POSSTR operates on a strict byte-count basis without regard to single-byte or double-byte characters. It is recommended that if either the *search-string* or *source-string* contains mixed data, POSITION should be used instead of POSSTR. The POSITION function operates on a character basis. In an EBCDIC encoding scheme, any shift-in and shift-out characters are not required to be in exactly the same position and their only significance is to indicate which characters are SBCS and which characters are DBCS.

The result of the function is a large integer. If either of the arguments can be null, the result can be null; if either of the arguments are null, the result is the null value. The value of the result is determined by applying these rules in the order in which they appear:

- If the length of *search-string* is zero, the result is 1.
- If the length of *source-string* is zero, the result is 0.
- If the value of *search-string* is equal to an identical length substring of contiguous positions from the value of *source-string*, the result is the starting position of the first such substring within the value of *source-string*.
- If none of the above conditions are met, the result is 0.

Example: Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD BEER' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

```
SELECT RECEIVED, SUBJECT, POSSTR(NOTE_TEXT, 'GOOD BEER')
FROM IN_TRAY
WHERE POSSTR(NOTE_TEXT, 'GOOD BEER') <> 0;
```

Related reference

“LOCATE” on page 403

“LOCATE_IN_STRING” on page 406

“POSITION” on page 466

POWER

The POWER function returns the value of the first argument to the power of the second argument.

►►—POWER(*numeric-expression-1*,*numeric-expression-2*)—◄◄

The schema is SYSIBM.

Each argument must be an expression that returns the value of any built-in numeric data type. If either argument includes a DECIMAL or REAL data type, but not a DECFLOAT data type, the arguments are converted to a double precision floating-point number for processing by the function. If either argument includes a DECFLOAT data type, the arguments are converted to DECFLOAT for processing by the function.

The result of the function depends on the data type of the arguments:

- If both arguments are SMALLINT or INTEGER, the result is INTEGER.
- If either argument is a DECFLOAT, the data type of the result is DECFLOAT(34).
 - If either argument is a DECFLOAT and one of the following statements is true, the result is NaN and an invalid operation condition:
 - both arguments are zero
 - the second argument has a non-zero fractional part
 - the second argument has more than 9 digits
 - the second argument is Infinite
- Otherwise, the result is DOUBLE.

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

Example 1: Assume that host variable *HPOWER* is INTEGER with a value of 3. The following statement returns the value 8.

```
SELECT POWER(2,:HPOWER)
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following statement returns the value 1.

```
SELECT POWER(0,0)
FROM SYSIBM.SYSDUMMY1;
```

QUANTIZE

The QUANTIZE function returns a DECFLOAT value that is equal in value (except for any rounding) and sign to the first argument and that has an exponent that is set to equal the exponent of the second argument.

►►—QUANTIZE(*expression-1*,*expression-2*)—►►

The schema is SYSIBM.

The number of digits that is returned (16 or 34) is the same as the number of digits in *expression-1*.

expression-1

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is not a DECFLOAT value, it is converted to DECFLOAT(34) for processing.

expression-2

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is not a DECFLOAT value, it is converted to DECFLOAT(34) for processing. *expression-2* is an expression that is used as an example pattern that will be used to rescale *expression-1*. The sign and coefficient of the second argument are ignored.

If one argument (after conversion) is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) argument is converted to DECFLOAT(34) before the function is processed.

The coefficient of the result is derived from that of *expression-1*. It is rounded, if necessary (if the exponent is being increased), multiplied by a power of ten (if the exponent is being decreased), or remains unchanged (if the exponent is already equal to that of *expression-2*).

For static SQL statements other than CREATE VIEW, the ROUNDING bind option or the native SQL procedure option determines the rounding mode.

For dynamic SQL statements (and static CREATE VIEW statements), the special register CURRENT DECFLOAT ROUNDING MODE determines the rounding mode.

Unlike other arithmetic operations on the DECFLOAT data type, if the length of the coefficient after the quantize operation is greater than the precision specified by *expression-2*, a warning occurs. This ensures that, unless there is an error condition, the exponent of the result of QUANTIZE is always equal to that of *expression-2*. Furthermore:

- If either argument is NaN, NaN is returned
- If either argument is sNaN, NaN is returned and an exception occurs
- If both arguments are infinity (positive or negative), infinity (positive or negative) is returned.
- If one argument is infinity (positive or negative) and the other argument is not infinity (positive or negative), NaN is returned and an exception occurs

The result of the function is a DECFLOAT(16) value if both arguments are DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. The result can be null; if any argument is null, the result is the null value.

Examples: The following examples illustrate the value that is returned for the QUANTIZE function given the input DECFLOAT values:

```
QUANTIZE(2.17, DECFLOAT(0.001)) = 2.170
QUANTIZE(2.17, DECFLOAT(0.01))  = 2.17
QUANTIZE(2.17, DECFLOAT(0.1))   = 2.2
QUANTIZE(2.17, DECFLOAT('1E+0')) = 2
QUANTIZE(2.17, DECFLOAT('1E+1')) = 0E+1
QUANTIZE(2, DECFLOAT(INFINITY))  = NAN -- exception
QUANTIZE(-0.1, DECFLOAT(1))      = 0
QUANTIZE(0, DECFLOAT('1E+5'))   = 0E+5
QUANTIZE(217, DECFLOAT('1E-1')) = 217.0
QUANTIZE(217, DECFLOAT('1E+0')) = 217
QUANTIZE(217, DECFLOAT('1E+1')) = 2.2E+2
QUANTIZE(217, DECFLOAT('1E+2')) = 2E+2
```

QUARTER

The QUARTER function returns an integer between 1 and 4 that represents the quarter of the year in which the date resides. For example, any dates in January, February, or March return the integer 1.

►►—QUARTER(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string data type, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 1: The following function returns 3 because August is in the third quarter of the year.

```
SELECT QUARTER('2008-08-25')
FROM SYSIBM.SYSDUMMY1
```

Example 2: Using sample table DSN8910.PROJ, set the integer host variable *QUART* to the quarter of the year in which activity number 70 for project 'AD3111' occurred. Activity completion dates are recorded in column ACENDATE.

```
SELECT QUARTER(ACENDATE)
  INTO :QUART
  FROM DSN8910.PROJ
 WHERE PROJNO = 'AD3111' AND ACTNO = 70;
```

QUART is set to 4.

RADIANS

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

►►—RADIANS(*numeric-expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *HDEG* is an INTEGER with a value of 180. The following statement returns a double precision floating-point number with an approximate value of 3.1415926536.

```
SELECT RADIANS(:HDEG)
FROM SYSIBM.SYSDUMMY1;
```

RAISE_ERROR

The RAISE_ERROR function causes the statement that invokes the function to return an error with the specified SQLSTATE (along with SQLCODE -438) and error condition. The RAISE_ERROR function always returns the null value with an undefined data type.

►—RAISE_ERROR(*sqlstate*,*diagnostic-string*)—►

The schema is SYSIBM.

sqlstate

An expression that returns a character string (CHAR or VARCHAR) of exactly 5 characters. The *sqlstate* value must follow these rules for application-defined SQLSTATES:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00', '01', or '02' because these are not error classes.
- If the SQLSTATE class (first two characters) starts with the character '0' through '6' or 'A' through 'H', the subclass (last three characters) must start with a letter in the range 'I' through 'Z'.
- If the SQLSTATE class (first two characters) starts with the character '7', '8', '9', or 'I' through 'Z', the subclass (last three characters) can be any of '0' through '9' or 'A' through 'Z'.

diagnostic-string

An expression that returns a character string with a data type of CHAR or VARCHAR and a length of up to 70 bytes. The string contains EBCDIC data that describes the error condition. If the string is longer than 70 bytes, it is truncated.


Since the data type of the result of RAISE_ERROR is undefined, it can only be used in a SET *host-variable* or SQL procedure language *assignment-statement*. To use this function in another context, such as alone in a select list, you must use a cast specification to give a data type to the null value that is returned. The RAISE_ERROR function is most useful with CASE expressions.

Example: For each employee in sample table DSN8910.EMP, list the employee number and education level. List the education level as 'Post Graduate', 'Graduate' and 'Diploma' instead of the integer that it is stored as in the table. If an education level is greater than '20', raise an error ('70001') with a description.

```
SELECT EMPNO,  
CASE WHEN EDLEVEL < 16 THEN 'Diploma'  
      WHEN EDLEVEL < 18 THEN 'Graduate'  
      WHEN EDLEVEL < 21 THEN 'Post Graduate'  
      ELSE RAISE_ERROR('70001',  
                      'EDUCLVL has a value greater than 20')  
END  
FROM DSN8910.EMP;
```

RAND

The RAND function returns a random floating-point value between 0 and 1. An argument can be specified as an optional seed value.



The diagram shows the syntax for the RAND function. It consists of the word "RAND" followed by an opening parenthesis "(", then a bracketed placeholder labeled "numeric-expression", and finally a closing parenthesis ")", all enclosed within a rectangular box with arrows at the ends.

The schema is SYSIBM.

numeric-expression

If *numeric-expression* is specified, it is used as the seed value. The argument must be an expression that returns a value of a built-in integer data type (SMALLINT or INTEGER). The value must be between 0 and 2,147,483,646.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

A specific seed value, other than zero, will produce the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. The seed value is used only for the first invocation of an instance of the RAND function within a statement. RAND(0) is processed the same as RAND().

The RAND function is a not deterministic.

Example: Assume that host variable *HRAND* is an INTEGER with a value of 100. The following statement returns a random floating-point number between 0 and 1, such as the approximate value .0121398:

```
SELECT RAND(:HRAND)
FROM SYSIBM.SYSDUMMY1;
```

To generate values in a numeric interval other than 0 to 1, multiply the RAND function by the size of the desired interval. For example, to get a random number between 0 and 10, such as the approximate value 5.8731398, multiply the function by 10:

```
SELECT (RAND(:HRAND) * 10)
FROM SYSIBM.SYSDUMMY1;
```

REAL

The REAL function returns a single-precision floating-point representation of either a number or a string representation of a number.

Numeric to Real:

►►—REAL(*numeric-expression*)—◄◄

String to Real:

►►—REAL(*string-expression*)—◄◄

The schema is SYSIBM.

Numeric to Real

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a single precision floating-point column or variable. If the numeric value of the argument is not within the range of single precision floating-point, an error occurs.

String to Real

string-expression

An expression that returns a value of a character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes. The string must contain a valid string representation of a number.

The result is the same number that would result from CAST(*string-expression* AS REAL). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL floating-point, integer, or decimal constant.

The result of the function is a single precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Recommendation: To increase the portability of applications, use the CAST specification. For more information, see “CAST specification” on page 202.

Example: Using sample table DSN8910.EMP, find the ratio of salary to commission for employees whose commission is not zero. The columns involved, SALARY and COMM, have decimal data types. To express the result in single precision floating-point, apply REAL to SALARY so that the division is carried out in floating-point (actually double precision) and then apply REAL to the complete expression so that the results are returned in single precision floating-point.


```
SELECT EMPNO, REAL(REAL(SALARY)/COMM)
FROM DSN8910.EMP
WHERE COMM > 0;
```

REPEAT

The REPEAT function returns a character string that is composed of an argument that is repeated a specified number of times.

►►—REPEAT(*expression*,*integer*)—◄◄

The schema is SYSIBM.

expression

An expression that specifies the string to be repeated. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB. The actual length of the string must be greater or equal to 1 and less than or equal to 32704 bytes.

integer

integer must be a positive large integer value that specifies the number of times to repeat the string.

If any argument can be null, the result can be null; if any argument is null, the result is the null value. The encoding scheme of the result is the same as *expression*. The data type of the result of the function depends on the data type of *expression*:

- VARBINARY if *expression* is a binary string
- VARCHAR if *expression* is a character string
- VARGRAPHIC if *expression* is graphic string

The CCSID of the result is the same as the CCSID of *expression*.

If *integer* is a constant, the length attribute of the result is the length attribute of *expression* times *integer*. Otherwise, the length attribute depends on the data type of the result:

- 4000 for VARBINARY and VARCHAR
- 2000 for VARGRAPHIC

The actual length of the result is the actual length of *expression* times *integer*. If the actual length of the result string exceeds the maximum for the return type, an error occurs.

Example 1: Repeat 'abc' two times to create 'abcabc'.

```
SELECT REPEAT('abc',2)
FROM SYSIBM.SYSDUMMY1;
```

Example 2: List the phrase 'REPEAT THIS' five times. Use the CHAR function to limit the output to 60 bytes.

```
SELECT CHAR(REPEAT('REPEAT THIS',5), 60)
FROM SYSIBM.SYSDUMMY1;
```

This example results in 'REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS'.

Example 3: For the following query, the LENGTH function returns a value of 0 because the result of repeating a string zero times is an empty string, which is a zero-length string.

```
SELECT LENGTH(REPEAT('REPEAT THIS',0))  
FROM SYSIBM.SYSDUMMY1;
```

Example 4: For the following query, the LENGTH function returns a value of 0 because the result of repeating an empty string any number of times is an empty string, which is a zero-length string.

```
SELECT LENGTH(REPEAT('', 5))  
FROM SYSIBM.SYSDUMMY1;
```

REPLACE

The REPLACE function replaces all occurrences of *search-string* in *source-string* with *replace-string*. If *search-string* is not found in *source-string*, *source-string* is returned unchanged.

►►—REPLACE(*source-string*,*search-string*,*replace-string*)—►►

The schema is SYSIBM.

source-string

An expression that specifies the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB and it cannot be an empty string. The length of *source-string* must be greater than or equal to the length of *search-string*.

search-string

An expression that specifies the string to be removed from the source string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB; the value cannot be an empty string.

replace-string

An expression that specifies the replacement string. The expression must return a value that is a built-in character string, graphic string, or binary string data type that is not a LOB. If the expression is an empty string, nothing replaces the string that is removed from the source string.

The actual length of each string must be 32704 bytes or less for character and binary strings or 16352 or less for graphic strings.

All three arguments must have compatible data types. If the expressions have different CCSID sets, then the expressions are converted to the CCSID set of *source-string*.

The data type of the result of the function depends on the data type of *source-string*, *search-string*, and *replace-string*:

- VARCHAR if *source-string* is a character string. The encoding scheme of the result is the same as *source-string*. The CCSID of the result depends on the arguments:
 - If *source-string*, *search-string*, or *replace-string* is bit data, the result is bit data.
 - If *source-string*, *search-string*, and *replace-string* are all SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.
 - If *source-string* is SBCS Unicode data, and *search-string* or *replace-string* is not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
 - Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *source-string*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *source-string*.

- VARGRAPHIC if *source-string* is a graphic. The encoding scheme of the result is the same as *source-string*. The CCSID of the result is the same as the CCSID of *source-string*.
- VARBINARY if *source-string*, *search-string*, and *replace-string* are binary strings.

The length attribute of the result depends on the arguments:

- If the length attribute of *replace-string* is less than or equal to the length attribute of *search-string*, the length attribute of the result is the length attribute of *source-string*.
- If the length attribute of *replace-string* is greater than the length attribute of *search-string*, the length attribute of the result is determined as follows depending on the data type of the result:
 - For VARCHAR or VARBINARY:
 - If $L1 \leq 4000$, the length attribute of the result is $\text{MIN}(4000, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$
 - Otherwise, the length attribute of the result is $\text{MIN}(32704, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$
 - For VARGRAPHIC:
 - If $L1 \leq 2000$, the length attribute of the result is $\text{MIN}(2000, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$
 - Otherwise, the length attribute of the result is $\text{MIN}(16352, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$

where:

$L1$ is the length attribute of *source-string*

$L2$ is the length attribute of *search-string* if the search string is a string constant. Otherwise, $L2$ is 1.

$L3$ is the length attribute of *replace-string*

If the result is a character string or binary string, the length attribute of the result must not exceed 32704. If the result is a graphic string, the length attribute of the result must not exceed 16352.

The actual length of the result is the actual length of *source-string* plus the number of occurrences of *search-string* that exist in *source-string* multiplied by the actual length of *replace-string* minus the actual length of *search-string*. If the actual length of the result string exceeds the maximum for the return data type, an error occurs.

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

Example 1: Replace all occurrences of the character 'N' in the string 'DINING' with 'VID'. Use the CHAR function to limit the output to 10 bytes.

```
SELECT CHAR(REPLACE('DINING', 'N', 'VID'), 10)
FROM SYSIBM.SYSDUMMY1;
```

The result is the string 'DIVIDIVIDG'.

Example 2: Replace string 'ABC' in the string 'ABCXYZ' with nothing, which is the same as removing 'ABC' from the string.

```
SELECT REPLACE('ABCXYZ', 'ABC', '')
FROM SYSIBM.SYSDUMMY1;
```

The result is the string 'XYZ'.

Example 3: Replace string 'ABC' in the string 'ABCCABCC' with 'AB'. This example illustrates that the result can still contain the string that is to be replaced (in this case, 'ABC') because all occurrences of the string to be replaced are identified prior to any replacement.

```
SELECT REPLACE('ABCCABCC','ABC','AB')
FROM SYSIBM.SYSDUMMY1;
```

The result is the string 'ABCABC'.

RID

The RID function returns the record ID (RID) of a row. The RID is used to uniquely identify a row.

►►—RID(*table-designator*)—◄◄

The schema is SYSIBM.

The function might return a different value when it is invoked multiple times for a row. For example, after the REORG utility is run, the RID function might return a different value for a row than would have been returned prior to the REORG utility being run. The RID function is not deterministic.

table-designator

table-designator must uniquely identify a base table, a view, or a nested table expression of a subselect in which the function is referenced.

If *table-designator* specifies a view or a nested table expression, the RID function returns the RID of the base table of the view or nested table expression. The specified view or nested table expression must contain only one base table in its outer subselect. *table-designator* must not specify a view or a nested table expression that is materialized.

The result of the function is BIGINT. The result can be null.

Considerations for RID values: DB2 might reuse RID numbers when a REORG operation is performed. If the RID function is used to obtain a value for a row and an application depends on that value remaining the same as long as the row exists, consider the following alternatives:

- Add a ROWID column to the table to provide a value that can be associated with each row, rather than invoking the RID function to generate a value for a row.
- Define a primary key for the table, using the columns of the primary key to ensure uniqueness, rather than invoking the RID function to generate a value for a row.

Example 1: Return the RID and last name of employees who are in department '20':

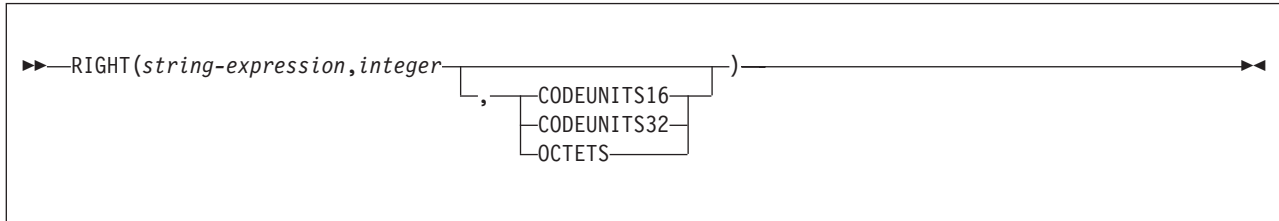
```
SELECT RID(EMP), LASTNAME
FROM EMP
WHERE DEPTNO = '20';
```

Example 2: Set the host variable *HV_EMP_RID* as the value of the RID for the employee with the employee number of '3500':

```
SELECT RID(EMP) INTO :HV_EMP_RID
FROM EMP
WHERE EMPNO = '3500';
```

RIGHT

The RIGHT function returns a string that consists of the specified number of rightmost bytes or specified string unit from a string.



The schema is SYSIBM.

string-expression

An expression that specifies the string from which the result is derived. The string must be any built-in string data type. A substring of *string-expression* is zero or more contiguous code points of *string-expression*.

The string can contain mixed data. Depending on the units that are specified to evaluate the function, the result is not necessarily a properly formed mixed data character string.

integer

An expression that specifies the length of the result. The value must be an integer between 0 and *n*, where *n* is the length attribute of *string-expression*, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the unit that is used to express *integer*. If *string-expression* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified. If *string-expression* is a graphic string, OCTETS cannot be specified. If *string-expression* is a binary string, CODEUNITS16, CODEUNITS32, and OCTETS cannot be specified.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *integer* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

The *string-expression* is effectively padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The encoding scheme of the data determines the padding character:

- For ASCII SBCS data or ASCII mixed data, the padding character is X'20'.

- For ASCII DBCS data, the padding character depends on the CCSID; for example, for Japanese (CCSID 301) the padding character is X'8140', while for simplified Chinese it is X'A1A1'.
- For EBCDIC SBCS data or EBCDIC mixed data, the padding character is X'40'.
- For EBCDIC DBCS data, the padding character is X'4040'.
- For Unicode SBCS data or UTF-8 data (Unicode mixed data), the padding character is X'20'.
- For UTF-16 data (Unicode DBCS data), the padding character is X'0020'.
- For binary data, the padding character is X'00'.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *string-expression* and a data type that depends on the data type of *string-expression*:

- VARBINARY if *string-expression* is BINARY or VARBINARY
- VARCHAR if *string-expression* is CHAR or VARCHAR
- CLOB if *string-expression* is CLOB
- VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string-expression* is DBCLOB
- BLOB if *string-expression* is BLOB

The actual length of the result is determined from *integer*.

If any argument of the function can be null, the result can be null; if any argument is null, the result is the null value. The CCSID of the result is the same as that of *string-expression*.

Example 1: Assume that host variable *ALPHA* has a value of 'ABCDEF'. The following statement returns the value 'DEF', which are the three rightmost characters in *ALPHA*.

```
SELECT RIGHT(ALPHA,3)
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following statement returns a zero length string.

```
SELECT RIGHT('ABCABC',0)
FROM SYSIBM.SYSDUMMY1;
```

Example 3: FIRSTNME is a VARCHAR(12) column in table T1. When FIRSTNME has the 6-character string 'Jürgen' as a value:

Function ...	Returns ...
RIGHT(FIRSTNME,5,CODEUNITS32)	'ürgen' -- x'C3BC7267656E'
RIGHT(FIRSTNME,5,CODEUNITS16)	'ürgen' -- x'C3BC7267656E'
RIGHT(FIRSTNME,5,OCTETS)	' rgen' -- x'207267656E' a truncated string

ROUND

The ROUND function returns a number that is rounded to the specified number of places to the right or left of the decimal place.

►►—ROUND(*numeric-expression-1*,*numeric-expression-2*)—◄◄

The schema is SYSIBM.

numeric-expression-1

An expression that returns a value of any built-in numeric data type.

If *expression-1* is a decimal floating-point data type, the DECFLOAT ROUNDING MODE will not be used. The rounding behavior of ROUND corresponds to a value of ROUND_HALF_UP. If a different rounding behavior is desired, use the QUANTIZE function.

numeric-expression-2

An expression that returns a value of a built-in small integer data type or large integer data type.

The absolute value of integer specifies the number of places to the right of the decimal point for the result if *numeric-expression-2* is not negative. If *numeric-expression-2* is negative, *numeric-expression-1* is rounded to the sum of the absolute value of *numeric-expression-2*+1 number of places to the left of the decimal point.

If the absolute value of *numeric-expression-2* is larger than the number of digits to the left of the decimal point, the result is 0. (For example, ROUND(748.58,-4) returns 0.)

If *numeric-expression-1* is positive, a digit value of 5 is rounded to the next higher positive number. If *numeric-expression-1* is negative, a digit value of 5 is rounded to the next lower negative number.

The result of the function has the same data type and length attribute as the first argument except that the precision is increased by one if the argument is DECIMAL and the precision is less than 31. For example, an argument with a data type of DECIMAL(5,2) results in DECIMAL(6,2). An argument with a data type of DECIMAL(31,2) results in DECIMAL(31,2).

The result can be null. If any argument is null, the result is the null value.

Example 1: Calculate the number '873.726' rounded to '2', '1', '0', '-1', and '-2' decimal places respectively.

```
SELECT ROUND(873.726,2),
       ROUND(873.726,1),
       ROUND(873.726,0),
       ROUND(873.726,-1),
       ROUND(873.726,-2),
       ROUND(873.726,-3),
       ROUND(873.726,-4),
FROM SYSIBM.SYSDUMMY1;
```

This example returns the values '0873.730', '0873.700', '0874.000', '0870.000', '0900.000', '1000.000', and '0000.000'.

Example 2: To demonstrate how numbers are rounded in positive and negative values, calculate the numbers '3.5', '3.1', '-3.1', '-3.5' rounded to '0' decimal places.

```
SELECT ROUND(3.5,0),  
       ROUND(3.1,0),  
       ROUND(-3.1,0),  
       ROUND(-3.5,0)  
FROM SYSIBM.SYSDUMMY1;
```

This example returns the values '04.0', '03.0', '-03.0', and '-04.0'. (Notice that in the positive value '3.5' is rounded up to the next higher number while in the negative value '-3.5' is rounded down to the next lower negative number.)

ROUND_TIMESTAMP

The ROUND_TIMESTAMP scalar function returns a timestamp that is rounded to the unit that is specified by the timestamp format string.

```

>> ROUND_TIMESTAMP ( expression [ , 'DD' ] [ , format-string ] )

```

The schema is SYSIBM.

expression

An expression that returns a value of any of the following built-in data types: a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

format-string

An expression that returns a built-in character string or graphic string data type, with a length that is not greater than 255 bytes. *format-string* contains a template of how the timestamp represented by *expression* should be rounded. For example, if *format-string* is 'DD', the timestamp that is represented by *expression* is rounded to the nearest day. *format-string* must be a valid template for a timestamp, and not include leading or trailing blanks.

Allowable values for *format-string* are listed in the following table.

Table 57. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
CC SCC	Century	Input Value: 1897-12-04-12.22.22.000000	Input Value: 1897-12-04-12.22.22.000000
	Rounds up to the start of the next century after the 50th year of the century (for example on 1951-01-01-00.00.00).	Result: 1900-01-01-00.00.00.000000	Result: 1800-01-01-00.00.00.000000
	Not valid for a TIME argument.		
YYYYY YYYY YEAR SYEAR YYY YY Y	Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-01-00.00.00.000000
IYYY IYY IY I	ISO Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-01-00.00.00.000000

Table 57. *ROUND_TIMESTAMP* and *TRUNC_TIMESTAMP* format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
Q	Quarter (Rounds up on the sixteenth day of the second month of the quarter)	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-04-01-00.00.00.000000
MONTH MON MM RM	Month (Rounds up on the sixteenth day of the month)	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-18-12.15.00.000000 Result: 1999-06-01-00.00.00.000000
WW	Same day of the week as the first day of the year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-06-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-04-29-00.00.00.000000
IW	Same day of the week as the first day of the ISO year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the ISO year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-08-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-05-01-00.00.00.000000
W	Same day of the week as the first day of the month (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the month)	Input Value: 2000-05-17-12.12.30.000000 Result: 2000-05-15-00.00.00.000000	Input Value: 2000-05-17-12.15.00.000000 Result: 2000-05-15-00.00.00.000000
DDD DD J	Day (Rounds up on the 12th hour of the day)	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-17-00.00.00.000000
DAY DY D	Starting day of the week (Rounds up with respect to the 12th hour of the third day of the week. The first day of the week is always Sunday).	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-21-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-14-00.00.00.000000
HH HH12 HH24	Hour (Rounds up at 30 minutes)	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-17-23.00.00.000000
MI	Minute (Rounds up at 30 seconds)	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.59.00.000000	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.58.00.000000
SS	Second (Rounds up at 500000 microseconds)	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.46.000000	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.45.000000

The result of the function is a **TIMESTAMP**.

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

| The result CCSID is the appropriate CCSID of the argument encoding scheme and
| the result subtype is the appropriate subtype of the CCSID.

Example 1: Set the host variable *RND_TMSTMP* with the input timestamp rounded to the nearest year value.

```
SET :RND_TMSTMP = ROUND_TIMESTAMP(TIMESTAMP_FORMAT('2000-08-14 17:30:00',  
                                                    'YYYY-MM-DD HH24:MI:SS'), 'YEAR');
```

The value set is '2001-01-01-00.00.00.000000'.

ROWID

The ROWID function returns a row ID representation of its argument.

►►—ROWID(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of a built-in character string data type, other than a CLOB, with a maximum length that is no greater than 255 bytes. Although the character string can contain any value, it is recommended that the character string contain a ROWID value that was previously generated by DB2 to ensure a valid ROWID value is returned. For example, the function can be used to convert a ROWID value that was cast to a CHAR value back to a ROWID value.

If the actual length of *expression* is less than 40, the result is not padded. If the actual length of *expression* is greater than 40, the result is truncated. If non-blank characters are truncated, a warning is returned.

The result of the function is a ROWID value.

The length attribute of the result is 40. The actual length of the result is the length of *expression*.

If the argument can be null, the result can be null; if the argument is null, the result is the null value. However, a null ROWID value cannot be used as the value for a row ID column in the database.

Example: Assume that table EMPLOYEE contains a row ID column, 'EMP_ROWID'. Also assume that the table contains a row that is identified by a ROWID value that is equivalent to X'F0DFD230E3C0D80D81C201AA0A28010000000000203'. Using direct row access, select the employee number for that row.

```
SELECT EMPNO
FROM EMPLOYEE
WHERE EMP_ROWID=ROWID(X'F0DFD230E3C0D80D81C201AA0A28010000000000203');
```

RPAD

The RPAD function returns a string that is padded on the right with blanks or a specified string.

The diagram shows the function signature `RPAD(string-expression, integer [, pad])` enclosed in a box. A horizontal line with arrowheads at both ends extends from the closing parenthesis to the right. A bracket is positioned below the `integer` and `[, pad]` arguments, indicating they are optional.

The schema is SYSIBM.

The RPAD function returns a string composed of *string-expression* padded on the right, with *pad* or blanks. The RPAD function treats leading or trailing blanks in *string-expression* as significant. Padding will only occur if the actual length of *string-expression* is less than *integer*, and *pad* is not an empty string.

string-expression

An expression that specifies the source string. The expression must return a value that is a built-in string data type that is not a LOB.

integer

An integer constant that specifies the length of the result. The value must be zero or a positive integer that is less than or equal to *n*, where *n* is 32704 if *string-expression* is a character or binary string, or where *n* is 16352 if *string-expression* is a graphic string.

pad

An expression that specifies the string with which to pad. The expression must return a value that is a built-in string data type that is not a LOB. If *pad* is not specified, the pad character is determined as follows:

- SBCS blank character if *string-expression* is a character string.
- DBCS blank character if *string-expression* is a graphic string.
- Hexadecimal zero (X'00'), if *string-expression* is a binary string.

The result of the function is a varying length string that has the same CCSID of *string-expression*. *string-expression* and *pad* must have compatible data types. If the string expressions have different CCSID sets, then *pad* is converted to the CCSID set of *string-expression*. If either *string-expression* or *pad* is FOR BIT DATA, no character conversion occurs. The actual length of the result is determined from *integer*.

The length attribute of the result depends on *integer*. If *integer* is greater than 0, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the result is 1.

The actual length of the result is determined from *integer*. If *integer* is 0, the actual length is 0, and the result is the empty string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

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

Example 1: Assume that NAME is a VARCHAR(15) column that contains the values 'Chris', 'Meg', and 'Jeff'. The following query will completely pad out a value on the right with periods.

```
SELECT RPAD(NAME,15,'.' ) AS NAME
FROM T1;
```

The results are similar to the following:

```
NAME
-----
Chris.....
Meg.....
Jeff.....
```

Example 2: Similar to Example 1, the following query will completely pad out a value on the right with *pad* (note that in some cases there is a partial instance of the padding specification):

```
SELECT RPAD(NAME,15,'123' ) AS NAME
FROM T1;
```

The results are similar to the following:

```
NAME
-----
Chris1231231231
Meg123123123123
Jeff12312312312
```

Example 3: Similarly, the following query will only pad each value to a length of 5:

```
SELECT RPAD(NAME,5,'.') AS NAME
FROM T1;
```

The results are similar to the following:

```
NAME
-----
Chris
Meg..
Jeff.
```

Example 4: Assume that NAME is a CHAR(15) column that contains the values 'Chris', 'Meg', and 'Jeff'. Note that the result of RTRIM in the following example is a varying length string with the blanks removed:

```
SELECT RPAD(RTRIM(NAME),15,'.') AS NAME
FROM T1;
```

The results are similar to the following:

```
NAME
-----
Chris.....
Meg.....
Jeff.....
```

RTRIM

The RTRIM function removes blanks or hexadecimal zeros from the end of a string expression.

►►—RTRIM(*string-expression*)—◄◄

The schema is SYSIBM.

The RTRIM function returns the same results as the STRIP function with TRAILING specified:

STRIP(*string-expression*,TRAILING)

The argument must be an expression that returns a value that is a built-in character string data type, graphic data type, or binary string data type. The argument must not be a LOB.

The trailing blanks or hexadecimal zeros depend on the encoding scheme of the data and the data type:

- If the argument is a graphic string, then the trailing DBCS blanks are removed.
 - For ASCII, the ASCII CCSID determines the hex value that represents a double-byte blank. For example, for Japanese (CCSID 301), X'8140' represents a double-byte blank, while it is X'A1A1' for Simplified Chinese.
 - For EBCDIC-encoded data, X'4040' represents a double-byte blank.
 - For Unicode-encoded data, X'0020' represents a double-byte blank.
- If the argument is a binary string, the trailing hexadecimal zeros (BX'00') are removed.
- Otherwise, the trailing SBCS blanks are removed.
 - For data that is encoded in ASCII, X'20' represents a blank.
 - For EBCDIC-encoded data, X'40' represents a blank.
 - For Unicode-encoded data, X'20' represents an SBCS or UTF-8 blank.

The result of the function depends on the data type of its argument:

- VARBINARY if *string-expression* is a binary string
- VARCHAR if *string-expression* is a character string
- VARGRAPHIC if *string-expression* is a graphic string

The length attribute of the result is the same as the length attribute of *string-expression*. The actual length of the result is the length of the expression minus the number of characters removed. If all of the characters are removed, the result is an empty string.

If the argument can be null, the result can be null; if the argument is null, the result is the null value. The CCSID of the result is the same as that of *string-expression*.

Example 1: Assume that host variable *HELLO* is defined as CHAR(9) and has a value of 'Hello '.

```
SELECT RTRIM(:HELLO)
FROM SYSIBM.SYSDUMMY1;
```

This example removes the trailing blanks and results in 'Hello'.

Example 2: The following function returns a VARBINARY string with a length attribute 10, actual length 6 and a value BX'C1D5C4D9C5C9'.

```
SELECT RTRIM(BINARY(X'C1D5C4D9C5C900000000'))
FROM SYSIBM.SYSDUMMY1;
```

- 498 SQL Reference

The options that can be specified as part of the *search-argument-options* are as follows:

QUERYLANGUAGE = *value*

Specifies the query language. The value can be any of the supported language codes. If the QUERYLANGUAGE option is not specified, the default is the language value of the text search index that is used when this function is invoked. If the language value of the text search index is AUTO, the default value for QUERYLANGUAGE is en_US.

RESULTLIMIT = *value*

Specifies the maximum number of results that are to be returned from the underlying search engine. The *value* can be an integer value between 1 and 2 147 483 647. If the RESULTLIMIT option is not specified, no result limit is in effect for the query.

This scalar function cannot be called for each row of the result table, depending on the plan that the optimizer chooses. This function can be called once for the query to the underlying search engine, and a result set of all of the primary keys that match are returned from the search engine. This result set is then joined to the table containing the column to identify the result rows. In this case, the RESULTLIMIT value acts like a FETCH FIRST ?? ROWS from the underlying text search engine and can be used as an optimization. If the search engine is called for each row of the result because the optimizer determines that is the best plan, then the RESULTLIMIT option has no effect.

SYNONYM = OFF or SYNONYM = ON

Specifies whether to use a synonym dictionary that is associated with the text search index. Use the Synonym Tool to add a synonym dictionary to the collection. The default is OFF.

OFF Do not use a synonym dictionary.

ON Use the synonym dictionary that is associated with the text search index.

The result of the function is a double-precision floating-point number. If the second argument can be null, the result can be null. If the second argument is null, the result is the null value. If the third argument is null, the result is as if the third argument was not specified.

The result is greater than 0 but less than 1 if the column contains a match for the search criteria that the search argument specifies. The better a document matches the query, the more relevant the score and the larger the result value. If the column does not contain a match, the result is 0.

SCORE is a non-deterministic function.

Example

The following statement generates a list of employees in the order of how well their resumes matches the query "programmer AND (java OR cobol)", along with a relevance value that is normalized between 0 (zero) and 100.

```
SELECT EMPNO, INTEGER(SCORE(RESUME, 'programmer AND
(java OR cobol)') * 100) AS RELEVANCE
FROM EMP_RESUME
WHERE RESUME_FORMAT = 'ascii'
AND CONTAINS(RESUME, 'programmer AND (java OR cobol)') = 1
ORDER BY RELEVANCE DESC
```

| DB2 first evaluates the CONTAINS predicate in the WHERE clause, and therefore,
| does not evaluate the SCORE function in the SELECT list for every row of the
| table. In this case, the arguments for SCORE and CONTAINS must be identical.

SECOND

The SECOND function returns the seconds part of a value.

►►—SECOND(*expression*)—◄◄

The schema is SYSIBM.

expression

expression must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String representations of datetime values” on page 89.
- If *expression* is a number, it must be a time or timestamp duration. For the valid formats of time and timestamp durations, see “Datetime operands” on page 121.

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:

If the argument is a time, timestamp, or string representation of a time or a timestamp:

The result is the seconds part of the value, 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.

Example 1: Assume that the variable *TIME_DUR* is declared in a PL/I program as DECIMAL(6,0) and can therefore be interpreted as a time duration. When *TIME_DUR* has the value 153045, the following function returns the value 45.

```
SECOND(:TIME_DUR)
```

Example 2: Assume that *RECEIVED* is a *TIMESTAMP* column and that one of its values is the internal equivalent of '1988-12-25-17.12.30.000000'. The following function returns the value 30.

```
SECOND(RECEIVED)
```

SIGN

The SIGN function returns an indicator of the sign of the argument.

►►—SIGN(*numeric-expression*)—◄◄

The schema is SYSIBM.

The returned value is one of the following values:

- 1 if the argument is less than zero
- 0 if the argument is DECFLOAT negative zero
- 0 if the argument is zero
- 1 if the argument is greater than zero

The argument must be an expression that returns a value of any built-in numeric data type, except DECIMAL(31,31).

The result has the same data type and length attribute as the argument, except that precision is increased by one if the argument is DECIMAL and the scale of the argument is equal to its precision. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(6,5).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *PROFIT* is a large integer with a value of 50000.

```
SELECT SIGN(:PROFIT)
FROM SYSIBM.SYSDUMMY1;
```

This example returns the value 1.

SIN

The SIN function returns the sine of the argument, where the argument is an angle, expressed in radians.

►►—SIN(*numeric-expression*)—◄◄

The schema is SYSIBM.

The SIN and ASIN functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *SINE* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 0.99.

```
SELECT SIN(:SINE)
FROM SYSIBM.SYSDUMMY1;
```

SINH

The SINH function returns the hyperbolic sine of the argument, where the argument is an angle, expressed in radians.

►►—SINH(*numeric-expression*)—◄◄

The schema is SYSIBM.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *HSINE* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 2.12.

```
SELECT SINH(:HSINE)
FROM SYSIBM.SYSDUMMY1;
```

SMALLINT

The SMALLINT function returns a small integer representation either of a number or of a string representation of a number.

Numeric to Smallint:

►►—SMALLINT(*numeric-expression*)—►►

String to Smallint:

►►—SMALLINT(*string-expression*)—►►

The schema is SYSIBM.

Numeric to Smallint

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. If present, the decimal part of the argument is truncated.

String to Smallint

string-expression

An expression that returns a value of character or graphic string (except a CLOB or DBCLOB) with a length attribute that is not greater than 255 bytes for a character string or 127 for a graphic string. The string must contain a valid string representation of a number.

The result is the same number that would result from CAST(*string-expression* AS SMALLINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. If the whole part of the argument is not within the range of small integers, an error is returned.

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.

Recommendation: To increase the portability of applications, use the CAST specification. For more information, see “CAST specification” on page 202.

Example: Using sample table DSN8910.EMP, find the average education level (EDLEVEL) of the employees in department 'A00'. Round the result to the nearest full education level.

```
SELECT SMALLINT(AVG(EDLEVEL)+.5)
FROM DSN8910.EMP
WHERE DEPT = 'A00';
```

Assuming that the five employees in the department have education levels of '19', '18', '14', '18', and '14', the result is '17'.

SOUNDEX

The SOUNDEX function returns a 4-character code that represents the sound of the words in the argument. The result can be compared to the results of the SOUNDEX function of other strings.

►►—SOUNDEX(*expression*)—◄◄

The schema is SYSIBM.

expression

An *expression* that must return a value of any built-in numeric, character, or graphic string data type that is not a LOB. A numeric, mixed character, or graphic string value is cast to a Unicode SBCS character string before the function is evaluated. For more information about converting numeric data to a character string, see “VARCHAR” on page 558. For more information about converting mixed or graphic strings to Unicode SBCS, see “CAST specification” on page 202.

The data type of the result is CHAR(4). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the Unicode SBCS CCSID.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search for words with similar sounds. The comparison of words can be done directly or by passing the strings as arguments to the DIFFERENCE function. For more information, see “DIFFERENCE” on page 351.

Example 1: Use the SOUNDEX function to find a row where the sound of the LASTNAME value closely matches the phonetic spelling of 'Loucesy':

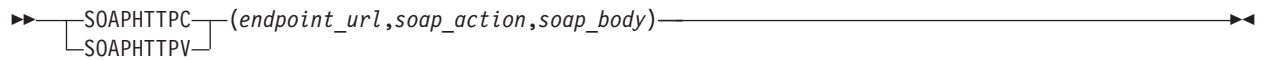
```
SELECT EMPNO, LASTNAME
FROM DSN910.EMPLOYEE
WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy');
```

This example returns the following row:

```
000110 LUCCHESI;
```

SOAPHTTTPC and SOAPHTTPV

The SOAPHTTTPC function returns a CLOB representation of XML data that results from a SOAP request to the web service that is specified by the first argument. The SOAPHTTPV function returns a VARCHAR representation of XML data that results from a SOAP request to the web service that is specified by the first argument.



The schema is DB2XML.

endpoint_url

An expression that returns a value of a built-in character string or graphic string data type that is not a LOB. The value specifies the URL of the web service endpoint for which DB2 is acting as a client.

soap_action

An expression that returns a value of a built-in character string or graphic string data type that is not a LOB. The value specifies a SOAP action URI reference. If it is required for the web service that is specified in *endpoint_url*, the required value is defined in the WSDL of that web service.

soap_body

An expression that returns a value of a built-in character string data type that is defined as VARCHAR(3072) or CLOB(1M). The value specifies the name of an operation with the requested namespace URI, an encoding style, and input arguments. *soap_body* can include well-formed XML content for the SOAP body. The specific operations and arguments for a web service are defined in the WSDL of the specified web service.

If the arguments can be null, the result can be null; if all of the arguments are null, the result is the null value.

Example 1: The following SQL statement retrieves information (as VARCHAR data) about a web service:

```
SELECT DB2XML.SOAPHTTPV(
    'http://www.myserver.com/services/db2sample/ivt.dadx/SOAP',
    'http://tempuri.org/db2sample/ivt.dadx',
    '<testInstallation xmlns="http://tempuri.org/db2sample/ivt.dadx" />')
FROM SYSIBM.SYSDUMMY1
```

Example 2: The following SQL statement inserts the results (as CLOB data) from a request to a web service into a table:

```
INSERT INTO EMPLOYEE(XMLCOL)
VALUES (DB2XML.SOAPHTTTPC(
    'http://www.myserver.com/services/db2sample/list.dadx/SOAP',
    'http://tempuri.org/db2sample/list.dadx',
    '<listDepartments xmlns="http://tempuri.org/db2sample/list.dadx">
    <deptNo>A00</deptNo>
</listDepartments>'))
```

SOAPHTTPNC and SOAPHTTPNV

The SOAPHTTPNC and SOAPHTTPNV functions allow you to specify a complete SOAP message as input and to return complete SOAP messages from the specified web service. The returned SOAP messages are CLOB or VARCHAR representations of the returned XML data.

```
graph LR; A[SOAPHTTPNC  
SOAPHTTPNV] -- "(endpoint_url, soap_action, soap_input)" --> B[Return Arrow]
```

The schema is DB2XML.

endpoint_url

Specifies the URL of the web service for which DB2 is acting as a client.

endpoint_url is defined as a VARCHAR(256) value. The URL is in the following format:

`proto://[user[:password]@]hostname[:port]/[path]`

Where *proto* can be http or https.

soap_action

Specifies a SOAP action URI reference. Depending on the web server,

soap_action might be required. If it is required for the web service that is specified in *endpoint_url*, the required value is defined in the WSDL of that web service.

soap_input

Specifies an XML document that contains the complete SOAP message.

soap_input can contain optional SOAP headers and must contain a SOAP body that specifies the operation name and parameters to the web service. *soap_input* should be well-formed XML that is defined as VARCHAR(32672) or CLOB(1M).

Example 1: The following SQL statement retrieves information (as VARCHAR data) about a web service:

```
SELECT DB2XML.SOAPHTTPNV(
  'http://rpc.geocoder.us/service/soap/',
  '"http://rpc.geocoder.us/Geo/Coder/US#geocode_address"',
  '<?xml version="1.0" encoding="UTF-8" ?>' ||
  '<SOAP-ENV:Envelope ' ||
  'xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" ' ||
  'xmlns:xsd="http://www.w3.org/2001/XMLSchema" ' ||
  'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' ||
  '<SOAP-ENV:Body>' ||
  '<ns0:geocode_address ' ||
  'xmlns:ns0="http://rpc.geocoder.us/Geo/Coder/US/" ' ||
  'SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">' ||
  '<address xsi:type="xsd:string">555 Bailey Avenue, San Jose,' ||
  'CA,95141</address>' ||
  '</ns0:geocode_address>' ||
  '</SOAP-ENV:Body>' ||
  '</SOAP-ENV:Envelope>')
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following SQL statement inserts the results (as CLOB data) from a request to a web service into a table:

```

|      INSERT INTO EMPLOYEE(XMLCOL)
|      VALUES (DB2XML.SOAPHTTPNC(
|          'http://www.myserver.com/services/db2sample/list.dadx/SOAP',
|          'http://tempuri.org/db2sample/list.dadx',
|          '<?xml version="1.0" encoding="UTF-8" ?>' ||
|          '<SOAP-ENV:Envelope ' ||
|          'xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" ' ||
|          'xmlns:xsd="http://www.w3.org/2001/XMLSchema" ' ||
|          'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' ||
|          '<SOAP-ENV:Body>' ||
|          '<listDepartments xmlns="http://tempuri.org/db2sample/list.dadx">
|              <deptNo>A00</deptNo>
|          </listDepartments>' ||
|          '</SOAP-ENV:Body>' ||
|          '</SOAP-ENV:Envelope>'))

```


SPACE

The SPACE function returns a character string that consists of the number of SBCS blanks that the argument specifies.

►►—SPACE(*numeric-expression*)—►◄

The schema is SYSIBM.

numeric-expression

An expression that returns the value of any built-in integer data type. The expression specifies the number of SBCS blanks for the result, and it must be between 0 and 32767.

The result of the function is a varying-length character string (VARCHAR) that contains SBCS data.

If *numeric-expression* is a constant, the length attribute of the result is the constant. Otherwise, the length attribute of the result is 4000. The actual length of the result is the value of *numeric-expression*. The actual length of the result must not be greater than the length attribute of the result.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: The following statement returns a character string that consists of 5 blanks followed by a zero-length string.

```
SELECT SPACE(5), SPACE(0)
FROM SYSIBM.SYSDUMMY1;
```

SQRT

The SQRT function returns the square root of the argument.

►►—SQRT(*numeric-expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns the value of any built-in numeric data type. If the argument is DECFLOAT, the operation is performed in DECFLOAT. Otherwise, the argument is converted to a double precision floating-point number for processing by the functions.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*). Otherwise, the result of the function is a double precision floating-point number. If the argument is a special decimal floating point value, the general rules for arithmetic operations apply. See “General Arithmetic Operation Rules for DECFLOAT” on page 186 for more information.

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

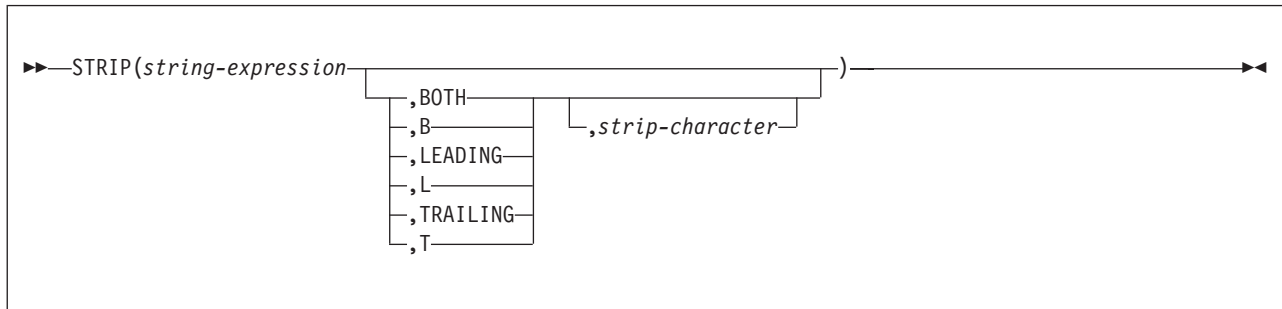
Example: Assume that host variable *SQUARE* is defined as DECIMAL(2,1) and has a value of 9.0. Find the square root of *SQUARE*.

```
SELECT SQRT(:SQUARE)
FROM SYSIBM.SYSDUMMY1;
```

This example returns a double precision floating-point number with an approximate value of 3.

STRIP

The STRIP function removes blanks or another specified character from the end, the beginning, or both ends of a string expression.



The schema is SYSIBM.

string-expression must be an expression that returns a value that is a built-in character string data type, graphic data type, or binary string data type. *string-expression* must not be a LOB.

The second argument indicates whether characters are removed from the beginning (LEADING or L), the end (TRAILING or T), or both the beginning and end (BOTH or B) of the string. If you do not specify a second argument, blanks are removed from both the beginning and end of the string.

strip-character must be an SBCS, or DBCS single-character (2 bytes) constant or a single-byte binary constant that is to be removed. If the data type is not appropriate or the value contains more than one character, an error is returned.

If *strip-character* is not specified, the default strip character is:

- SBCS space character if *string-expression* is a character string
- DBCS space character if *string-expression* is a graphic string
- Hexadecimal zero (BX'00') if *string-expression* is a binary string

The data type of the result depends on the data type of *string-expression*:

- If *string-expression* is a character string data type, the result is VARCHAR.
- If *string-expression* is a graphic string data type, the result is VARGRAPHIC.
- If *string-expression* is a binary string data type, the result is VARBINARY.

The result of the function has the same maximum length as the length attribute of *string-expression*. The actual length of the result is the length of the expression minus the number of characters removed. If all of the characters are removed, the result is an empty, varying-length string.

If *string-expression* and *strip-character* have different CCSID sets, *strip-character* is converted to the CCSID of *string-expression*. The CCSID of the result is the same as that of the 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.

Example 1: Assume that host variable *HELLO* is defined as CHAR(9) and has a value of ' Hello':

```
STRIP(:HELLO)
```

This example results in 'Hello'. If there had been any ending blanks, they would have been removed as well because the second argument was not specified.

No beginning blanks are removed from the following example:

```
STRIP(:HELLO,TRAILING)
```

This results in ' Hello'.

Example 2: Assume that host variable *BALANCE* is defined as CHAR(9) and has a value of '000345.50':

```
STRIP(:BALANCE,L,'0')
```

This example results in '345.50'.

Example 3: The following function returns a VARBINARY string with a length attribute 6, actual length 3 and a value BX'C1C1C1'

```
SELECT STRIP(BINARY(X'C1C1C1D2D2D2'), B, BX'D2')  
FROM SYSIBM.SYSDUMMY1;
```

SUBSTR

The SUBSTR function returns a substring of a string.

►—SUBSTR(*string-expression*,*start*—length)—►

The schema is SYSIBM.

string-expression

An expression that specifies the string from which the result is derived. The string must be a character, graphic, or binary string. If *string-expression* is a character string, the result of the function is a character string. If it is a graphic string, the result of the function is a graphic string. If it is a binary string, the result of the function is a binary string.

A substring of *string-expression* is zero or more contiguous characters of *string-expression*. If *string-expression* is a graphic string, a character is a DBCS character. If *string-expression* is a character string or a binary string, a character is a byte. The SUBSTR function accepts mixed data strings. However, because SUBSTR operates on a strict byte-count basis, the result will not necessarily be a properly formed mixed data string.

start

An expression that specifies the position within *string-expression* to be the first character of the result. The value of the large integer must be between 1 and the length attribute of *string-expression*. (The length attribute of a varying-length string is its maximum length.) A value of 1 indicates that the first character of the substring is the first character of *string-expression*.

length

An expression that specifies the length of the resulting substring. If specified, *length* must be an expression that returns a value that is a built-in large integer data type. The value must be greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *string-expression* - *start* + 1. The specified length must not, however, be the large integer constant 0.

If *length* is explicitly specified, *string-expression* is effectively padded on the right with the necessary number of characters so that the specified substring of *string-expression* always exists. Hexadecimal zeros are used as the padding character when *string-expression* is binary data. Otherwise, a blank is used as the padding character.

If *string-expression* is a fixed-length string, omission of *length* is an implicit specification of $\text{LENGTH}(\text{string-expression}) - \text{start} + 1$, which is the number of characters (or bytes) from the character (or byte) specified by *start* to the last character (or byte) of *string-expression*. If *string-expression* is a varying-length string, omission of *length* is an implicit specification of the greater of zero or $\text{LENGTH}(\text{string-expression}) - \text{start} + 1$. If the resulting length is zero, the result is an empty string.

If *length* is explicitly specified by a large integer constant that is 255 or less, and *string-expression* is not a LOB, the result is a fixed-length string with a length attribute of *length*. If *length* is not explicitly specified, but *string-expression* is a fixed-length string and *start* is an integer constant, the

result is a fixed-length string with a length attribute equal to $\text{LENGTH}(\text{string-expression}) - \text{start} + 1$. In all other cases, the result is a varying-length string. If *length* is explicitly specified by a large integer constant, the length attribute of the result is *length*; otherwise, the length attribute of the result is the same as the length attribute of *string-expression*.

If any argument of SUBSTR can be null, the result can be null. If any argument is null, the result is the null value. The CCSID of the result is the CCSID of *string-expression*.

Example 1: FIRSTNME is a VARCHAR(12) column in sample table DSN8910.EMP. When FIRSTNME has the value 'MAUDE':

Function:	Returns:
SUBSTR(FIRSTNME,2,3)	-- 'AUD'
SUBSTR(FIRSTNME,2)	-- 'AUDE'
SUBSTR(FIRSTNME,2,6)	-- 'AUDE' followed by two blanks
SUBSTR(FIRSTNME,6)	-- a zero-length string
SUBSTR(FIRSTNME,6,4)	-- four blanks

Example 2: Sample table DSN8910.PROJ contains column PROJNAME, which is defined as VARCHAR(24). Select all rows from that table for which the string in PROJNAME begins with 'W L PROGRAM'.

```
SELECT * FROM DSN8910.PROJ
WHERE SUBSTR(PROJNAME,1,12) = 'W L PROGRAM ';
```

Assume that the table has only the rows that were supplied by DB2. Then the predicate is true for just one row, for which PROJNAME has the value 'W L PROGRAM DESIGN'. The predicate is not true for the row in which PROJNAME has the value 'W L PROGRAMMING' because, in the predicate's string constant, 'PROGRAM' is followed by a blank.

Example 3: Assume that a LOB locator named *my_loc* represents a LOB value that has a length of 1 gigabyte. Assign the first 50 bytes of the LOB value to host variable *PORTION*.

```
SET :PORTION = SUBSTR(:my_loc,1,50);
```

Example 4: Assume that host variable *RESUME* has a CLOB data type and holds an employee's resume. This example shows some of the statements that find the section of department information in the resume and assign it to host variable *DeptBuf*. First, the POSSTR function is used to find the beginning and ending location of the department information. Within the resume, the department information starts with the string 'Department Information Section' and ends immediately before the string 'Education Section'. Then, using these beginning and ending positions, the SUBSTR function assigns the information to the host variable.

```
SET :DInfoBegPos = POSSTR(:RESUME, 'Department Information Section');
SET :DInfoEnPos = POSSTR(:RESUME, 'Education Section');
SET :DeptBuf = SUBSTR(:RESUME, :DInfoBegPos, :DInfoEnPos - :DInfoBegPos);
```

SUBSTRING

The SUBSTRING function returns a substring of a string.

Character:

►► SUBSTRING (*character-expression* , *start* [, *length*] , CODEUNITS16
CODEUNITS32
OCTETS) ►

Graphic:

►►SUBSTRING(*graphic-expression*, *start* CODEUNITS16, CODEUNITS32 *length*)►►

Binary:

►► SUBSTRING(*binary-expression*, *start* . *length*)

The schema is SYSIBM.

Character

character-expression

An expression that specifies the string from which the result is derived. The string must be a built-in character string. The result of the function is a character string.

A substring of *character-expression* is zero or more contiguous units of *character-expression*. If CODEUNITS32 is specified, a unit is a Unicode UTF-32 character. If CODEUNITS16 is specified, a unit is a Unicode UTF-16 character. If OCTETS is specified, a unit is a byte.

start

An expression that specifies the position within the *character-expression* that is to be the first string unit of the result. *start* is expressed in the specified string unit, and must return a large integer value. The value of *start* can be positive, negative, or zero. A value of 1 indicates that the first string unit of the result is the first string unit of *character-expression*.

length

An expression that specifies the maximum length of the resulting substring.

If *character-expression* is a fixed-length string, omission of *length* is an implicit specification of `CHARACTER_LENGTH(character-expression) - start + 1`, which is the number of string units (CODEUNITS16, CODEUNITS32, or OCTETS) from *start* to the last position of *character-expression*.

If *character-expression* is a varying length string, omission of *length* is an implicit specification of zero or `CHARACTER_LENGTH(character-expression) - start + 1`, whichever is greater. If the resulting length is zero, the result is an empty string.

If specified, *length* must be an expression that returns a value that is a built-in large integer data type.

The value must be greater than or equal to 0. If a value greater than *n* is specified, where *n* is the length attribute of *character-expression* - *start* + 1, then *n* is used as the length of the resulting substring. The value is expressed in the units that are explicitly specified.

A rigorous description of the actual length and result: In this description, the term “character” means the “unit specified by string units”.

Let *C* be the value of the first argument, let *LC* be the length in characters of *C*, and let *S* be the value of the *start*.

- If *length* is specified, let *L* be the value of *length* and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.
- If either *C*, *S*, or *L* is the null value, the result of the function is the null value.
- If *E* is less than *S*, an exception condition is raised: data exception — substring error.
- Otherwise:
 - If *S* is greater than *LC* or if *E* is less than 1 (one), the result of the function is a zero-length string.
 - Otherwise:
 - Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*–*S1*.
 - The result of the function is a character string that contains the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit that is used to express *start* and *length*. If *character-expression* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified.

CODEUNITS16

Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *start* and *length* are expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

Graphic

graphic-expression

An expression that specifies the string from which the result is derived. The string must be a built-in graphic string. The result of the function is a graphic string.

A substring of *graphic-expression* is zero or more contiguous units of *graphic-expression*. If CODEUNITS32 is specified, a unit is a Unicode UTF-32 character. If CODEUNITS16 is specified, a unit is a Unicode UTF-16 character.

start

An expression that specifies the position within the *graphic-expression* that is to be the first string unit of the result. *start* is expressed in the specified string unit, and must return a large integer value. The value of *start* can be positive, negative, or zero. A value of 1 indicates that the first string unit of the result is the first string unit of *graphic-expression*.

length

An expression that specifies the maximum length of the resulting substring.

If *graphic-expression* is a fixed-length string, omission of *length* is an implicit specification of $\text{CHARACTER_LENGTH}(\text{graphic-expression}) - \text{start} + 1$, which is the number of units (CODEUNITS16, CODEUNITS32) either explicitly or implicitly specified, from the start position to the last position of *graphic-expression*. If *graphic-expression* is a varying length string, omission of *length* is an implicit specification of zero or $\text{CHARACTER_LENGTH}(\text{graphic-expression}) - \text{start} + 1$, which is the number of units (CODEUNITS16, CODEUNITS32) either explicitly or implicitly specified, whichever is greater. If the resulting length is zero, the result is an empty string.

If specified, *length* must be an expression that returns a value that is a built-in large integer data type.

The value must be greater than or equal to 0. If a value greater than *n* is specified, where *n* is the length attribute of *graphic-expression* - *start* + 1, then *n* is used as the length of the resulting substring. The value is expressed in the units that are explicitly specified.

A rigorous description of the actual length and result: In this description, the term “character” means the “unit specified by string units”.

Let *C* be the value of the first argument, let *LC* be the length in characters of *C*, and let *S* be the value of the *start*.

- If *length* is specified, let *L* be the value of *length* and let *E* be *S*+*L*. Otherwise, let *E* be the larger of *LC* + 1 and *S*.
- If either *C*, *S*, or *L* is the null value, the result of the function is the null value.
- If *E* is less than *S*, an exception condition is raised: data exception — substring error.
- Otherwise:
 - If *S* is greater than *LC* or if *E* is less than 1 (one), the result of the function is a zero-length string.
 - Otherwise:
 - Let *S1* be the larger of *S* and 1 (one). Let *E1* be the smaller of *E* and *LC*+1. Let *L1* be *E1*−*S1*.
 - The result of the function is a character string that contains the *L1* characters of *C* starting at character number *S1* in the same order that the characters appear in *C*.

CODEUNITS16 or CODEUNITS32

Specifies the string unit that is used to express *start* and *length*.

CODEUNITS16

Specifies that *start* and *length* are expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *start* and *length* are expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

Binary

binary-expression

An expression that specifies the string from which the result is derived. The string must be a built-in binary string. The result of the function is a binary string.

A substring of *binary-expression* is zero or more contiguous units of *binary-expression*.

start

An expression that specifies the position within *binary-expression* to be the first character of the result. It must be a binary large integer. *start* can be negative or zero. (The length attribute of a varying-length string is its maximum length.) A value of 1 indicates that the first string unit of the substring is the first string unit of *binary-expression*.

length

An expression that specifies the length of the resulting substring.

If *binary-expression* is a fixed-length string, omission of *length* is an implicit specification of $\text{CHARACTER_LENGTH}(\text{binary-expression}) - \text{start} + 1$, which is the number of units either explicitly or implicitly specified, from the start position to the last position of *binary-expression*. If *binary-expression* is a varying length string, omission of *length* is an implicit specification of zero or $\text{CHARACTER_LENGTH}(\text{binary-expression}) - \text{start} + 1$, which is the number of units either explicitly or implicitly specified, whichever is greater. If the resulting length is zero, the result is an empty string.

If specified, *length* must be a value that is a built-in large integer data type. The value must be greater than or equal to 0 and less than or equal to n , where n is the length attribute of *binary-expression* - *start* + 1. The specified length must not, however, be the large integer constant 0.

A rigorous description of the actual length and result: In this description, the term “character” means the “unit specified by string units”.

Let C be the value of the first argument, let LC be the length in characters of C , and let S be the value of the *start*.

- If *length* is specified, let L be the value of *length* and let E be $S+L$. Otherwise, let E be the larger of $LC + 1$ and S .
- If either C , S , or L is the null value, the result of the function is the null value.
- If E is less than S , an exception condition is raised: data exception — substring error.
- Otherwise:
 - If S is greater than LC or if E is less than 1 (one), the result of the function is a zero-length string.
 - Otherwise:
 - Let $S1$ be the larger of S and 1 (one). Let $E1$ be the smaller of E and $LC+1$. Let $L1$ be $E1-S1$.
 - The result of the function is a character string that contains the $L1$ characters of C starting at character number $S1$ in the same order that the characters appear in C .

The data type of the result depends on the data type of the first argument, as shown in the following table.

Table 58. Data type of the result of SUBSTRING

Data type of the first argument	Data type of the result
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
	If <i>character-expression</i> is mixed data, the result is mixed data. Otherwise, the result is SBCS data.
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result is equal to the length attribute of the first argument. If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If any argument of SUBSTRING can be null, the result can be null. If any argument is null, the result is the null value.

If the first argument is character or graphic data, the CCSID of the result is the same as that of the first argument.

Example 1: FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has the value 'Jürgen':

Function:	Returns:
SUBSTRING(FIRSTNAME,1,2,CODEUNITS32)	'Jü' -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,CODEUNITS16)	'Jü' -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,OCTETS)	'J ' -- x'4A20' (a truncated string)
SUBSTRING(FIRSTNAME,8,CODEUNITS16)	-- a zero-length string
SUBSTRING(FIRSTNAME,8,4,OCTETS)	-- a zero-length string

Example 2: C1 is a VARCHAR(12) column in table T1. One of its values is the string 'ABCDEFGF'. When C1 has the value 'ABCDEFGF':

Function:	Returns:
SUBSTRING(C1,-2,2,OCTETS)	-- a zero-length string
SUBSTRING(C1,-2,4,OCTETS)	'A'
SUBSTRING(C1,-2,OCTETS)	'ABCDEFGF'
SUBSTRING(C1,0,1,OCTETS)	-- a zero-length string

TAN

The TAN function returns the tangent of the argument, where the argument is an angle, expressed in radians.

►►—TAN(*numeric-expression*)—◄◄

The schema is SYSIBM.

The TAN and ATAN functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type that is not DECFLOAT. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *TANGENT* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 14.10 .

```
SELECT TAN(:TANGENT)
FROM SYSIBM.SYSDUMMY1;
```

TANH

The TANH function returns the hyperbolic tangent of the argument, where the argument is an angle, expressed in radians.

►►—TANH(*numeric-expression*)—◄◄

The schema is SYSIBM.

The TANH and ATANH functions are inverse operations.

| The argument must be an expression that returns the value of any built-in numeric
| data type that is not DECFLOAT. If the argument is not a double precision
| floating-point number, it is converted to one for processing by the function.

The result of the function is a double precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable *HTANGENT* is DECIMAL(2,1) with a value of 1.5. The following statement returns a double precision floating-point number with an approximate value of 0.90.

```
SELECT TANH(:HTANGENT)
FROM SYSIBM.SYSDUMMY1;
```

TIME

The TIME function returns a time that is derived from a value.

►►—TIME(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String representations of datetime values” on page 89.

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:

If the argument is a time

the result is that time.

If the argument is a timestamp

the result is the time part of the timestamp.

If the argument is a string

the result is the time or time part of the timestamp represented by the string. If the CCSID of the string is not the same as the corresponding default CCSID at the server, the string is first converted to that CCSID.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID.

Example: Assume that a table named CLASSES contains one row for each scheduled class. Assume also that the class starting times are in the TIME column named STARTTM. Using these assumptions, select those rows in CLASSES that represent classes that start at 1:30 P.M.

```
SELECT *
FROM CLASSES
WHERE TIME(STARTTM) = '13:30:00';
```

TIMESTAMP

The `TIMESTAMP` function returns a timestamp that is derived from its argument or arguments.

```
►►—TIMESTAMP(expression-1 [ ,expression-2 ] )—◄◄
```

The schema is `SYSIBM`.

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

- **If only one argument is specified:**

The argument must be an expression that returns a value of one of the following built-in data types: a timestamp, a character string, or a graphic string. If *expression-1* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be one of the following:

- A valid string representation of a timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see “String representations of datetime values” on page 89.
- A character string or graphic string with an actual length of 8 that is assumed to be a System z[®] Store Clock value.
- A character string with an actual length of 13 that is assumed to be a result from the `GENERATE_UNIQUE` function.
- A character string or graphic string with an actual length of 14 that represents a valid date and time in the form *yyyyxxddhhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.

- **If both arguments are specified:**

The first argument must be an expression that returns a value of one of the following built-in data types: a date, a character string, or a graphic string. The second argument must be an expression that returns a value of one of the following built-in data types: a time, a character string, or a graphic string. A character or graphic string must be a valid string representation of a time.

If *expression-1* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date with an actual length that is not greater than 255 bytes. If *expression-2* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a time with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and times, see “String representations of datetime values” on page 89.

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

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. The timestamp represented by a string of length 14 has a microsecond part of zero. The interpretation of a string as a Store Clock value will yield a timestamp with a year between 1900 to 2042.

If an argument is a string with a CCSID that is not the same as the corresponding default CCSID at the server, the string is first converted to that CCSID.

The result CCSID is the appropriate CCSID of the argument encoding scheme and the result subtype is the appropriate subtype of the CCSID. If both arguments are specified and their encoding schemes are different, the result CCSID is the appropriate CCSID of the application encoding scheme.

Example 1: Assume that table TABLEX contains a DATE column named DATECOL and a TIME column named TIMECOL. For some row in the table, assume that DATECOL represents 25 December 2008 and TIMECOL represents 17 hours, 12 minutes, and 30 seconds after midnight. The following function returns the value '2008-12-25-17.12.30.000000'.

```
TIMESTAMP(DATECOL, TIMECOL)
```


TIMESTAMPADD

The `TIMESTAMPADD` function returns the result of adding the specified number of the designated interval to the timestamp value.

►►—`TIMESTAMPADD(interval,number,expression)`—◄◄

The schema is `SYSIBM`.

interval

An expression that returns a value of a built-in `SMALLINT` or `INTEGER` data type. The following values are valid values for *interval*:

Table 59. Valid values for intervals

Valid values for <i>interval</i>	equivalent intervals
1	Microseconds
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

number

An expression that returns a value of a built-in `SMALLINT` or `INTEGER` data type.

timestamp

An expression that returns a value of a built-in timestamp data type.

The result of the function is a timestamp.

The result is determined using the normal rules for datetime arithmetic. See “Datetime arithmetic in SQL” on page 193. When the interval to add is expressed as weeks, the result is calculated as if *number* x 7 days had been specified. When the interval to add is expressed as quarters, the result is calculated as if *number* x 3 months had been specified.

The result must be a valid timestamp value. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Example 1: The following example will add 40 years to the specified timestamp. An interval of 256 designates years, while 40 specifies the number of intervals to add. The following statement returns the value '2005-07-27-15.30.00.000000'.

```
SELECT TIMESTAMPADD(256,40,TIMESTAMP('1965-07-27-15.30.00'))
FROM SYSIBM.SYSDUMMY1;
```

| *Example 2:* The following example will add 18 months to the specified timestamp.
| An interval of 64 designates months, while 18 specifies the number of intervals to
| add. The following statement returns the value '2008-07-20-08.08.00.000000'.

```
|      SELECT TIMESTAMPADD(64,18,TIMESTAMP('2007-01-20-08.08.00'))  
|      FROM SYSIBM.SYSDUMMY1;
```

| *Example 3:* The following example will subtract 16 quarters (4 years) from the
| specified timestamp. An interval of 128 designates quarters, while -16 specifies the
| number of intervals to add (the '-' adds a negative amount). The following
| statement returns the value '2003-09-28-05.30.00.000000'.

```
|      SELECT TIMESTAMPADD(128,-16,TIMESTAMP('2007-09-28-05.30.00'))  
|      FROM SYSIBM.SYSDUMMY1;
```

| *Example 4:* The following example will add 18 weeks to the specified timestamp.
| An interval of 32 designates weeks, while 18 specifies the number of intervals to
| add. The following statement returns the value '2007-05-27-08.08.00.000000'.

```
|      SELECT TIMESTAMPADD(32,18,TIMESTAMP('2007-01-20-08.08.00'))  
|      FROM SYSIBM.SYSDUMMY1;
```

TIMESTAMP_FORMAT

The `TIMESTAMP_FORMAT` function returns a timestamp that is based on interpreting the input string by using the specified format.

►►—`TIMESTAMP_FORMAT`—(*—string-expression—*,*—format-string—*)—◄◄

The schema is `SYSIBM`.

string-expression

An expression that returns a value of any built-in character or graphic string data type, other than a `CLOB` or `DBCLOB`, with a length attribute that is not greater than 255 bytes. The *string-expression* must contain the components of a timestamp that correspond to the format that is specified in *format-string*, except for hour, minute, second, or fractional seconds.

format-string

A character string constant with a length that is not greater than 255 bytes. The value is a template for how *string-expression* is interpreted and then converted to a timestamp value.

A valid *format-string* must contain at least one format element, must not contain multiple specifications for any component of a timestamp, and can contain any combination of the format elements, unless otherwise noted in the following table. For example, *format-string* cannot contain both `YY` and `YYYY`, because both are used to interpret the year component of a *string-expression*. Two format elements can be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- forward slash (/)
- comma (,)
- apostrophe (')
- semicolon (;)
- colon (:)
- blank ()

Separator characters can also be specified at the start and end of *format-string*. These separator characters can be used in any combination in the format string, for example `'YYYY/MM-DD HH:MM.SS'`. Separator character that is specified in a *string-expression* are used to separate components and are not required to match the separator character that is specified in the *format-string*.

Table 60. Format elements for the `TIMESTAMP_FORMAT` function

Format element	Related component of a timestamp	Description
AM or PM ¹	hour	Meridian indicator (morning or evening) without periods. This format element uses the exact strings "AM" or "PM".
DD	day	Day of the month (0-31).

Table 60. Format elements for the `TIMESTAMP_FORMAT` function (continued)

Format element	Related component of a timestamp	Description
FF or FF n	fractional seconds	Fractional seconds (0-999999). The number n is used to specify the number of digits that is expected in the <i>string-expression</i> . Valid values for n are 1-6 with no leading zeros. Specifying FF is equivalent to specifying FF6. When the number of digits for the fractional seconds is less than what is specified by the format element, zero digits are padded onto the right of the number of specified digits.
HH	hour	HH behaves the same as HH12.
HH12	hour	Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.
HH24	hour	Hour of the day (00-24) in 24-hour format.
MI	minute	Minute (00-59).
MM	month	Month (01-12).
MONTH, Month, or month ^{1, 2}	month	Name of the month in English.
MON, Mon, or mon ^{1, 2}	month	Abbreviated name of the month in English.
NNNNNN	microseconds	Microseconds (000000-999999).
RR	year	Last two digits of the adjusted year (00-99).
RRRR	year	Four digit adjusted year (0000-9999).
SS	seconds	Seconds (00-59).
Y	year	Last digit of the year (0-9). First three digits of the current year are used to determine the full 4-digit year.
YY	year	Last two digits of the year (00-99). First two digits of the current year are used to determine the full 4-digit year.

Table 60. Format elements for the `TIMESTAMP_FORMAT` function (continued)

Format element	Related component of a timestamp	Description
YYY	year	Last three digits of the year (000-999). First digit of the current year is used to determine the full 4-digit year.
YYYY	year	4-digit year (0000-9999).

Notes:

1. This format element is case sensitive.
2. Only these exact spellings and case combinations can be used. If this format element is specified in an invalid case combination an error is returned.

The RR and RRRR format elements can be used to change how a specification for a year is to be interpreted by adjusting the value to produce a 2-digit or a 4-digit value depending on the leftmost two digits of the current year according to the following table:

Table 61. Correspondence of adjusted year value and timestamp component

Digits of the current year	Two-digit year in <i>string-expression</i>	First two digits of the year component of timestamp
00-50	00-49	First two digits of the current year
51-99	00-49	First two digits of the current year + 1
00-50	50-99	First two digits of the current year -1
51-99	50-99	First two digits of the current year

For example, if the current year is 2007, '86' with format 'RR' means 1986, but if the current year is 2052, it means 2086.

The following defaults are used when a format-string does not include a format element for one of the following components of a timestamp:

Timestamp component	Default
year	current year, as 4 digits
month	current month, as 2 digits
day	01 (first day of the month)
hour	00
minute	00
second	00
fractional seconds	a number of zeros to match the timestamp precision of the result

If *string-expression* does not include a value that corresponds to an hour, minute, second, or fractional seconds format element that is specified in the *format-string*, the same defaults are used.

Leading zeros can be specified for any component of the timestamp value (that is, month, day, hour, minutes, seconds) that does not have the maximum number of significant digits for the corresponding format element in the *format-string*.

A substring of the *string-expression* that represents a component of a timestamp (such as year, month, day, hour, minutes, seconds) can include fewer than the maximum number of digits for that component of the timestamp that is indicated by the corresponding format element. Any missing digits default to zero. For example, with a format-string of 'YYYY-MM-DD HH24:MI:SS', an input value of '999-3-9 5:7:2' produces the same result as '0999-03-09 05:07:02'.

The result of the function is a timestamp.

If either of the first two arguments can be null, the result can be null; if either of the first two arguments is null, the result is the null value.

The result CCSID is the appropriate CCSID of the encoding scheme of the first argument and the result subtype is the appropriate subtype of the CCSID.

Notes

Determinism:

TIMESTAMP_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT_TIMESTAMP.

- *format-string* is a constant and includes format elements that are locale sensitive
- *format-string* is a constant and does not include a format element that fully defines the year (that is, YYYY). In this case the current year is used.
- *format-string* is a constant and does not include a format element that fully defines the month (for example, MM, MONTH, or MON). In this case the current month is used.

These invocations, which depend on the value of a special register, cannot be used wherever special registers cannot be used.

Using the 'D', 'Y', and 'y' format elements:

DB2 for z/OS does not support the 'DY', 'dy', and 'Dy' format elements that are supported by other platforms. If 'DY' or 'Dy' is specified in the format string, it is interpreted as the 'D' format element followed by the 'Y' or 'y' format element. This behavior might change in a future release. To ensure that a 'D' followed by 'Y' or 'y' is interpreted as two separate format elements, include a separator character after the 'D' format element.

Syntax alternatives:

TO_DATE can be specified as a synonym for TIMESTAMP_FORMAT.

Example 1:

Insert a row into the IN_TRAY table with a receiving timestamp that is equal to one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59).

```
INSERT INTO IN_TRAY (RECEIVED)
VALUES (TIMESTAMP_FORMAT('1999-12-31 23:59:59', 'YYYY-MM-DD HH24:MI:SS'))
```

Example 2:

An application receives strings of date information into a variable called INDATEVAR. This value is not strictly formatted and might include two or

four digits for years, and one or two digits for months and days. Date components might be separated with minus sign (-) or forward-slash (/) characters and are expected to be in day, month, and year order. Time information consists of hours (in 24-hour format) and minutes, and is usually separated by a colon. Sample values include '15/12/98 13:48' and '9-3-2004 8:02'. Insert such values into the IN_TRAY table.

```
INSERT INTO IN_TRAY (RECEIVED)
VALUES (TIMESTAMP_FORMAT(:INDATEVAR, 'DD/MM/RRRR HH24:MI'))
```

The use of 'RRRR' in the format allows for 2-digit and 4-digit year values and assigns the missing first two digits based on the current year. If 'YYYY' is used, input values with a 2-digit year will have leading zeros. The forward-slash separator also allows the minus sign character. Assuming a current year of 2007, resulting timestamp values from the sample values are as follows:

```
'15/12/98 13:48' --> 1998-12-15-13.48.00.000000
'9-3-2004 8:02'  --> 2004-03-09-08.02.00.000000
```

TIMESTAMP_ISO

The `TIMESTAMP_ISO` function returns a timestamp value that is based on a date, a time, or a timestamp argument.

►—`TIMESTAMP_ISO(expression)`—►

The schema is SYSIBM.

If the argument is a date, `TIMESTAMP_ISO` inserts a value of zero for the time and the partial seconds parts of the timestamp. If the argument is a time, `TIMESTAMP_ISO` inserts the value of `CURRENT DATE` for the date part of the timestamp and a value of zero for the partial seconds part of the timestamp.

expression

An expression that returns a value of one of the following built-in data types:

- a timestamp
- a date
- a time
- a character string
- or a graphic string

If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB and its value must be a valid string representation of a date, a time, or a timestamp. For the valid formats of string representations of dates, times, and timestamps, see “String representations of datetime values” on page 89.

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

Recommendation: Use the `CAST` specification for maximum portability. For more information, see “`CAST` specification” on page 202.

Example: Assume the following date value '1965-07-27'. The following statement returns the value '1965-07-27-00.00.00.000000'.

```
SELECT TIMESTAMP_ISO( DATE( '1965-07-27' ) )  
FROM SYSIBM.SYSDUMMY1
```


TIMESTAMPDIFF

The `TIMESTAMPDIFF` function returns an estimated number of intervals of the type that is defined by the first argument, based on the difference between two timestamps.

►►—`TIMESTAMPDIFF(numeric-expression,string-expression)`—►►

The schema is `SYSIBM`.

numeric-expression

An expression that returns a value that is a built-in `SMALLINT` or `INTEGER` data type. The value specifies the interval that is used to determine the difference between two timestamps. The following table lists the valid values for *numeric-expression*:

Table 62. Valid values for *numeric-expression* and equivalent intervals that are used to determine the difference between two timestamps

Valid values for <i>numeric-expression</i>	equivalent intervals
1	Microseconds
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

string-expression

string-expression must be the equivalent of subtracting two timestamps and converting the result to a string of length 22. The argument must be an expression that returns a value of a built-in character string or a graphic string data type that is not a LOB. If the supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The following table lists the valid input value ranges for *string-expression*:

Table 63. *TIMESTAMPDIFF* String Elements

String elements	Valid values	Character position from the decimal point (negative is left)
Years	1-9998 or blank	-14 to -11
Months	0-11 or blank	-10 to -9
Days	0-30 or blank	-8 to -7
Hours	0-24 or blank	-6 to -5
Minutes	0-59 or blank	-4 to -3

Table 63. *TIMESTAMPDIFF* String Elements (continued)

String elements	Valid values	Character position from the decimal point (negative is left)
Seconds	0-59	-2 to -1
Decimal separator	period	0
Microsecond	000000-999999	1 to 6

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

The returned value is determined for each interval as indicated by the following table:

Table 64. *TIMESTAMPDIFF* Computations

Result interval	Computation using duration elements
Years	years
Quarters	integer value of $(\text{months} + (\text{years} * 12)) / 3$
Months	$\text{months} + (\text{years} * 12)$
Weeks	integer value of $((\text{days} + \text{months} * 30) / 7) + (\text{years} * 52)$
Days	$\text{days} + (\text{months} * 30) + (\text{years} * 365) * 24$
Minutes (the absolute value of the duration must not exceed 40850913020759.999999)	$\text{minutes} + (\text{hours} + ((\text{days} + (\text{months} * 30) + (\text{years} * 365) * 24)) * 60$
Seconds (the absolute value of the duration must be less than 680105031408.000000)	$\text{seconds} + (\text{minutes} + (\text{hours} + ((\text{days} + (\text{months} * 30) + (\text{years} * 365) * 24)) * 60) * 60$
Microseconds (the absolute value of the duration must be less than 3547.483648)	$\text{microseconds} + (\text{seconds} + (\text{minutes} * 60)) * 1000000$

The following assumptions are used when converting the information in the second argument, which is a timestamp duration, to the interval type that is specified in the first argument, and can be used in estimating the difference between the timestamps:

- One year has 365 days
- One year has 52 weeks
- One year has 12 months
- One month has 30 days
- One week has 7 days
- One day has 24 hours
- One hour has 60 minutes
- One minute has 60 seconds

The use of these assumptions imply that some result values are an estimate of the interval. Consider the following examples:

- Difference of 1 month where the month has less than 30 days.

```
TIMESTAMPDIFF(16, CHAR(TIMESTAMP('1997-03-01-00.00.00')
- TIMESTAMP('1997-02-01-00.00.00')))
```

The result of the timestamp arithmetic is a duration of 00000100000000.000000, or 1 month. When the *TIMESTAMPDIFF* function is invoked with 16 for the interval argument (days), the assumption of 30 days in a month is applied and the result is 30.

- Difference of 1 day less than 1 month where the month has less than 30 days.

```
TIMESTAMPDIFF(16, CHAR(TIMESTAMP('1997-03-01-00.00.00'))
- TIMESTAMP('1997-02-02-00.00.00')))
```

The result of the timestamp arithmetic is a duration of 00000027000000.000000, or 27 days. When the `TIMESTAMPDIFF` function is invoked with 16 for the interval argument (days), the result is 27.

- Difference of 1 day less than 1 month where the month has 31 days.

```
TIMESTAMPDIFF(64, CHAR(TIMESTAMP('1997-09-01-00.00.00'))
- TIMESTAMP('1997-08-02-00.00.00')))
```

The result of the timestamp arithmetic is a duration of 00000030000000.000000, or 30 days. When the `TIMESTAMPDIFF` function is invoked with 64 for the interval argument (months), the result is 0. The days portion of the duration is 30, but it is ignored because the interval specified months.

Example: The following statement estimates the age of employees in months and returns that value as `AGE_IN_MONTHS`:

```
SELECT
  TIMESTAMPDIFF(64, CAST(CURRENT_TIMESTAMP-CAST(BIRTHDATE AS TIMESTAMP)
                        AS CHAR(22)))
  AS AGE_IN_MONTHS
FROM EMPLOYEE;
```

TO_CHAR

The TO_CHAR function returns a character string representation of a timestamp value that has been formatted using a specified character template.

►►—TO_CHAR—(—*string-expression*—,—*format-string*—)————►◄

The schema is SYSIBM.

The TO_CHAR scalar function is a synonym for the VARCHAR_FORMAT scalar function.

TO_DATE

The TO_DATE function returns a timestamp value that is based on the interpretation of the input string using the specified format.

►► TO_DATE (—*string-expression*—, —*format-string*—) ◀◀

The schema is SYSIBM.

The TO_DATE scalar function is a synonym for the TIMESTAMP_FORMAT scalar function.

TOTALORDER

The TOTALORDER function returns an ordering for DECFLOAT values. The TOTALORDER function returns a small integer value that indicates how *expression1* compares with *expression2*.

►—TOTALORDER(*expression1*,*expression2*)—►

The schema is SYSIBM.

expression1

An expression that returns a built-in DECFLOAT value.

expression2

An expression that returns a built-in DECFLOAT value.

Numeric comparison is exact, and the result is determined for finite operands as if range and precision are unlimited. An overflow or underflow conditions cannot occur.

If one value is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

TOTALORDER determines ordering based on the total order predicate rules of IEEE 754R, with the following result:

- -1 if the first argument is lower in order compared to the second.
- 0 if both arguments have the same order.
- 1 if the first argument is higher in order compared to the second.

The ordering of the special values and finite numbers is as follows:

-NAN<-SNAN<-INFINITY<-0.10<-0.100<-0<0<0.100<0.10<INFINITY<SNAN<NAN

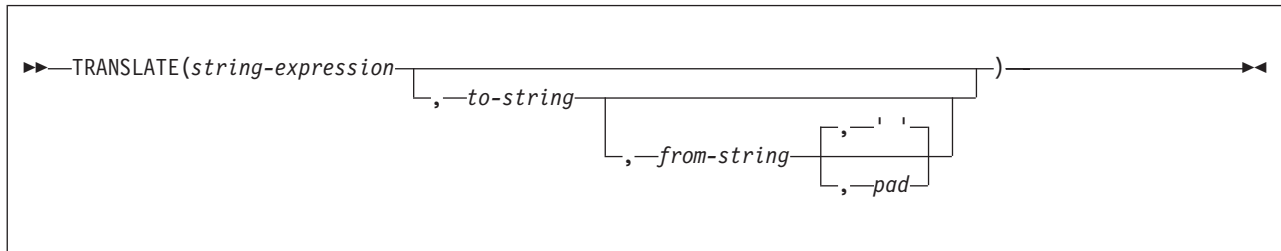
The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples: The following examples show the use of the TOTALORDER function to compare decimal floating point values:

TOTALORDER(-INFINITY, -INFINITY)	= 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.0))	= 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.00))	= -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-0.5))	= -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(0.5))	= -1
TOTALORDER(DECFLOAT(-1.0), INFINITY)	= -1
TOTALORDER(DECFLOAT(-1.0), SNAN)	= -1
TOTALORDER(DECFLOAT(-1.0), NAN)	= -1
TOTALORDER(NAN, DECFLOAT(-1.0))	= 1
TOTALORDER(-NAN, -NAN)	= 0
TOTALORDER(-SNAN, -SNAN)	= 0
TOTALORDER(NAN, NAN)	= 0
TOTALORDER(SNAN, SNAN)	= 0

TRANSLATE

The TRANSLATE function returns a value in which one or more characters of the first argument might have been converted to other characters.



The schema is SYSIBM.

string-expression

An expression that specifies the string to be converted. *string-expression* must return a value that is a built-in character or graphic string data type that is not a LOB.

to-string

An expression that specifies the characters to which certain characters in *string-expression* are to be converted. This string is sometimes called the *output translation table*. *to-string* must return a value that is a built-in character or graphic string data type that is not a LOB.

If the length of *to-string* is less than the length of *from-string*, *to-string* is padded to the length of *from-string* with the *pad* or a blank. If the length of *to-string* is greater than *from-string*, the extra characters in *to-string* are ignored without warning.

from-string

An expression that specifies the characters that if found in *string-expression* are to be converted. This string is sometimes called the *input translation table*. When a character in *from-string* is found, the character in *string-expression* is converted to the character in *to-string* that is in the corresponding position of the character in *from-string*.

from-string must return a value that is a built-in character or graphic string data type that is not a LOB.

If *from-string* contains duplicate characters, the first occurrence of the character is used, and no warning is issued. The default value for *from-string* is a string that starts with the character X'00' and ends with the character X'FF' (decimal 255).

pad

An expression that specifies the character with which to pad *to-string* if its length is less than *from-string*. *pad* is an expression that must return a value that is a built-in character or graphic string data type that is not a LOB and has a length of 1. A length of 1 is one single byte for character strings and one double byte string for graphic strings. The default is a blank that is appropriate for *string-expression*.

If *string-expression* is the only argument that is specified, the SBCS characters of its value are converted to uppercase based on the LC_CTYPE locale in effect for the statement, which is determined by special register CURRENT LOCALE

LC_CTYPE. For example, a-z are converted to A-Z, and characters with diacritical marks are converted to their uppercase equivalent, if any. (For a description of the uppercase tables that are used for this conversion, see *IBM National Language Support Reference Manual Volume 2*.) For Unicode data, usage of the TRANSLATE function (the TRANSLATE function with one argument is equivalent to the UPPER function) can result in expansion if certain characters are processed. See the “UPPER” on page 552 function for more information. You should ensure that the result string is large enough to contain the result of the expression.

If the LC_CTYPE locale is blank when the function is executed, the result of the function depends on the data type of *string-expression*.

- For ASCII and EBCDIC, *string-expression* must not specify a graphic string expression. For a character string expression, characters a-z are converted to A-Z and characters with diacritical marks are not translated. If the string contains MIXED or DBCS characters, full-width Latin uppercase letters A-Z are converted to full-width lowercase letters a-z.
- For Unicode, *string-expression* can be either a character string expression or a graphic string expression, and LOCALE LC_CTYPE must be blank (no locale specified). The characters a-z are converted to A-Z and all other characters, including characters with diacritic marks, are left unchanged. Full-width Latin uppercase letters A-Z are converted to full-width Latin lowercase letters a-z.

If more than one argument is specified, the result string is built character-by-character from *string-expression* with each character in *from-string* being converted to the corresponding character in *to-string*. For each character in *string-expression*, the *from-string* is searched for the same character. If the character is found to be the *n*th character in *from-string*, the resulting string will contain the *n*th character from *to-string*. If *to-string* is less than *n* characters long, the resulting string will contain the *pad*. If the character is not found in *from-string*, it is moved to the result string without being converted.

The string can contain mixed data. If only one argument is specified, the UPPER function is performed on the argument, and the rules for operating on mixed data in the UPPER function are observed. Full-width Latin lowercase a-z are converted to full-width Latin uppercase letters A-Z. Otherwise, the function operates on a strict byte-count basis, and the result is not necessarily a properly formed mixed data character string.

The encoding scheme of the result is the same as *string-expression*. The data type of the result of the function depends on the data type of *string-expression*, *to-string*, *from-string*, and *pad*:

- VARCHAR if *string-expression* is a character string. The CCSID of the result depends on the arguments:
 - If *string-expression*, *to-string*, *from-string*, or *pad* is bit data, the result is bit data.
 - If *string-expression*, *to-string*, *from-string*, and *pad* are all SBCS:
 - If *string-expression*, *to-string*, *from-string*, and *pad* are all SBCS Unicode data, the CCSID of the result is the CCSID for SBCS Unicode data.
 - If *string-expression* is SBCS Unicode data, and *to-string*, *from-string*, or *pad* are not SBCS Unicode data, the CCSID of the result is the mixed CCSID for Unicode data.
 - Otherwise, the CCSID of the result is the same as the CCSID of *string-expression*.

- Otherwise, the CCSID of the result is the mixed CCSID that corresponds to the CCSID of *string-expression*. However, if the input is EBCDIC or ASCII and there is no corresponding system CCSID for mixed, the CCSID of the result is the CCSID of *string-expression*.
- VARGRAPHIC if *string-expression* is a graphic. The CCSID of the result is the same as the CCSID of *source-string*.

If the first argument can be null, the result can be null. If the argument is null, the result is the null value.

Example 1: Return the string 'abcdef' in uppercase characters. Assume that the locale in effect is blank.

```
SELECT TRANSLATE ('abcdef')
FROM SYSIBM.SYSDUMMY1
```

The result is the value 'ABCDEF'.

Example 2: Assume that host variable *SITE* has a data type of VARCHAR(30) and contains 'Hanauma Bay'.

```
SELECT TRANSLATE (:SITE)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'HANAUMA BAY'. The result is all uppercase characters because only one argument is specified.

```
SELECT TRANSLATE (:SITE, 'j', 'B')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Hanauma jay'.

```
SELECT TRANSLATE (:SITE, 'ei', 'aa')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Heneume Bey'.

```
SELECT TRANSLATE (:SITE, 'bA', 'Bay', '%')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'HAnAumA bA%'.

```
SELECT TRANSLATE (:SITE, 'r', 'Bu')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Hana ma ray'.

Example 3: Assume that host variable *SITE* has a data type of VARCHAR(30) and contains 'Pivabiska Lake Place'.

```
SELECT TRANSLATE (:SITE, '$$', 'Ll')
FROM SYSIBM.SYSDUMMY1
```

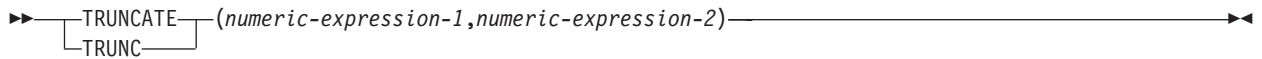
Returns the value 'Pivabiska \$ake P\$ace'.

```
SELECT TRANSLATE (:SITE, 'pLA', 'Place', '.')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'pivAbiskA LAk. pLA..'.

TRUNCATE or TRUNC

The TRUNCATE function returns the first argument, truncated as specified. Truncation is to the number of places to the right or left of the decimal point this is specified by the second argument.



The schema is SYSIBM.

numeric-expression-1

An expression that returns a value of any built-in numeric data type.

If *expression-1* is a decimal floating-point data type, the DECFLOAT ROUNDING MODE will not be used. The rounding behavior of TRUNCATE corresponds to a value of ROUND_DOWN. If a different rounding behavior is desired, use the QUANTIZE function.

numeric-expression-2

An expression that returns a value that is a built-in SMALLINT or INTEGER data type. The absolute value of the integer specifies the number of places to truncate. The value of *numeric-expression-2* determines whether truncation is to the right or left of the decimal point.

If *numeric-expression-2* is not negative, *numeric-expression-1* is truncated to the absolute value of *numeric-expression-2* places to the right of the decimal point.

If *numeric-expression-2* is negative, *numeric-expression-1* is truncated to 1 + (the absolute value of *numeric-expression-2*) places to the left of the decimal point. If 1 + (the absolute value of *numeric-expression-2*) is greater than or equal to the number of digits to the left of the decimal point, the result is 0. For example, TRUNCATE(748.58, -4) returns 0.

The result of the function has the same data type and length attribute as the first argument. The result can be null. If any argument is null, the result is the null value.

Example 1: Using sample employee table DSN8910.EMP, calculate the average monthly salary for the highest paid employee. Truncate the result to two places to the right of the decimal point.

```
SELECT TRUNCATE(MAX(SALARY/12),2)
FROM DSN8910.EMP;
```

Because the highest paid employee in the sample employee table earns \$52750.00 per year, the example returns the value 4395.83.

Example 2: Return the number 873.726 truncated to 2, 1, 0, -1, -2, -3, and -4 decimal places respectively.

```
SELECT TRUNC(873.726,2),
       TRUNC(873.726,1),
       TRUNC(873.726,0),
       TRUNC(873.726,-1),
       TRUNC(873.726,-2)
```

```

        TRUNC(873.726,-3),
        TRUNC(873.726,-4)
FROM TABLEX
WHERE INTCOL = 1234;

```

This example returns the values 873.720, 873.700, 873.000, 870.000, 800.000, 0000.000, and 0000.000.

Example 3: Calculate both positive and negative numbers.

```

SELECT TRUNCATE( 3.5, 0),
       TRUNCATE( 3.1, 0),
       TRUNCATE(-3.1, 0),
       TRUNCATE(-3.5, 0)
FROM TABLEX;

```

This example returns: the values 3.0, 3.0, -3.0, -3.0.

TRUNC_TIMESTAMP

The TRUNC_TIMESTAMP function returns a timestamp that is the *expression*, truncated to the unit that is specified by the *format-string*.

```
TRUNC_TIMESTAMP(expression ['DD'], format-string)
```

The schema is SYSIBM.

expression

An expression that returns a value of any of the following built-in data types: a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

format-string

An expression that returns a built-in character string or graphic string data type, with a length that is not greater than 255 bytes. *format-string* contains a template of how the timestamp represented by *expression* should be truncated. For example, if *format-string* is 'DD', the timestamp that is represented by *expression* is truncated to the nearest day. *format-string* must be a valid template for a timestamp, and not include leading or trailing blanks. Allowable values for *format-string* are listed in the following table.

Table 65. ROUND_TIMESTAMP and TRUNC_TIMESTAMP format models

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
CC SCC	Century	Input Value: 1897-12-04-12.22.22.000000	Input Value: 1897-12-04-12.22.22.000000
	Rounds up to the start of the next century after the 50th year of the century (for example on 1951-01-01-00.00.00).	Result: 1900-01-01-00.00.00.000000	Result: 1800-01-01-00.00.00.000000
	Not valid for a TIME argument.		
YYYYY YYYY YEAR SYEAR YYY YY Y	Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000	Input Value: 1897-12-04-12.22.22.000000
		Result: 1898-01-01-00.00.00.000000	Result: 1897-01-01-00.00.00.000000
IYYY IYY IY I	ISO Year (Rounds up on July 1st)	Input Value: 1897-12-04-12.22.22.000000	Input Value: 1897-12-04-12.22.22.000000
		Result: 1898-01-01-00.00.00.000000	Result: 1897-01-01-00.00.00.000000

Table 65. *ROUND_TIMESTAMP* and *TRUNC_TIMESTAMP* format models (continued)

Format model	Rounding or truncating unit	ROUND_TIMESTAMP example	TRUNC_TIMESTAMP example
Q	Quarter (Rounds up on the sixteenth day of the second month of the quarter)	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-04-01-00.00.00.000000
MONTH MON MM RM	Month (Rounds up on the sixteenth day of the month)	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-18-12.15.00.000000 Result: 1999-06-01-00.00.00.000000
WW	Same day of the week as the first day of the year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-06-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-04-29-00.00.00.000000
IW	Same day of the week as the first day of the ISO year (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the ISO year)	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-08-00.00.00.000000	Input Value: 2000-05-05-12.15.00.000000 Result: 2000-05-01-00.00.00.000000
W	Same day of the week as the first day of the month (Rounds up on the 12th hour of the 3rd day of the week, with respect to the first day of the month)	Input Value: 2000-05-17-12.12.30.000000 Result: 2000-05-15-00.00.00.000000	Input Value: 2000-05-17-12.15.00.000000 Result: 2000-05-15-00.00.00.000000
DDD DD J	Day (Rounds up on the 12th hour of the day)	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-17-00.00.00.000000
DAY DY D	Starting day of the week (Rounds up with respect to the 12th hour of the third day of the week. The first day of the week is always Sunday).	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-21-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-14-00.00.00.000000
HH HH12 HH24	Hour (Rounds up at 30 minutes)	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-17-23.00.00.000000
MI	Minute (Rounds up at 30 seconds)	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.59.00.000000	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.58.00.000000
SS	Second (Rounds up at 500000 microseconds)	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.46.000000	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.45.000000

The result of the function is a **TIMESTAMP**. The result can be null; if any argument is null, the result is the null value.

| The result CCSID is the appropriate CCSID of the argument encoding scheme and
| the result subtype is the appropriate subtype of the CCSID.

Example: Set the host variable *TRNK_TMSTMP* with the specified date rounded to the nearest year value.

```
SET :TRNK_TMSTMP = TRUNC_TIMESTAMP('2008-03-14-17.30.00', 'YEAR');
```

The host variable *TRNK_TMSTMP* is set with the value '2008-01-01-00.00.00.000000'.

UCASE

The UCASE function returns a string in which all the characters have been converted to uppercase characters, based on the CCSID of the argument. The UCASE function is identical to the UPPER function.

I ►► UCASE(*string-expression* [, *—locale-name*] [, *—integer*]) —►◄

The schema is SYSIBM.

For more information, see “UPPER” on page 552.

UNICODE

The UNICODE function returns the Unicode UTF-16 code value of the leftmost character of the argument as an integer.

►►—UNICODE(*string-expression*)—◄◄

The schema is SYSIBM.

string-expression can be of any built-in string data type that is not a LOB.

If the argument is ASCII, EBCDIC, or Unicode UTF-8, it is first converted to a Unicode UTF-16 string (CCSID 1200) before the function is executed.

The result of the function is an INTEGER. The result can be null; if the argument is the null value, the result is the null value.

Example: The following example returns the Unicode value of 峰 as an integer and assigns the value to the host variable *hv*:

Set :hv = UNICODE('峰'); *hv* is set to an integer with a value '23792'.

UNICODE_STR

The UNICODE_STR function returns a string in Unicode UTF-8 or UTF-16, depending on the specified option. The string represents a Unicode encoding of the input string.

```
►►—UNICODE_STR(string-expression [ , UTF8 ] [ , UTF16 ])—————►
```

The schema is SYSIBM.

string-expression

An expression that returns a value of a built-in character or graphic string. If the string is a character string, it cannot be bit data. Values that are preceded by a backslash ('\') are treated as Unicode UTF-16 characters (for example '\0041' is the Unicode UTF-16 representation for 'A'). A double backslash '\\' indicates a backslash in the string.

UTF8 or UTF16

Specifies the Unicode encoding of the result. If UTF8 is specified, the result is returned as a Unicode UTF-8 character string. If UTF16 is specified, the result is returned as a Unicode UTF-16 graphic string. UTF8 is the default.

The result of the function depends on the second argument:

- VARCHAR if **UTF8** is specified
- VARGRAPHIC if **UTF16** is specified

The length attribute of the result depends on the second argument (**UTF8** or **UTF16**). The length attribute of the result is calculated using the formulas in Table 35 on page 126. If the result is a character string, the length attribute of the result is $\text{MAX}(n, 32704)$. If the result is a graphic string, the length attribute of the result is $\text{MAX}(n, 16352)$. Where n is the result of applying the formulas in Table 35 on page 126 based on input and output data types.

If the actual length of the result string exceeds the maximum for the return type, an error occurs.

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

UNISTR can be specified as a synonym for UNICODE_STR.

Example: The following example sets the host variable *HV1* to a VARCHAR value that represents the Unicode UTF-8 string that corresponds to the argument:

```
SET :HV1 = UNICODE_STR('Hi, my name is \5CF0');
```

HV1 is assigned a Unicode UTF-8 string with the following value 'Hi, my name is

峰 ,

UPPER

The UPPER function returns a string in which all the characters have been converted to uppercase characters.

```
UPPER(string-expression [, —locale-name—] [, —integer—])
```

The schema is SYSIBM.

string-expression

An expression that specifies the string to be converted. *string-expression* must return a value that is a built-in character or graphic string. A character string argument must not be a CLOB, and a graphic string argument must not be a DBCLOB.

locale-name

A string constant or a string host variable other than a CLOB or DBCLOB that specifies a valid locale name. If *locale-name* is not in EBCDIC, it is converted to EBCDIC. The length of *locale-name* must be between 1 and 255 bytes of the EBCDIC representation. The value of *locale-name* is not case sensitive and must be a valid locale. For information on locales and their naming conventions, see *z/OS C/C++ Programming Guide*. Some examples of locales include:

```
Fr_BE
Fr_FR@EURO
En_US
Ja_JP
```

For EBCDIC and ASCII data, there are two options for *locale-name*:

- blank — *string-expression* must not specify a graphic string. For a character string, characters a-z are converted to A-Z and characters with diacritical marks are not converted. If the string contains MIXED or DBCS characters, full-width Latin lowercase letters a-z are converted to full-width uppercase letters A-Z. This is the default value specified in the LOCALE LC_CTYPE on the installation panel DSNTIPF.
- a locale — locale specific casing will be done using the "LOCAL" casing capabilities, as specified during installation

For optimal performance, use blank for *locale-name* unless your data must be interpreted using the rules provided for specific locales.

For Unicode data, there are three options for *locale-name*:

- blank — simple casting on A-Z, a-z, full-width Latin lowercase letters a-z, and full-width Latin uppercase letters A-Z. Characters with diacritics are not affected.
- "UNI" — If the value "UNI" is specified, casting will use both the "NORMAL" and "SPECIAL" casing capabilities as described in *z/OS Support for Unicode: Using Conversion Services*.
- a locale — In this case, locale specific casing will be performed using the "LOCALE" casing capabilities as described in *z/OS Support for Unicode: Using Conversion Services*.

The value of the host variable must not be null. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value. The locale name must be:

- left justified within the host variable
- padded on the right with blanks if its length is less than that of the host variable and the host variable is in fixed length CHAR or GRAPHIC data type

If *locale-name* is not specified, the locale is determined by special register CURRENT LOCALE LC_CTYPE. For information about the special register, see “CURRENT LOCALE LC_CTYPE” on page 140.

If the UPPER function is referenced in an index that is based on an expression, *locale-name* must be specified

integer

An integer value that specifies the length attribute of the result. If specified, *integer* must be an integer constant between 1 and 32704 bytes in the representation of the encoding scheme of *string-expression*.

If *integer* is not specified, the length attribute of the result is the same as the length of *string-expression*.

For Unicode data, usage of the UPPER function can result in expansion if certain characters are processed. For example, UPPER(UX'FB03') will result in UX'004600460049'. You should ensure that the result string is large enough to contain the result of the expression.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example 1: Return the string 'abcdef' in uppercase characters. Assume that the locale in effect is blank.

```
SELECT UPPER('abcdef')
FROM SYSIBM.SYSDUMMY1
```

The result is the value 'ABCDEF'.

Example 2: Return the string 'ffi' in the uppercase characters ('FFI'). Assume that the locale in effect is "UNI".

```
SELECT UPPER(UX'FB03')
FROM SYSIBM.SYSDUMMYU;
```

This would result in an error because of the expansion that occurs when certain Unicode characters are processed. To avoid the error, you would need to use the following statement instead:

```
SELECT UPPER(CAST(UX'FB03' AS VARCHAR(3)))
FROM SYSIBM.SYSDUMMYU;
```

The result of the preceding statement is the value 'FFI'.

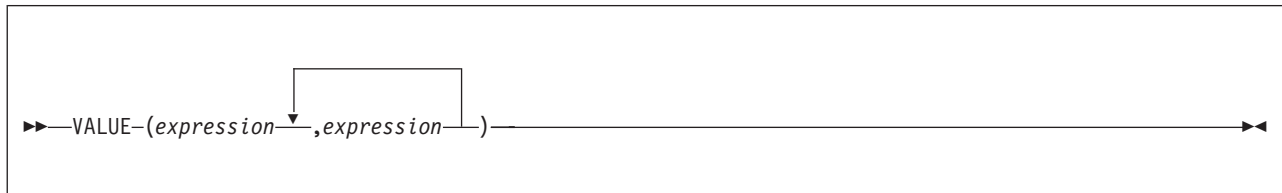
Example 3: Create an index EMPLOYEE_NAME_UPPER for table EMPLOYEE based on built-in function UPPER with locale name 'Fr_FR@EURO'.

```
CREATE INDEX EMPLOYEE_NAME_UPPER
ON EMPLOYEE (UPPER(LASTNAME, 'Fr_FR@EURO', 60),
             UPPER(FIRSTNAME, 'Fr_FR@EURO', 60),
             ID);
```

| The result is the value 'ABCDEF'.

VALUE

The VALUE function returns the value of the first non-null expression.



The schema is SYSIBM.

Syntax alternatives: The VALUE function can be specified in place of the COALESCE function. COALESCE should be used for conformance to SQL 2003 Core. For more information, see “COALESCE” on page 315.

VARBINARY

The VARBINARY function returns a VARBINARY (varying-length binary string) representation of a string of any type.

The diagram shows the syntax for the VARBINARY function. It starts with a double arrow pointing right, followed by the text 'VARBINARY(' in a monospace font. Then, 'string-expression' is written in an italicized font. This is followed by a bracketed section containing a comma and the text 'integer' in an italicized font. The bracketed section is followed by a closing parenthesis ')'. Finally, a long double arrow points to the right, indicating the return value.

```
»—VARBINARY(string-expression[, integer])—»
```

The schema is SYSIBM.

string-expression

An expression that returns a value that is a built-in character string, graphic string, binary string, or a row ID type.

integer

An integer value that specifies the length attribute of the resulting binary string. The value must be an integer between 1 and 32704 inclusive. If integer is not specified:

- If the *string-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the *string-expression*, unless the *string-expression* is a graphic string. In this case, the length attribute of the result is twice the length attribute of the *string-expression*.

The result of the function is a varying-length binary string. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The actual length of the result is the minimum of the length attribute of the result and the actual length of the *string-expression* (or twice the length of the *string-expression* if *string-expression* returns a graphic string). If the length of the *string-expression* is less than the length of the result, the result is padded with hexadecimal zeros up to the length of the result. If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed, and a warning is returned unless the *string-expression* is a character string and all the truncated characters are blanks, or the *string-expression* is a graphic string and all the truncated characters are double-byte blanks.

Example 1: The following function returns a varying-length binary string with a length attribute 1, actual length 0, and a value of empty string:

```
SELECT VARBINARY('')
FROM SYSIBM.SYSDUMMY1;
```

Example 2: The following function returns a varying-length binary string with a length attribute 5, actual length 3, and a value BX'D2C2C8':

```
SELECT VARBINARY('KBH',5)
FROM SYSIBM.SYSDUMMY1;
```

Example 3: The following function returns a varying-length binary string with a length attribute 3, actual length 3, and a value BX'D2C2C8'

```
|      SELECT VARBINARY('KBH ',3)
|      FROM SYSIBM.SYSDUMMY1;
```

| *Example 4:* The following function returns a varying-length binary string with a
| length attribute 3, actual length 3, and a value BX'D2C2C8', a warning is also
| returned.

```
|      SELECT VARBINARY('KBH-93',3)
|      FROM SYSIBM.SYSDUMMY1;
```

| *Example 5:* The following function returns a varying-length binary string with a
| length attribute 3, actual length 3, and a value BX'C1C2C3', a warning is also
| returned.

```
|      SELECT VARBINARY(BINARY('ABC',5),3)
|      FROM SYSIBM.SYSDUMMY1;
```

|

VARCHAR

The VARCHAR function returns a varying-length character string representation of the value specified by the first argument. The first argument can be a character string, a graphic string, a datetime value, an integer number, a decimal number, a floating-point number, or a row ID value.

Character to Varchar:

►►—VARCHAR(*character-expression* [, —*integer* [, —CODEUNITS16 [—CODEUNITS32 [—OCTETS]]]])—►►

Graphic to Varchar:

►►—VARCHAR(*graphic-expression* [, —*integer* [, —CODEUNITS16 [—CODEUNITS32]]])—►►

Datetime to Varchar:

►►—VARCHAR(*datetime-expression*)—►►

Integer to Varchar:

►►—VARCHAR(*integer-expression*)—►►

Decimal to Varchar:

►►—VARCHAR(*decimal-expression* [, —*decimal-character*])—►►

Decimal floating point to Varchar:

►►—VARCHAR(*decimal-floating-point-expression*)—►►

Floating-point to Varchar:

►►—VARCHAR(*floating-point-expression*)—►►

Row ID to Varchar:

►►—VARCHAR(*row-ID-expression*)—►►

The schema is SYSIBM.

The result of the function is a varying-length character string (VARCHAR). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Character to Varchar

character-expression

An expression that returns a value that is a built-in character data type.

integer

Specifies the length attribute for the resulting varying-length character string. The value must be between 1 and 32767, expressed in the units that are either implicitly or explicitly specified. If the length is not specified, the length of the result is the same as the length of *character-expression*.

If CODEUNITS16, CODEUNITS32, or OCTETS is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If a length attribute is not specified and if the *character-expression* is an empty string constant, the length attribute of the result is 1 and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the unit that is used to express *integer*. If *character-expression* is a character string that is defined as bit data, CODEUNITS16 and CODEUNITS32 cannot be specified.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

OCTETS

Specifies that *integer* is expressed in terms of bytes.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String unit specifications” on page 76.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression* is greater than the length attribute of the result, the result is truncated. Unless all the truncated characters are blanks appropriate for *character-expression*, a warning is returned.

If *character-expression* is bit data, the result is bit data. Otherwise, the CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to Varchar

graphic-expression

An expression that returns a value that is a built-in graphic data type.

integer

The length attribute for the resulting varying-length graphic string. The value must be between 1 and 32740, expressed in the units that are either implicitly or explicitly specified.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If a length attribute is not specified, the length attribute of the result is determined as follows (where *n* is the length attribute of the first argument):

- If the *graphic-expression* is the empty graphic string constant, the length attribute of the result is 1.
- If the result is SBCS data, the result length is *n*.
- If the result is mixed data, the result length is $3 * (\text{length}(\text{graphic-expression}))$.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 or CODEUNITS32, see “String unit specifications” on page 76.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of the graphic expression is greater than the length attribute of the result, the result is truncated. Unless all the truncated characters were blanks appropriate for *graphic-expression*, a warning is returned.

The CCSID of the result is the character mixed CCSID that corresponds to the graphic CCSID of *graphic-expression*.

Datetime to Varchar

datetime-expression

An expression whose value has one of the following three built-in data types:

date The result is a varying-length character string representation of the date in the format that is specified by the DATE precompiler option, if one is provided, or else field DATE FORMAT on installation panel DSNTIP4 specifies the format. If the format is to be LOCAL, field LOCAL DATE LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length attribute and actual length of the result is 10.

LOCAL denotes the local format at the DB2 that executes the SQL statement. If LOCAL is used for the format, a time exit routine must be installed at that DB2.

An error occurs if the second argument is specified and is not a valid value.

time The result is a varying-length character string representation of the time in the format specified by the TIME precompiler option, if one is provided, or else field TIME FORMAT on installation panel DSNTIP4 specifies the format. If the format is to be LOCAL, the field LOCAL TIME LENGTH on installation panel DSNTIP4 specifies the length of the result. Otherwise, the length attribute and actual length of the result is 8.

LOCAL denotes the local format at the DB2 that executes the SQL statement. If LOCAL is used for the format, a time exit routine must be installed at that DB2.

An error occurs if the second argument is specified and is not a valid value.

timestamp

The result is the varying-length character string representation of the timestamp. The length attribute and actual length of the result is 26.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Integer to Varchar

integer-expression

An expression that returns a value that is a built-in integer data type (SMALLINT, INTEGER, BIGINT).

The result is a varying-length character string representation (VARCHAR) of the argument in the form of an SQL integer constant.

The length attribute of the result depends on whether the argument is a small or large integer as follows:

- If the argument is a small integer, the length attribute of the result is 6 bytes.
- If the argument is a large integer, the length attribute of the result is 11 bytes.
- If the argument is a big integer, the length attribute of the result is 20 bytes.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit.

The CCSID of the result is the SBCS CCSID of the appropriate encoding scheme.

Decimal to Varchar

decimal-expression

An expression that returns a value that is a built-in decimal data type. To specify a different precision and scale for the expression's value, apply the DECIMAL function to the expression before applying the VARCHAR function.

decimal-character

Specifies the single-byte character constant (CHAR or VARCHAR) that is used to delimit the decimal digits in the result character string. The character must not be a digit, a plus sign (+), a minus sign (-), or a blank. The default is the period (.) or comma (,). For information on what factors govern the choice, see "Decimal point representation" on page 251.

The result is a varying-length character string representation of the argument. The result includes a decimal character and up to p digits where p is the precision of *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is $2+p$ where p is the precision of *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing zeros are included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit. If the scale of *decimal-expression* is zero, the decimal character is not returned.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 43.

Decimal floating-Point to Varchar

decimal-floating-point-expression

An expression that returns a value that is the built-in DECFLOAT data type.

The result is the varying-length character string representation of the argument in the form of an SQL decimal floating-point constant.

The length attribute of the result is 42 bytes.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see "Determining the encoding scheme and CCSID of a string" on page 43.

Floating-Point to Varchar

floating-point-expression

An expression that returns a value that is a built-in floating-point data type.

The result is a varying-length character string representation (VARCHAR) of the argument in the form of an SQL floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by a period

and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If the argument is zero, the result is '0E0'.

The CCSID of the result is determined from the context in which the function was invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Row ID to Varchar

row-ID-expression

An expression that returns a value that is a built-in row ID data type.

The result is a varying-length character string representation (VARCHAR) of the argument. It is bit data.

The length attribute of the result is 40. The actual length of the result is the length of *row-ID-expression*.

Example 1: Assume that host variable *JOB_DESC* is defined as VARCHAR(8). Using sample table DSN8910.EMP, set *JOB_DESC* to the varying-length string equivalent of the job description (column JOB defined as CHAR(8)) for the employee with the last name of 'QUINTANA'.

```
SELECT VARCHAR(JOB)
      INTO :JOB_DESC
      FROM DSN8910.EMP
      WHERE LASTNAME = 'QUINTANA';
```

Example 2: FIRSTNME is a VARGRAPHIC(6) column in a Unicode table T1. One of its values is the string 'Jürgen' (X'004A00FC007200670055006E'). When FIRSTNME has this value:

Function ...	Returns ...
VARCHAR(FIRSTNME,3,CODEUNITS32)	'Jür' -- x'4AC3BC72'
VARCHAR(FIRSTNME,3,CODEUNITS16)	'Jür' -- x'4AC3BC72'
VARCHAR(FIRSTNME,3,OCTETS)	'Jü' -- x'4AC3BC'

VARCHAR_FORMAT

The VARCHAR_FORMAT function returns a character string representation of a timestamp that is based on applying the specified format string argument to the first argument. The expression must be formatted according to a specified character template.

►►—VARCHAR_FORMAT(*timestamp-expression*,*format-string*)—◄◄

The schema is SYSIBM.

Timestamp to varchar

timestamp-expression

An expression that returns a value that must be a DATE or TIMESTAMP.

format-string

An expression that returns a built-in character string data type that is not a LOB and has a length attribute that is not greater than 255 bytes.

The value is a template for how timestamp-expression is to be formatted.

A valid *format-string* can contain a combination of the format elements listed below. Two format elements can be separated by one or more of the following separator characters.

- minus sign (-)
- period (.)
- forward slash (/)
- comma (,)
- apostrophe (')
- semicolon (;)
- colon (:)
- blank ()

Separator characters can also be specified at the start or end of *format-string*. *format-string* can also be an empty string, a string of blanks, or a string of separator characters.

The following table lists the valid format elements that *format-string* can contain.

Table 66. Valid format elements of format-string

Format element	Description (assuming the default is to return leading zeros)
CC	Century (00-99). If the last two digits of the four digit year are zero, the result is the first two digits of the year. Otherwise, the result is the first two digits of the year plus one.
D ¹	Day of the week (1-7). 1 is Sunday and 7 is Saturday.

Table 66. Valid format elements of format-string (continued)

Format element	Description (assuming the default is to return leading zeros)
DD	Day of the month (01-31).
DDD	Day of the year (001-366).
FF or FF n	Fractional seconds (0-999999999999).
	The number n is used to specify the number of digits to include in the returned value. Valid values for n are 1-12 with no leading zeros. Specifying FF is equivalent to specifying FF6. If the timestamp precision of <i>timestamp-expression</i> is less than what is specified by the format, zero digits are padded onto the right of the specified digits.
HH	Hour of the day (01-12).
HH12	Hour of the day (01-12).
HH24	Hour of the day (00-24).
I	ISO year (0-9). The last digit of the year based on the ISO week that is returned.
ID	ISO day of the week (1-7).
	1 is Monday and 7 is Sunday.
IW	ISO week of the year (01-53).
	The week starts on Monday and includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week of the year to contain January 4.
IY	ISO year (00-99).
	The last two digits of the year based on the ISO week that is returned.
IYY	ISO year (000-999).
	The last three digits of the year based on the ISO week that is returned.
IYYY	ISO year (0000-9999).
	The last four digits of the year based on the ISO week that is returned.
J	Julian date (0000000-9999999).
MI	Minute (00-59).
MM	Month (01-12).
	January is 01.
MONTH, Month, or month ^{1, 2}	Name of the month in uppercase, sentence case, or lowercase format in English.
MON, Mon, or mon ^{1, 2}	Three-character abbreviated name of the month in uppercase, sentence case, or lowercase format in English.
NNNNNN	Microseconds (000000-999999).
	This format is equivalent to specifying FF6.

Table 66. Valid format elements of format-string (continued)

Format element	Description (assuming the default is to return leading zeros)
Q	Quarter (1-4). January through March is 1.
RRRR	Year (0000-9999). RRRR behaves the same as YYYY.
RR	Last two digits of the year (00-99).
SS	Seconds (00-59).
SSSS	Seconds since the previous midnight (00000-86400).
W	Week of the month (1-5). Week 1 starts on the first day of the month and ends on the seventh day.
WW	Week of the year (01-53). Week 1 begins on January 1 and ends on January 7.
Y	Last digit of the year (0-9).
YY	Last two digits of the year (00-99).
YYY	Last three digits of the year (000-999).
YYYY	Year (0000-9999).
Notes:	
1. This format element is case sensitive. In cases where the format elements are ambiguous, the case insensitive format elements will be considered first.	
2. Only these exact spellings and case combinations can be used. If this format element is specified in an invalid case combination an error is returned.	

The result is a representation of *timestamp-expression* in the format specified by *format-string*. *format-string* is interpreted as a series of format elements that can be separated by one or more separator characters. A string of characters in *format-string* is interpreted as the longest matching element in the previous table. If two format elements that contain the same characters are not delimited by a separator character, the specification is interpreted, starting from the left, as the longest matching element in the table, and continues until matches are found for the remainder of the format string. For example, 'YYYYYYDD' is interpreted as the format elements, 'YYYY', 'YY', and 'DD'.

The result is the varying-length character string. The length attribute of the result is the maximum of 255 and the length attribute of *format-string*. The *format-string* determines the actual length of the result. The actual length must not be greater than the length attribute of the result. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is determined from the context in which the function is invoked. For more information, see “Determining the encoding scheme and CCSID of a string” on page 43.

Notes

Determinism:

VARCHAR_FORMAT is a deterministic function.

Using the 'D', 'Y', and 'y' format elements:

DB2 for z/OS does not support the 'DY', 'dy', and 'Dy' format elements that are supported by other platforms. If 'DY' or 'Dy' is specified in the format string, it is interpreted as the 'D' format element followed by the 'Y' or 'y' format element. This behavior might change in a future release. To ensure that a 'D' followed by 'Y' or 'y' is interpreted as two separate format elements, include a separator character after the 'D' format element.

Syntax alternatives:

TO_CHAR can be specified as a synonym for **VARCHAR_FORMAT**.

Example 1:

Set the character variable *TVAR* to a string representation of the timestamp value of *RECEIVED* from *CORPDATA.IN_TRAY*, formatted as 'YYYY-MM-DD HH24:MI:SS'.

```
SELECT VARCHAR_FORMAT(RECEIVED, 'YYYY-MM-DD HH24:MI:SS')
       INTO :TVAR
       FROM CORPDATA.IN_TRAY;
```

Assuming that the value in the *RECEIVED* column is 'January 1, 2000 at 10am', the following string is returned:

'2000-01-01 10:00:00'

Assuming that the value in the *RECEIVED* column is now one second before the beginning of the year 2000 ('December 31, 1999 at 23:59:59pm', the following string is returned:

'1999-12-31 23:59:59'

The result would be different if HH12 had been specified instead of HH24 in the format string:

'1999-12-31 11:59:59'

Example 2:

Assume that the variable *TMSTAMP* is defined as a **TIMESTAMP** and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result data type in each case is **VARCHAR(255)**.

Function invocation	Result
-----	-----
VARCHAR_FORMAT (TMSTAMP , 'YYYYMMDDHHMISSFF3')	20070309020738123
VARCHAR_FORMAT (TMSTAMP , 'YYYYMMDDHH24MISS')	20070309140738
VARCHAR_FORMAT (TMSTAMP , 'YYYYMMDDHHMI')	200703090207
VARCHAR_FORMAT (TMSTAMP , 'DD/MM/YY')	09/03/07
VARCHAR_FORMAT (TMSTAMP , 'MM-DD-YYYY')	03-09-2007
VARCHAR_FORMAT (TMSTAMP , 'J')	2454169
VARCHAR_FORMAT (TMSTAMP , 'Q')	1
VARCHAR_FORMAT (TMSTAMP , 'W')	2
VARCHAR_FORMAT (TMSTAMP , 'IW')	10
VARCHAR_FORMAT (TMSTAMP , 'WW')	10
VARCHAR_FORMAT (TMSTAMP , 'Month')	March
VARCHAR_FORMAT (TMSTAMP , 'MONTH')	MARCH
VARCHAR_FORMAT (TMSTAMP , 'MON')	MAR

Example 4:

Format the hour of the specified string representation of a timestamp using a 12 hour clock and a 24 hour clock:

```

SELECT
  VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH12'),
  VARCHAR_FORMAT(TIMESTAMP('1979-04-07-14.00.00.000000'), 'HH24'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH12'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH12'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH12'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-00.00.00.000000'), 'HH24'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-12.00.00.000000'), 'HH24'),
  VARCHAR_FORMAT(TIMESTAMP('2000-01-01-24.00.00.000000'), 'HH24')
FROM SYSIBM.SYSDUMMY1;

```

The previous SELECT statement returns the following values:

```
'02' '02' '14' '12' '12' '12' '12' '12' '12' '00' '12' '24'
```

Note that the values '00' and '24' on a 24 hour scale both map to a value of '12' on a 12 hour scale.

Example 5:

Format the month, day, and hour of the specified string representation of a timestamp using a 24 hour clock, and indicate that the result should not contain leading zeros for the components:

```

SELECT
  VARCHAR_FORMAT(TIMESTAMP('1979-04-07-09.14.00.000000'),
    'FM MM DD HH24'),
FROM SYSIBM.SYSDUMMY1;

```

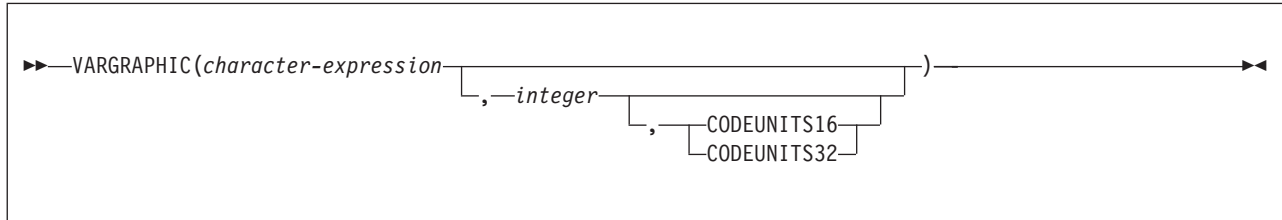
The previous SELECT statement returns the following values:

```
4 7 9
```

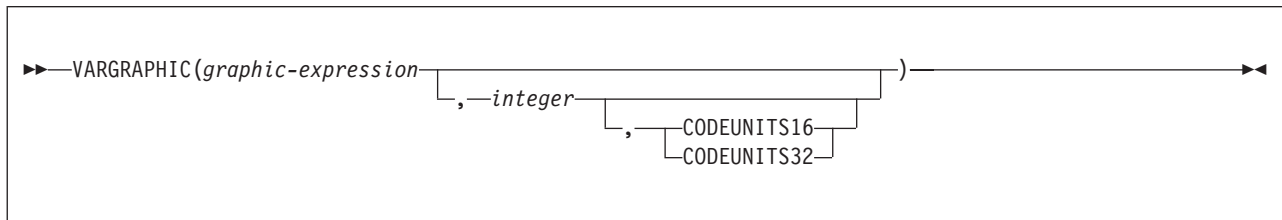
VARGRAPHIC

The VARGRAPHIC function returns a varying-length graphic string representation of a the first argument. The first argument can be a character string value or a graphic string value.

Character to Vargraphic:



Graphic to Vargraphic:



The schema is SYSIBM.

The result of the function is a varying-length graphic string (VARGRAPHIC).

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The length attribute and actual length of the result are measured in double-byte characters because the result is a graphic string.

Character to Vargraphic

character-expression

An expression that returns a value of a built-in character string data type that contains an EBCDIC-encoded or Unicode-encoded character string value. It cannot be BIT data. The argument does not need to be mixed data, but any occurrences of X'0E' and X'0F' in the string must conform to the rules for EBCDIC mixed data. (See “Character strings” on page 73 for these rules.)

integer

The length attribute of the resulting varying-length graphic string. The value must be an integer constant between 1 and 16352.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If *integer* is not specified and if the *character-expression* is an empty string constant or has a value X'0E0F', the length attribute of the result is 1 and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no corresponding CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression*, as measured in single-byte characters, is greater than the specified length of the result, as measured in double-byte characters, the result is truncated. Unless all the truncated characters are blanks appropriate for *character-expression*, a warning is returned.

The CCSID of the result is the graphic CCSID that corresponds to the character CCSID of *character-expression*. If the input is EBCDIC and there is no system CCSID for EBCDIC GRAPHIC data, the CCSID of the result is X'FFFE'.

For EBCDIC input data:

Each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data and its CCSID is *M*.
- The argument is SBCS data and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*. If there is no system CCSID for mixed data, conversion is to the coded character set that the system CCSID for SBCS data identifies.

The result is derived from *S* using the following steps:

- Each shift character (X'0E' or X'0F') is removed.
- Each double-byte character remains as is.
- Each single-byte character is replaced by a double-byte character.

The replacement for a single-byte character is the equivalent DBCS character if an equivalent exists. Otherwise, the replacement is X'FEFE'. The existence of an equivalent character depends on *M*. If there is no system CCSID for mixed data, the DBCS equivalent of X'*xx*' for EBCDIC is X'42*xx*', except for X'40', whose DBCS equivalent is X'4040'.

For Unicode input data:

Each character of *character-expression* determines a character of the result. The argument might need to be converted to the native form of mixed data before the

result is derived. Let *M* denote the system CCSID for mixed data. The argument is not converted if any of the following conditions is true:

- The argument is mixed data, and its CCSID is *M*.
- The argument is SBCS data, and its CCSID is the same as the system CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is *M*.

Otherwise, the argument is a new string *S* derived by converting the characters to the coded character set identified by *M*.

The result is derived from *S* using the following steps:

- Each non-supplementary character is replaced by a Unicode double-byte character (a UTF-16 code point). A non-supplementary character in UTF-8 is between 1 and 3 bytes.
- Each supplementary character is replaced by a pair of Unicode double-byte characters (a pair of UTF-16 code points).

The replacement for a single-byte character is the Unicode equivalent character if an equivalent exists. Otherwise, the replacement is X'FFFD'.

Graphic to Vargraphic

graphic-expression

An expression that returns a value of a built-in graphic string data type that contains an EBCDIC-encoded or Unicode-encoded graphic string value.

integer

The length attribute for the resulting varying-length graphic string. The value must be an integer constant between 1 and 16352.

If CODEUNITS16 or CODEUNITS32 is specified, see “Determining the length attribute of the final result” on page 79 for information about how to calculate the length attribute of the result string.

If *integer* is not specified and if the *graphic-expression* is an empty string constant, the length attribute of the result is 1 and the result is an empty string. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

CODEUNITS16 or CODEUNITS32

Specifies the unit that is used to express *integer*. If CODEUNITS16 or CODEUNITS32 is specified, the input is EBCDIC, and there is no corresponding CCSID for EBCDIC GRAPHIC data, an error occurs.

CODEUNITS16

Specifies that *integer* is expressed in terms of 16-bit UTF-16 code units.

CODEUNITS32

Specifies that *integer* is expressed in terms of 32-bit UTF-32 code units.

For more information about CODEUNITS16 and CODEUNITS32, see “String unit specifications” on page 76.

The actual length of the result depends on the number of characters in *graphic-expression*. If the length of *graphic-expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are double-byte blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *graphic-expression*.

Example 1: Assume that GRPHCOL is a VARGRAPHIC column in table TABLEX and MIXEDSTRING is a character string host variable that contains mixed data. For various rows in TABLEX, an application uses a positioned UPDATE statement to replace the value of GRPHCOL with the value of MIXEDSTRING. Before GRPHCOL can be updated, the current value of MIXEDSTRING must be converted to a varying-length graphic string. The following statement shows how to code the VARGRAPHIC function within the UPDATE statement to ensure this conversion.

```
EXEC SQL UPDATE TABLEX
  SET GRPHCOL = VARGRAPHIC(:MIXEDSTRING)
  WHERE CURRENT OF CRSNAME;
```

Example 2: FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the string 'Jürgen'. When FIRSTNAME has this value:

Function ...	Returns ...

VARGRAPHIC(FIRSTNAME,3,CODEUNITS32)	'Jür' -- x'004A00FC0072'
VARGRAPHIC(FIRSTNAME,3,CODEUNITS16)	'Jür' -- x'004A00FC0072'
VARGRAPHIC(FIRSTNAME,3,OCTETS)	An error because OCTETS not allowed

WEEK

The WEEK function returns an integer in the range of 1 to 54 that represents the week of the year. The week starts with Sunday, and January 1 is always in the first week.

►►—WEEK(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a graphic string. If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 1: Using sample table DSN8910.PROJ, set the integer host variable *WEEK* to the week of the year that project 'AD2100' ended.

```
SELECT WEEK(PRENDATE)
  INTO :WEEK
  FROM DSN8910.PROJ
  WHERE PROJNO = 'AD2100';
```

The result is that *WEEK* is set 6.

WEEK_ISO

The WEEK_ISO function returns an integer in the range of 1 to 53 that represents the week of the year. The week starts with Monday and includes seven days. Week 1 is the first week of the year that contains a Thursday, which is equivalent to the first week that contains January 4.

►►—WEEK_ISO(*expression*)—◄◄

With the WEEK_ISO function, the first one, two, or three days in January might be included in the last week of the previous year. Likewise, the last one, two, or three days in December might be included in the first week of the next year.

The schema is SYSIBM.

The argument must be a date, a timestamp, or a valid string representation of a date or timestamp. A string representation must not be a CLOB or DBCLOB value and must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.

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 1: Using sample table DSN8910.PROJ, set the integer host variable WEEKISO to the week of the year that project 'AD2100' ended.

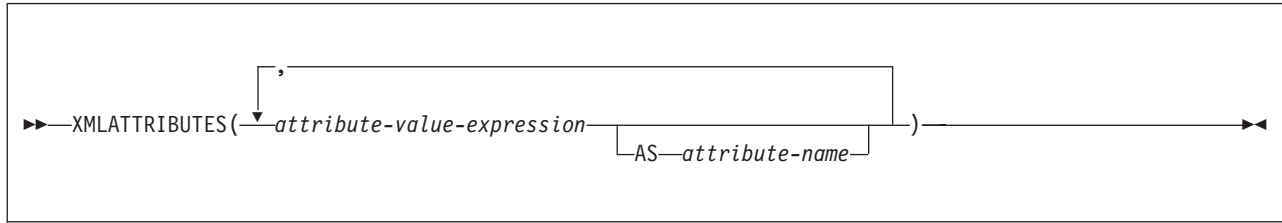
```
SELECT WEEK_ISO(PRENDATE)
  INTO :WEEKISO
  FROM DSN8910.PROJ
  WHERE PROJNO = 'AD2100';
```

Example 2: The following list shows what is returned by the WEEK_ISO function for various dates.

DATE:	WEEK_ISO returns:
-----	-----
2003-12-28	52
2003-12-31	1
2004-01-01	1
2005-01-01	53
2005-01-04	1
2005-12-31	52
2006-01-01	52
2006-01-03	1

XMLATTRIBUTES

The XMLATTRIBUTES function constructs XML attributes from the arguments.
This function can be used as an argument only for the XMLELEMENT function.



The schema is SYSIBM.

The result is an XML sequence that contains an XQuery attribute node for each non-null *attribute-value-expression* argument.

attribute-value-expression

An expression that returns a value for the attribute. The data type of *attribute-value-expression* must not be ROWID, a LOB, a distinct type that is based on a ROWID or a LOB, or XML.

The result of *attribute-value-expression* is mapped to an XML value according to the rules for mapping an SQL value to an XML value. If the expression is not a simple column reference, an attribute name must be specified.

AS *attribute-name*

Specifies an attribute name. The name is an SQL identifier that must be in the form of an XML qualified name, or QName. If *attribute-name* is a qualified name, the namespace prefix must be declared within the scope of the qualified name.

attribute-name cannot be 'xmlns' or prefixed with 'xmlns:'. A namespace is declared using the function XMLNAMESPACES. The attribute names for an element must be unique for the XML element to be well-formed.

If *attribute-name* is not specified, the expression for *attribute-value* must be a column name. The attribute name will be created from the column name using the fully escaped mapping from a column name to an XML attribute name.

The result of the function is an XML value. The result can be null; if all *attribute-value-expression* arguments are null, the result is the null value.

XMLCOMMENT

The XMLCOMMENT function returns an XML value with a single comment node from a string expression. The content of the comment node is the value of the input string expression, mapped to Unicode (UTF-8).

►►—XMLCOMMENT(*string-expression*)—◄◄

The schema is SYSIBM.

string-expression

An expression that returns a value of a built-in character or graphic string that is not a LOB and is not bit data. The result of *string-expression* is converted to UTF-8 and then parsed to check for conformance to the content of XML comment as specified by the following rules:

- '--' (double-hyphen) must not occur in the string expression
- The string expression must not end with a hyphen ('-')
- Each character of the string can be any Unicode character, excluding the surrogate blocks, X'FFFE', and X'FFFF'

If *string-expression* does not conform to the previous rules, an error is returned.

The result of the function is an XML value that is an XML sequence that contains one XML comment node. The result can be null; if the argument is null, the result is the null value.

Example: Generate an XML comment:

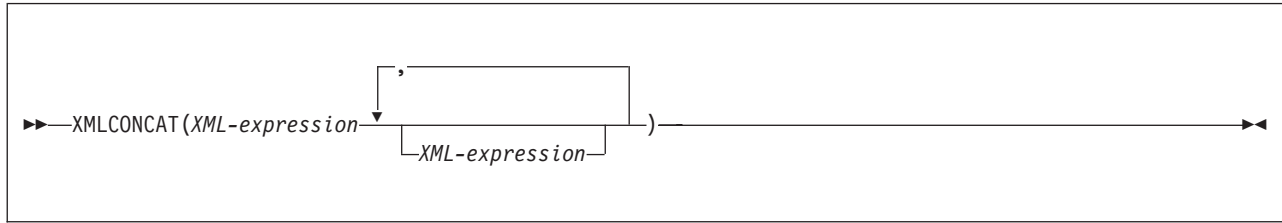
```
SELECT XMLCOMMENT('This is an XML comment')
FROM SYSIBM.SYSDUMMY1;
```

The result of the query would look similar to the following result:

```
<!--This is an XML comment-->
```

XMLCONCAT

The XMLCONCAT function returns an XML sequence that contains the concatenation of a variable number of XML input arguments.



The schema is SYSIBM.

XML-expression

An expression that returns an XML value.

The data type of the result is XML. The result of the function is an XML sequence that contains the concatenation of the non-null input XML values. Null values in the input are ignored. The result can be null; if the result of every input value is null, the result is the null value.

Example: Concatenate first name and last name elements by using 'first' and 'last' element names for each employee.

```
SELECT XMLSERIALIZE( XMLCONCAT
                     ( XMLELEMENT ( NAME "first", e.fname),
                       XMLELEMENT ( NAME "last", e.lname)
                     ) ) AS "result"
FROM employees e;
```

The result of the query would look similar to the following result:

result

```
<first>John</first><last>Smith</last>
<first>Mary</first><last>Smith</last>
```

XMLDOCUMENT

The XMLDOCUMENT function returns an XML value with a single document node and zero or more nodes as its children. The content of the generated XML document node is specified by a list of expressions.



The schema is SYSIBM.

XML-expression

An expression that returns an XML value. A sequence item in the XML value must not be an attribute node. If *XML-expression* returns a null value, it is ignored for further processing. However, if all *XML-expression* values are null, the result of the function is the null value.

The result of the function is an XML value. The result can be null; if all the arguments are null, the result is the null value.

The resulting XML value is built from the list of *XML-expression* arguments. The children of the resulting document node are constructed as follows:

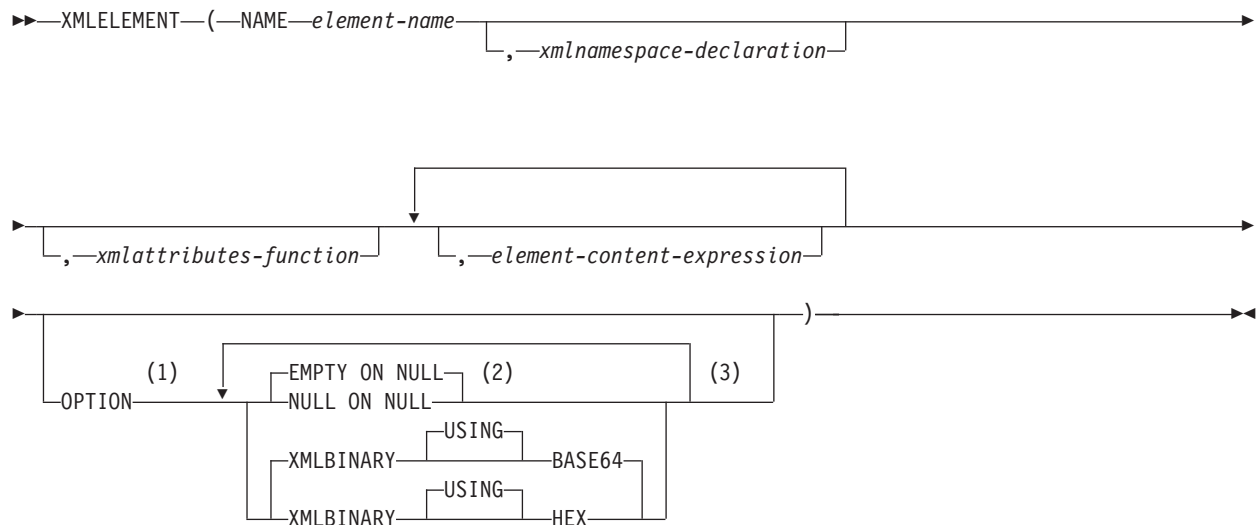
1. All of the non-null XML values that are returned by *XML-expression* are concatenated together. The result is a sequence of nodes or atomic values, which is referred to in the following steps as the *input sequence*. Any document node in the input sequence is replaced by copies of its children.
2. For each node in the input sequence, a new deep copy of the node is constructed. A *deep copy* of a node is a copy of the whole subtree that is rooted at that node, including the node itself and its descendants and attributes. Each copied node has a new node identity. Copied element nodes are given the type annotation 'xdt:untyped', and copied attribute nodes are given the type annotation 'xdt:untypedAtomic'. For each adjacent sequence of one or more atomic values that is returned in the input sequence, a new text node is constructed that contains the result of casting each atomic value to a string, with a single blank character inserted between adjacent values. The resulting sequence of nodes is called the *content sequence*. Adjacent text nodes in the content sequence are merged into a single text node by concatenating the contents of the text nodes with no intervening blanks. After concatenation, any text node that contains a zero-length string is deleted from the content sequence.
3. The nodes in the content sequence become the children of the new document node.

Example 1: Insert a constructed document into an XML column:

```
INSERT INTO T1 VALUES(123,
  (SELECT XMLDOCUMENT(XMLELEMENT(NAME "Emp",
    e.fname || ' ' || e.lname),
    XMLCOMMENT('This is just a simple example'))
   FROM EMPLOYEE e
   WHERE e.empid = 123));
```

XMLEMENT

The XMLEMENT function returns an XML value that is an XML element node.



Notes:

- 1 The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified
- 2 If *element-content-expression* is not specified, EMPTY ON NULL and NULL ON NULL must not be specified.
- 3 The same clause must not be specified more than one time.

The schema is SYSIBM.

NAME *element-name*

Specifies the name of an XML element. *element-name* is an SQL identifier that must be in the form of an XML qualified name, or QName. If the name is qualified, the namespace prefix must be declared within the scope.

xmlnamespace-declaration

Specifies the XML namespace declarations that are the result of the XMLNAMESPACES function. The namespaces that are declared are in the scope of the XMLEMENT function. The namespaces apply to any nested XML functions within the XMLEMENT function, regardless of whether or not they appear inside another subselect. See "XMLNAMESPACES" on page 587 for more information on declaring XML namespaces.

If *xmlnamespace-declaration* is not specified, namespace declarations are not associated with the constructed XML element node.

xmlattributes-function

Specifies the attributes for the XML element. The attributes are the result of the XMLATTRIBUTES function. See "XMLATTRIBUTES" on page 575 for more information on constructing attributes.

If *xmlattributes-function* is not specified, attributes are not explicitly part of the constructed XML element node.

element-content-expression

The content of the generated XML element node is specified by an expression or a list of expressions. Each *element-content-expression* must return a value of any built-in data type or distinct type. The expression is used to construct the namespace declarations, attributes, and content of the constructed element node.

If *element-content-expression* is not specified, an empty string is used as the content for the element and NULL ON NULL or EMPTY ON NULL must not be specified.

OPTION

Specifies additional options for constructing the XML element. This clause has no impact on nested invocations of the XMLELEMENT function invocations that are specified in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL

Specifies if a null value or an empty element is returned when the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The option is not inherited by a nested invocation of XMLELEMENT function within an *element-content-expression*.

EMPTY ON NULL

If the value of each *element-content-expression* is null, an empty element is returned.

EMPTY ON NULL is the default.

NULL ON NULL

If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX

Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, ROWID, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values.

XMLBINARY USING BASE64

Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+' and '/') to represent every 6 bits of the binary or bit data by one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING BASE64 is the default.

XMLBINARY USING HEX

Specifies that the assumed encoding is hexadecimal characters as defined for XML schema type xs:hexBinary encoding. The hex encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an element name, an optional collection of namespace declarations, an optional collection of attributes, and zero or more optional arguments that make up the content of the XML element. The result is an XML sequence that contains an XML element node or the null value. If the results of all *element-content-expression* arguments are empty strings, the result is an XML sequence that contains an empty element.

The result of the function is an XML value. The result can be null; if all *element-content-expression* arguments are null and the NULL ON NULL option is in effect, the result is the null value.

Constructing an element node: The resulting element node is constructed as follows:

1. *xmlnamespace-declaration* adds a set of in-scope namespaces for the constructed element. Each in-scope namespace associates a namespace prefix (or the default namespace) with a namespace URI. The in-scope namespaces define the set of namespace prefixes that are available for interpreting QNames within the scope of the element.
2. If the *xmlattributes-function* is specified, it is evaluated and the result is a sequence of attribute nodes.
3. Each *element-content-expression* is evaluated and the result is converted into a sequence of nodes as follows:
 - If the result type is not XML, it is converted to an XML text node that contains the result of the *element-content-expression* this is mapped to XML.
 - If the result type is XML, the result is a sequence of items. Some of the items in that sequence might be document nodes. Each document node in the sequence is replaced by the sequence of its top-level children. Then for each node in the resulting sequence, a new deep copy of the node is constructed, including its children and attributes. Each copied node has a new node identity. Copied element nodes are given the type annotation `xdt:untyped`, and copied attribute nodes are given the type annotation `xdt:untypedAtomic`. For each adjacent sequence of one or more atomic values that are returned in the sequence, a new text node is constructed that contains the result of casting each atomic value to a string, with a single blank character inserted between adjacent values. If any of these atomic values cannot be cast into a string, an error is returned.
4. The result sequence of *xmlattributes-function* and the resulting sequences of all *element-content-expression* clauses are concatenated into one sequence which is called the *content sequence*. Any sequence of adjacent text nodes in the content sequence is merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node that is a zero-length string is deleted from the content sequence.
5. If the content sequence contains an attribute node that follows a node that is not an attribute node, an error is returned. Attribute nodes that occur in the content sequence become attributes of the new element node. If two or more of these attribute nodes have the same name, an error is returned. A namespace declaration is created that corresponds to any namespace that is used in the names of the attribute nodes if the namespace URI is not in the in-scope namespaces of the constructed element.
6. Element, text, comment, and processing instruction nodes in the content sequence become the children of the constructed element node.

- The constructed element node is given a type annotation of `xdt:untyped`, and each of its attributes is given a type annotation of `xdt:untypedAtomic`. The node name of the constructed element node is the XML element name that is specified after the `NAME` keyword.

Rules for using namespaces within XMLELEMENT: The following rules describe scoping of namespaces:

- The namespaces that are declared in the `XMLNAMESPACES` function are the in-scope namespaces of the element node that are constructed by the `XMLELEMENT` function. If the element node is serialized, each of its in-scope namespaces will be serialized as a namespace attribute unless it is an in-scope namespace of the parent of the element node and the parent element is also serialized.
- The scope of these namespaces is the lexical scope of the `XMLELEMENT` function, including the element name, the attribute names that are specified in the `XMLATTRIBUTES` function, and all *element-content-expressions*. These are used to resolve the `QNames` in the scope.
- If an `XMLQUERY` or `XMLEXISTS` function is in an *element-content-expression*, the namespaces become the *statically known namespaces* of the XPath expression of the `XMLQUERY` or `XMLEXISTS` function. Statically known namespaces are used to resolve the `QNames` that are in the XPath expression. If the XPath prolog declares a namespace that has the same prefix within the scope of the XPath expression, the namespace that is declared in the prolog will override the namespaces that are declared in the `XMLNAMESPACES` function.
- If an attribute of the constructed element comes from *element-content-expression*, its namespace might not already be declared as an in-scope namespace of the constructed element. In this case, a new namespace is created for it. If the prefix of the attribute name is already bound to a different URI by a in-scope namespace, DB2 generates a different prefix to be used in the attribute name. A namespace is created for this generated prefix. The name of the generated prefix follows the following pattern: `db2ns-xx`, where `xx` is a pair of characters chosen from the set `[A-Z,a-z,0-9]`.

Example 1: The following statement uses the `XMLELEMENT` function to create an XML element that contains an employees name. The statement also stores the employee number as an attribute named `serial`. If there is a null value in the referenced column, the function returns the null value:

```
SELECT e.empno, e.firstnme, e.lastname,
       XMLELEMENT ( NAME "foo:Emp",
                    XMLNAMESPACES('http://www.foo.com' AS "foo"),
                    XMLATTRIBUTES(e.empno as "serial"),
                    e.firstnme,
                    e.lastname
                    OPTION NULL ON NULL ) AS "Result"
FROM EMP e
WHERE e.edlevel = 12;
```

The result of the query would look similar to the following result:

EMPNO	FIRSTNME	LASTNAME	Result
A0001	John	Parker	<foo:Emp xmlns:foo="http://www.foo.com" serial="A0001">JohnParker</foo:Emp>
B0001	(null)	Smith	<foo:Emp xmlns:foo="http://www.foo.com" serial="B0001">Smith</foo:Emp>
B0002	(null)	(null)	(null)
(null)	(null)	(null)	(null)

Example 2: The following example is similar to Example 1, however, when a null value is in one of the referenced columns, an empty element is returned:

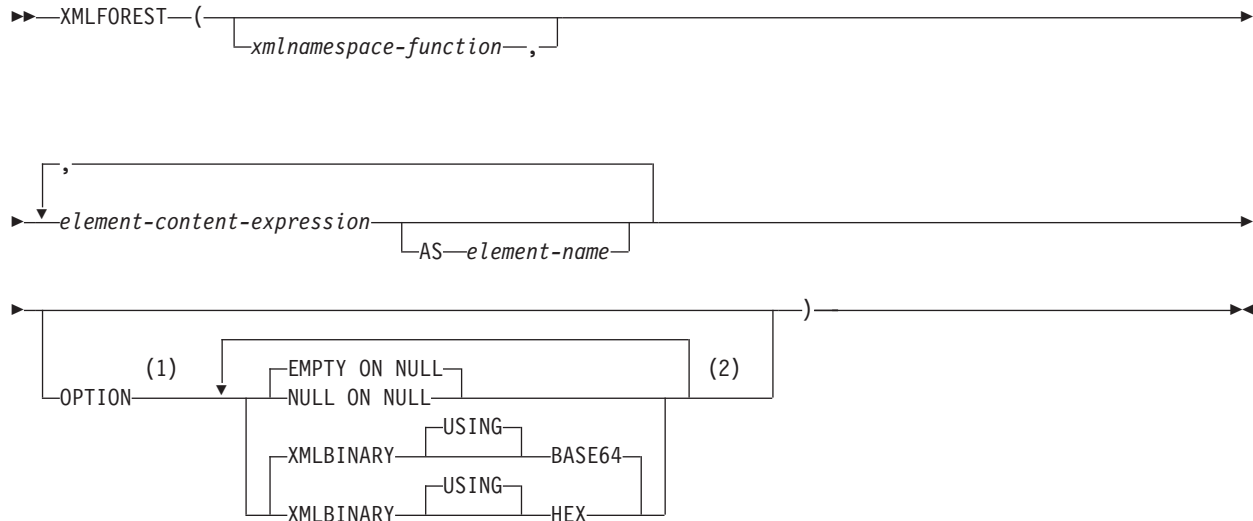
```
SELECT e.empno, e.firstnme, e.lastname,
       XMLELEMENT (NAME "foo:Emp",
                   XMLNAMESPACES('http://www.foo.com' AS "foo"),
                   XMLATTRIBUTES(e.empno as "serial"),
                   e.firstnme,
                   e.lastname
                   OPTION EMPTY ON NULL) AS "Result"
FROM EMP e
WHERE e.edlevel = 12;
```

The result of the query would look similar to the following result:

EMPNO	FIRSTNME	LASTNAME	Result
-----	-----	-----	-----
A0001	John	Parker	<foo:Emp xmlns:foo="http://www.foo.com" serial="A0001">JohnParker</foo:Emp>
B0001	(null)	Smith	<foo:Emp xmlns:foo="http://www.foo.com" serial="B0001">Smith</foo:Emp>
B0002	(null)	(null)	<foo:Emp xmlns:foo="http://www.foo.com" serial="B0002"></foo:Emp>
(null)	(null)	(null)	<foo:Emp></foo:Emp>

XMLFOREST

The XMLFOREST function returns an XML value that is a sequence of XML element nodes.



Notes:

- 1 The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified.
- 2 The same clause must not be specified more than one time.

The schema is SYSIBM.

xmlnamespace-function

Specifies the XML namespace declarations that are the result of the XMLNAMESPACES function. The namespaces that are declared are in the scope of the XMLFOREST function. The namespaces apply to any nested XML functions within the XMLFOREST function, regardless of whether or not those functions appear inside another subselect. See “XMLNAMESPACES” on page 587 for more information on declaring XML namespaces.

If *xmlnamespace-function* is not specified, namespace declarations are not associated with the constructed sequence of XML element nodes.

element-content-expression

Specifies an expression that returns a value that is used for the content of a generated XML element. The result of the expression is mapped to an XML value according to the mapping rules from an SQL value to an XML value. If the expression is not a simple column reference, *element-name* must be specified.

AS *element-name*

Specifies an identifier that is used for the XML element name.

An XML element name must be an XML QName. If the name is qualified, the namespace prefix must be declared within the scope.

If *element-name* is not specified, *element-content-expression* must be a column name. The element name is created from the column name using the fully escaped mapping from a column name to a QName.

OPTION

Specifies options for the result for NULL values, binary data, and bit data. The options will not be inherited by the XMLELEMENT or XMLFOREST functions that appear in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL

Specifies if a null value or an empty element is returned when the values of each *element-content-expression* is a null value. EMPTY ON NULL and NULL ON NULL only affect null handling of the *element-content-expression* arguments, not the handling of values from an *xmlattributes-function* argument.

EMPTY ON NULL

If the value of each *element-content-expression* is null, an empty element is returned.

EMPTY ON NULL is the default.

NULL ON NULL

If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX

Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, ROWID, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values.

XMLBINARY USING BASE64

Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+' and '/') to represent every 6 bits of the binary or bit data by one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING BASE64 is the default.

XMLBINARY USING HEX

Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hex encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

The XMLFOREST function can be expressed using the XMLCONCAT and XMLELEMENT functions.

This function takes an optional set of namespace declarations and one or more arguments that make up the name and element content for one or more element nodes. The result is an XML sequence containing a sequence of element nodes or the null value.

The result of the function is an XML value. The result can be null; if all the *element-content-expression* arguments are null and the NULL ON NULL option is in effect, the result is the null value.

Example: Generate an "Emp" element for each employee. Use employee name as its attribute and two subelements generated from columns HIRE and DEPT by using XMLFOREST as its content. The element names for the two subelements are "HIRE" and "department".

```
| SELECT e.id, XMLSERIALIZE ( XMLELEMENT
|   ( NAME "Emp",
|     XMLATTRIBUTES ( e.fname || ' ' || e.lname
|                     AS "name" ),
|     XMLFOREST ( e.hire,
|                 e.dept AS "department" )
|               ) ) AS "result"
| FROM employees e;
```

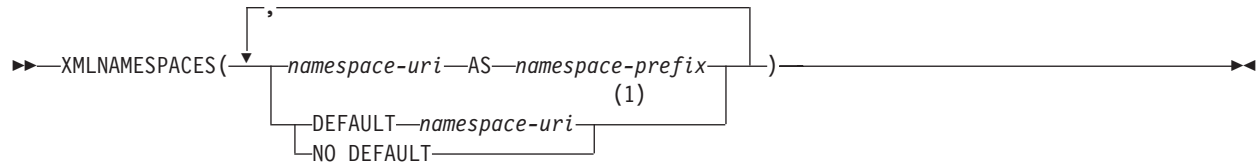
The result of the query would be similar to the following result:

ID	result

1001	<Emp name="John Smith"> <HIRE>2000-05-24</HIRE> <department>Accounting</department> </Emp>
1001	<Emp name="Mary Martin"> <HIRE>1996-02-01</HIRE> <department>Shipping</department> </Emp>

XMLNAMESPACES

The XMLNAMESPACES function constructs namespace declarations from the arguments. This function can be used as an argument only for specific functions, such as the XMLELEMENT function and the XMLFOREST function.



Notes:

- 1 The DEFAULT or NO DEFAULT clause can only be specified one time.

The schema is SYSIBM.

The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.

namespace-uri

Specifies an SQL character string constant that contains the namespace name or a universal resource identifier (URI). The character string constant must not be an empty string if it is used with *namespace-prefix*. *namespace-uri* cannot be `http://www.w3.org/XML/1998/namespace` or `http://www.w3.org/2000/xmlns/`.

AS *namespace-prefix*

Specifies a namespace prefix. The prefix is an SQL identifier that must be in the form of an XML NCName. The prefix must not be "xml" or "xmlns". The prefix must be unique within the list of namespace declarations.

The following namespace prefixes are pre-defined in SQL/XML: "xml", "xs", "xsd", "xsi", and "sqlxml". Their bindings are:

- `xmlns:xml = "http://www.w3.org/XML/1998/namespace"`
- `xmlns:xs = "http://www.w3.org/2001/XMLSchema"`
- `xmlns:xsd = "http://www.w3.org/2001/XMLSchema"`
- `xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"`
- `xmlns:sqlxml = "http://standards.iso.org/iso/9075/2003/sqlxml"`

DEFAULT *namespace-uri* **or NO DEFAULT**

Specifies whether a default namespace is to be used within the scope of this namespace declaration.

The scope of this namespace declaration is the specified XML element and all XML expressions that are contained in the specified XML element.

DEFAULT *namespace-uri*

Specifies the default namespace to use within the scope of this namespace declaration. The *namespace-uri* applies for unqualified names in the scope unless it is overridden in a nested scope by another DEFAULT declaration or by a NO DEFAULT declaration.

namespace-uri specifies an SQL character string constant that contains a namespace name or universal resource identifier (URI). The character string constant can be an empty string in the context of the DEFAULT clause.

NO DEFAULT

Specifies that no default namespace is to be used within the scope of this namespace declaration. There is no default namespace in the scope unless the NO DEFAULT clause is overridden in a nested scope by a DEFAULT declaration.

The result of the function is an XML value that is an XML sequence that contains an XML namespace declaration for each specified namespace. The result cannot be null.

Example 1: Generate an "employee" element for each employee. The employee element is associated with XML namespace "urn:bo", which is bound to prefix "bo". The element contains attributes for names and a hiredate subelement.

```
SELECT e.empno, XMLSERIALIZE(XMLELEMENT(NAME "bo:employee",
                                     XMLNAMESPACES('urn:bo' as "bo"),
                                     XMLATTRIBUTES(e.lastname, e.firstname),
                                     XMLELEMENT(NAME "bo:hiredate", e.hiredate)) AS CLOB(50))
FROM employee e where e.edlevel = 12;
```

The result of the query would be similar to the following result:

```
00029 <bo:employee xmlns:bo="urn:bo" LASTNAME="PARKER" FIRSTNME="JOHN">
      <bo:hiredate>198-5-3</bo:hiredate>
</bo:employee>
00031 <bo:employee xmlns:bo="urn:bo" LASTNAME="SETRIGHT"
      FIRSTNME="MAUDE">
      <bo:hiredate>1964-9-12</bo:hiredate>
</bo:employee>
```

Example 2: Generate two elements for each employee using XMLFOREST. The first "lastname" element is associated with the default namespace "http://hr.org", and the second "job" element is associated with XML namespace "http://fed.gov", which is bound to prefix "d".

```
SELECT empno, XMLSERIALIZE(XMLFOREST(
      XMLNAMESPACES(DEFAULT 'http://hr.org', 'http://fed.gov' AS "d"),
      lastname, job AS "d:job") AS CLOB(50))
FROM employee where edlevel = 12;
```

The result of the query would be similar to the following result:

```
00029 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER
</LASTNAME>
      <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">
      OPERATOR</d:job>
00031 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">
      SETRIGHT</LASTNAME>
      <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">
      OPERATOR</d:job>
```

1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
 12
 13
 14
 15
 16
 17
 18
 19
 20
 21
 22
 23
 24
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66
 67
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
 100
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
 111
 112
 113
 114
 115
 116
 117
 118
 119
 120
 121
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133
 134
 135
 136
 137
 138
 139
 140
 141
 142
 143
 144
 145
 146
 147
 148
 149
 150
 151
 152
 153
 154
 155
 156
 157
 158
 159
 160
 161
 162
 163
 164
 165
 166
 167
 168
 169
 170
 171
 172
 173
 174
 175
 176
 177
 178
 179
 180
 181
 182
 183
 184
 185
 186
 187
 188
 189
 190
 191
 192
 193
 194
 195
 196
 197
 198
 199
 200
 201
 202
 203
 204
 205
 206
 207
 208
 209
 210
 211
 212
 213
 214
 215
 216
 217
 218
 219
 220
 221
 222
 223
 224
 225
 226
 227
 228
 229
 230
 231
 232
 233
 234
 235
 236
 237
 238
 239
 240
 241
 242
 243
 244
 245
 246
 247
 248
 249
 250
 251
 252
 253
 254
 255
 256
 257
 258
 259
 260
 261
 262
 263
 264
 265
 266
 267
 268
 269
 270
 271
 272
 273
 274
 275
 276
 277
 278
 279
 280
 281
 282
 283
 284
 285
 286
 287
 288
 289
 290
 291
 292
 293
 294
 295
 296
 297
 298
 299
 300
 301
 302
 303
 304
 305
 306
 307
 308
 309
 310
 311
 312
 313
 314
 315
 316
 317
 318
 319
 320
 321
 322
 323
 324
 325
 326
 327
 328
 329
 330
 331
 332
 333
 334
 335
 336
 337
 338
 339
 340
 341
 342
 343
 344
 345
 346
 347
 348
 349
 350
 351
 352
 353
 354
 355
 356
 357
 358
 359
 360
 361
 362
 363
 364
 365
 366
 367
 368
 369
 370
 371
 372
 373
 374
 375
 376
 377
 378
 379
 380
 381
 382
 383
 384
 385
 386
 387
 388
 389
 390
 391
 392
 393
 394
 395
 396
 397
 398
 399
 400
 401
 402
 403
 404
 405
 406
 407
 408
 409
 410
 411
 412
 413
 414
 415
 416
 417
 418
 419
 420
 421
 422
 423
 424
 425
 426
 427
 428
 429
 430
 431
 432
 433
 434
 435
 436
 437
 438
 439
 440
 441
 442
 443
 444
 445
 446
 447
 448
 449
 450
 451
 452
 453
 454
 455
 456
 457
 458
 459
 460
 461
 462
 463
 464
 465
 466
 467
 468
 469
 470
 471
 472
 473
 474
 475
 476
 477
 478
 479
 480
 481
 482
 483
 484
 485
 486
 487
 488
 489
 490
 491
 492
 493
 494
 495
 496
 497
 498
 499
 500
 501
 502
 503
 504
 505
 506
 507
 508
 509
 510
 511
 512
 513
 514
 515
 516
 517
 518
 519
 520
 521
 522
 523
 524
 525

11



1

1

11

1

1

11

1

11

1

11

1

1
1
1

1

1

11

11

1
1
1
1
1
1

111

— — —

XMLPARSE will treat the value in *hv* for the insert statement as equivalent to the following value:

```
<a xml:space='preserve'> <b> <c>c</c>b </b>
</a>
```

Example 2: The following example inserts an XML document into the EMP table and strips the whitespace in the original XML document. Assume that *hv* contains the value, '<a xml:space='preserve'> <b xml:space='default'> <c>c</c>b ':

```
INSERT INTO EMP (id, xvalue) VALUES(1001,
                                     XMLPARSE(DOCUMENT :hv
                                                STRIP WHITESPACE));
```

XMLPARSE will treat the value in *hv* for the insert statement as equivalent to the following value:

```
<a xml:space='preserve'>
<b xml:space='default'><c>c</c>b </b>
</a>
```


XMLPI

The XMLPI function returns an XML value with a single processing instruction node.

```
XMLPI(—NAME—pi-name [, —string-expression—])
```

The schema is SYSIBM.

NAME *pi-name*

Specifies the name of a processing instruction. The name is an SQL identifier that must be in the form of an XML NCName. The name must not contain "xml" in any case combination.

string-expression

An expression that returns a value of a built-in character or graphic string that is not a LOB and is not bit data. The resulting string will be converted to UTF-8 and parsed to check for conformance to the content of XML processing instruction as specified by the following rules:

- The string must not contain the substring '?>' as this terminates a processing instruction.
- Each character can be any Unicode character, excluding the surrogate blocks, X'FFFE', and X'FFFF'.

If the resulting string does not conform to the preceding rules, an error is returned. The resulting string becomes the contents of the constructed processing instruction node. If *string-expression* is not specified or is an empty string, the contents of the processing instruction node are empty.

The result of the function is an XML value. The result can be null; if the *string-expression* argument is null, the result is the null value.

Example: Generate an XML processing instruction node:

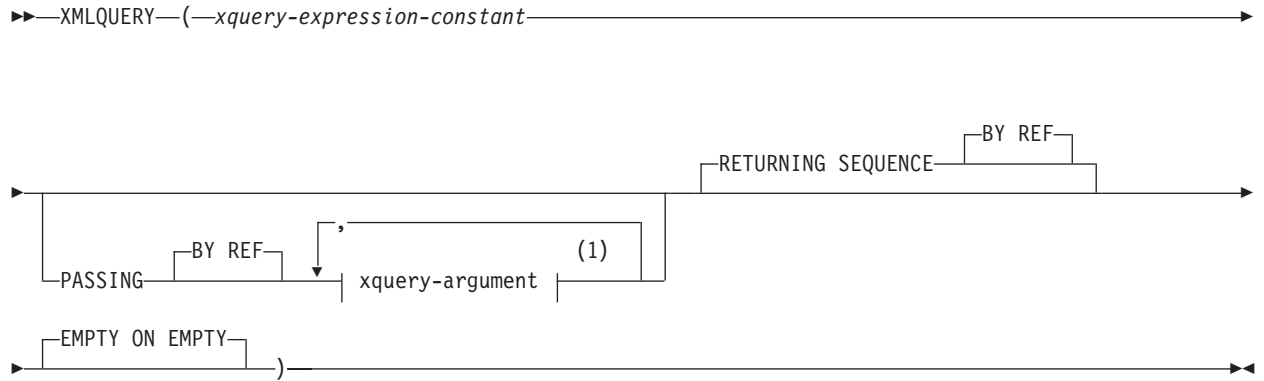
```
SELECT XMLPI(NAME "Instruction", 'Push the red button')
FROM SYSIBM.SYSDUMMY1;
```

The result looks similar to the following results:

```
<?Instruction Push the red button?>
```

XMLQUERY

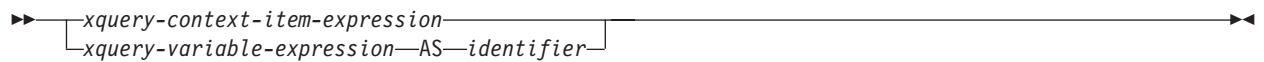
The XMLQUERY function returns an XML value from the evaluation of an XPath expression, by using specified input arguments, a context item, and XPath variables.



Notes:

- 1 *xquery-context-item-expression* must not be specified more than one time.

xquery-argument:



The schema is SYSIBM.

xquery-expression-constant

Specifies an SQL character string constant that is interpreted as an XPath expression using supported XPath language syntax. See *DB2 XML Guide* for information about the supported XPath expressions. The XPath expression is evaluated with the arguments specified in *xquery-argument*, and returns an output sequence that is also returned as the result of the XMLQUERY function. *xquery-expression-constant* must not be an empty string or a string of all blanks.

PASSING

Specifies input values and the manner in which these values are passed to the XPath expression that is specified by *xquery-expression-constant*.

BY REF

Specifies that the XML input value arguments are to be passed by reference. When XML values are passed by reference, the XPath evaluation uses the input node trees which preserves all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some

nodes that are contained between the two input arguments might refer to nodes that are within the same XML node tree.

BY REF has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

xquery-argument

Specifies an argument that is passed to the XPath expression that is specified by *xquery-expression-constant*. A query argument is an expression that returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-argument* must not return ROWID, TIMESTAMP, binary string, REAL, DECFLOAT data types, or a character string data type that is bit data, and must not reference a sequence expression.

xquery-argument specifies both a value and the manner in which that value is to be passed. How an argument in the PASSING clause is used in the XPath expression depends on whether the argument is specified as *xquery-context-item-expression* or *xquery-variable-expression*. *xquery-argument* includes an SQL expression that is evaluated before passing the result to the XPath expression.

- If the resulting value is of type XML, it becomes an *input-xml-value*. It is passed by reference, which means that the original values, not copies, are used in the evaluation of the XPath expression. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, the result of the expression must be able to be cast to an XML value. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When *xquery-expression-constant* is evaluated, an XPath variable receives a value that is equal to *input-xml-value* and a name as specified by the AS clause.

xquery-context-item-expression

xquery-context-item-expression specifies the initial context item in the XPath expression specified by *xquery-expression-constant*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-context-item-expression must not be a sequence of more than one item. If *input-xml-value* is an empty XML string, the XPath expression is evaluated with the initial context item set to an empty XML string. If the value of *input-xml-value* is null, the function returns a null value.

If the *xquery-context-item-expression* is not specified or is an empty sequence, the initial context item in the XPath expression is undefined and the XPath expression must not reference the initial context item.

An XPath variable is not created for the context item expression.

xquery-variable-expression

xquery-variable-expression specifies an SQL expression whose value is available to the XPath expression that is specified by *xquery-expression-constant* during execution. The sequence cannot contain a sequence reference.

An XPath variable is created for each *xquery-variable-expression*, and the XPath variable is set to a value equal to *input-xml-value*. For example, PASSING T.A + T.B AS "sum" creates an XPath variable named sum. The scope of the XPath variables that are created from the PASSING clause is the XPath expression that is specified by *xquery-expression-constant*.

AS identifier
 Specifies that the value that is generated by *xquery-variable-expression* is passed to *xquery-expression-constant* as an XPath variable named *identifier*. The length of the name must not be longer than 128 bytes. The leading dollar sign (\$) that precedes variable names in the XPath language is not included in *identifier*. The name must be an XML 1.0 NCName that is not the same as the identifier for another *xquery-variable-expression* in the same PASSING clause.

RETURNING SEQUENCE
 Specifies that the XPath expression returns a sequence.

BY REF
 Specifies that the result of the XPath expression is returned by reference. If this value contains nodes, any expression that is using the return value of the XPath expression will receive node references directly, preserving all node properties including the original node identities and document order.

EMPTY ON EMPTY
 Specifies that an empty sequence that results from processing the XPath expression is returned as an empty sequence.

The result of the function is an XML value. The result cannot be null.

If the evaluation of the XPath expression results in an error, the XMLQUERY function returns the XQuery error.

Implicit casting of a non XML value to an XML value: If the result of *xquery-argument* is not an XML type, the value is cast to XML as follows. The SQL data type of the expression is mapped to a corresponding XML Schema data type according to the following table:

Table 67. SQL data types and corresponding XML schema data types

SQL data type	XML schema data type
CHAR, VARCHAR	xs:string
GRAPHIC, VARGRAPHIC	xs:string
SMALLINT	xs:integer
INTEGER	xs:integer
BIGINT	xs:integer
DECIMAL	xs:decimal
DOUBLE	xs:double
FLOAT	xs:double

Let *V* be the value of the expression. An atomic value of the corresponding XML schema data type is constructed such that the result of cast (*V* as varchar) is a lexical representation of the constructed atomic value. For example, an SQL VARCHAR value '123' is converted to an atomic value '123' of xs:string type. An SQL integer '12' is converted to an atomic value '12' of xs:integer. An SQL decimal value '1.20' is converted to an atomic value '1.2' of xs:decimal.

Example 1: The following example returns an XML value from evaluation of the specified XPath expression:

```

|      SELECT XMLQUERY('///item[productName=$n]'
|                      PASSING PO.POrder,
|                      :hv AS "n") AS "Result"
|      FROM PurchaseOrders PO;

```

| Assume that the value of the host variable *hv* is 'Baby Monitor', the result is similar
 | to the following results:

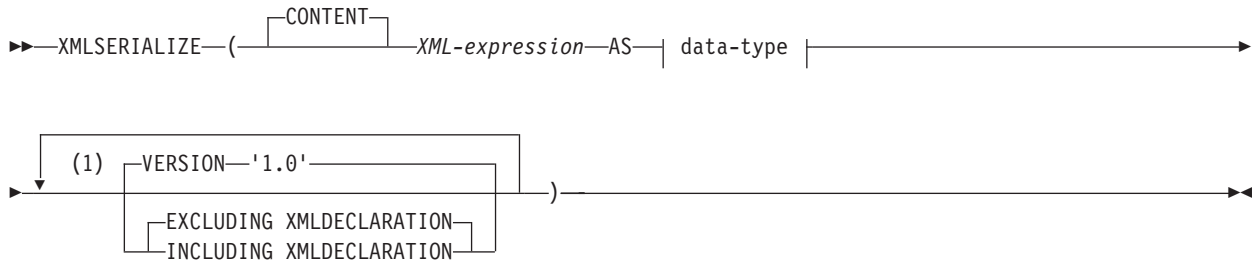
```

| Result
| -----
|
| <item partNum="926-AA"><productName>Baby Monitor</productName><quantity>1
| </quantity><USPrice>39.98</USPrice><shipDate>1999-05-21</shipDate></item>
|

```

XMLSERIALIZE

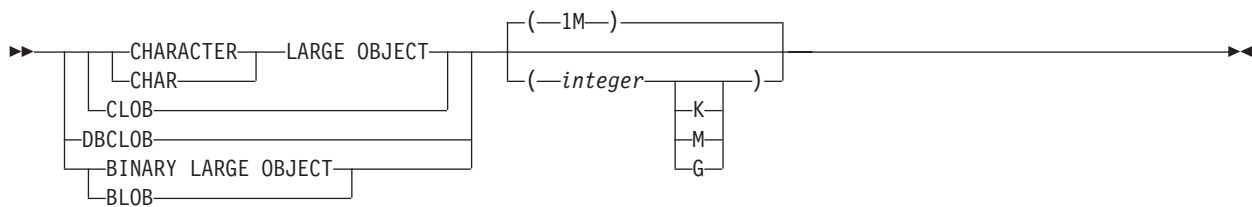
The XMLSERIALIZE function returns a serialized XML value of the specified data type that is generated from the first argument.



Notes:

- 1 The same clause must not be specified more than one time.

data-type



The schema is SYSIBM.

CONTENT

Specifies that any XML value can be specified and the result of the serialization is based on this input value.

XML-expression

An expression that returns an XML value that is not an attribute node. The atomic values in the input sequence must be able to be cast to xs:string. *XML-expression* is the input to the serialization process.

AS data type

Specifies the data type of the result. The implicit or explicit length attribute for the specified result data type must be sufficient to contain the serialized output.

The CCSID of a resulting character or graphic string is determined by the data type of the result:

- If the result is a CLOB, the CCSID for mixed Unicode data (1208).
- If the result is a DBCLOB, the CCSID for graphic Unicode data (1200).

VERSION '1.0'

Specifies the XML version of the serialized value. The only version that is supported is '1.0', which must be specified as a string constant.

EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION

Specifies whether an XML declaration is included in the result.

EXCLUDING XMLDECLARATION

Specifies that an XML declaration is not included in the result.

EXCLUDING XMLDECLARATION is the default.

INCLUDING XMLDECLARATION

Specifies that an XML declaration is included in the result. The XML declaration contains values for XML serialization version 1.0 and an encoding specification of UTF-8. An XML sequence is effectively converted to have a single document node by applying the XMLDOCUMENT function to *XML-expression* prior to serializing the resulting XML nodes.

The data type and length attribute of the result are determined from the specified *data-type*. The result can be null; if the *XML-expression* argument is null, the result is the null value.

Serializing a sequence: The value of the input argument to XMLSERIALIZE is a sequence. Before a sequence is serialized, it is normalized. The purpose of sequence normalization is to create a sequence that can be serialized as a well-formed XML document or external general parsed entity, that also reflects the content of the input sequence to the extent possible. If the input sequence is an XML empty string, the result of serialization is an empty string. Otherwise, the result is constructed as follows:

- For each item in the sequence, if the item is atomic, the lexical representation of the item is obtained by casting it to an xs:string
- Each subsequence of adjacent strings in the sequence is merged into a single string with the values of the adjacent strings separated by a single space.
- For each item in the sequence, if the item is a string, a text node is created with a value that is equal to the string.
- For each node in the sequence, if the node is a document node, it is replaced it by its children.
- Each node must not be an attribute node.
- Each subsequence of adjacent text nodes in the sequence are merged into a single text node that with the values of the adjacent text nodes concatenated in order without a space between each node. Any text nodes of zero length are dropped.
- A document node is created and the sequence of nodes that was generated is copied as the children of the new document node.

Let *S* be any sequence, the normalization described in the preceding list is equivalent to XMLDOCUMENT(*S*). Therefore, the following two expressions produce the same result:

- XMLSERIALIZE(*S* AS CLOB)
- XMLSERIALIZE(XMLDOCUMENT(*S*) AS CLOB)

Each instance of the following characters that appear in the content of a text node or in the value of an attribute node is mapped as following during serialization:

Character in content of text node	during serialization, the character is mapped to
'&' (X'26')	'&'
'<' (X'3C')	'<'
'>' (X'3E')	'>'
carriage return (X'0D')	''
quote (X'22') ¹	'"'
Note: The quote character is only mapped if it is inside of an attribute value.	

Syntax alternatives: XML2CLOB(*XML-expression*) can be specified as an alternative to XMLSERIALIZE(*XML-expression* AS CLOB(2G)). XML2CLOB is supported only for compatibility with previous releases of DB2.

Example 1: Serialize into CLOB of UTF-8, the XML value that is returned by the XMLELEMENT function, which is a simple XML element with "Emp" as the element name, and an employee name as the element content:

```
SELECT e.id, XMLSERIALIZE(XMLELEMENT ( NAME "Emp",
                                     e.fname || ' ' || e.lname)
                        AS CLOB(100)) AS "result"
FROM employees e;
```

The result looks similar to the following results:

```
ID      result
---  -----
1001  <Emp>John Smith</Emp>
1206  <Emp>Mary Martin</Emp>
```

Example 2: Serialize into a string of BLOB type, the XML value that is returned by the XMLELEMENT function:

```
SELECT XMLSERIALIZE(XMLELEMENT(NAME "emp",
                               e.fname || ' ' || e.lname))
      AS BLOB(1K)
      VERSION '1.0') AS result
FROM employee e WHERE e.id = '1001';
```

The result looks similar to the following results:

```
result
-----
<emp>John Smith</emp>
```


XMLTEXT

The XMLTEXT function returns an XML value with a single text node that contains the value of the argument.

►►—XMLTEXT—(—*string-expression*—)————►►

The schema is SYSIBM.

string-expression

An expression that returns a value of a built-in character or graphic string that is not bit data. Any character in the resulting string must be a valid XML 1.0 character when it is converted to UTF-8.

If *string-expression* is an empty string, an empty text node is returned.

The result of the function is an XML value. The result can be null; if the argument is null, the result is the null value.

Example 1: The following example returns an XML value with a single text node that contains the specified value:

```
SELECT XMLTEXT('The stock symbol for Johnson&Johnson is JNJ.') AS "Result"
FROM SYSIBM.SYSDUMMY1;
```

The result looks similar to the following results:

Result

The stock symbol for Johnson&Johnson is JNJ.

Example 2: The XMLTEXT function enables the XMLAGG function to construct mixed content, as in the following example:

```
SELECT XMLELEMENT(NAME "para",
  XMLAGG(XMLCONCAT( XMLTEXT( plaintext),
    XMLELEMENT( NAME "emphasis",
      emphtext ))
  ORDER BY seqno ), '.' ) as "result"
FROM T;
```

Suppose that the content of the table T is as the following:

seqno	plaintext	emphtext
1	This query shows how to construct	mixed content
2	using XMLAGG and XMLTEXT. Without	XMLTEXT
3	XMLAGG cannot group text nodes with other nodes, therefore, cannot generate	mixed content

The result looks like the following result:

result

<para>This query shows how to construct <emphasis>mixed content</emphasis>
using XMLAGG and XMLTEXT. Without <emphasis>XMLTEXT</emphasis>, XMLAGG
cannot group text nodes with other nodes, therefore, cannot generate
<emphasis>mixed content</emphasis>.</para>

YEAR

The YEAR function returns the year part of a value that is a character or graphic string. The value must be a valid string representation of a date or timestamp.

►►—YEAR(*expression*)—◄◄

The schema is SYSIBM.

The argument must be an expression that returns one of the following built-in data types: a date, a timestamp, a character string, a graphic string, or a numeric data type.

- If *expression* is a character or graphic string, it must not be a CLOB or DBCLOB, and its value must be a valid string representation of a date or timestamp with an actual length of not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” on page 89.
- If *expression* is a number, it must be a date or timestamp duration. For the valid formats of date and timestamp durations, see “Datetime operands” on page 121.

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, a timestamp, or a string representation of either, the result is the year part of the value, which is an integer between 1 and 9999.

If the argument is a date duration or a timestamp duration, the result is the year part of the value, which is an integer between -9999 and 9999. A nonzero result has the same sign as the argument.

Example 1: From the table DSN8910.EMP, select all rows for employees who were born in 1941.

```
SELECT *  
  FROM DSN8910.EMP  
 WHERE YEAR(BIRTHDATE) = 1941;
```

Table functions

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

ADMIN_TASK_LIST

The ADMIN_TASK_LIST function returns a table with one row for each of the tasks that are defined in the administrative scheduler task list.

►►—ADMIN_TASK_LIST()—◄◄

The schema is DSNADM.

The result of the function is a table with the format shown in the following table. All the columns are nullable except TASK_NAME.

Table 68. Format of the resulting table for ADMIN_TASK_LIST

Column name	Data type	Contains
BEGIN_ TIMESTAMP	TIMESTAMP	<p>Contains the timestamp of when the task can first run. When the task begins to run depends on what values this and other columns contain:</p> <ul style="list-style-type: none">• If BEGIN_TIMESTAMP contains a non-null value:<ul style="list-style-type: none">– If POINT_IN_TIME and TRIGGER_TASK_NAME contain null values, the task begins to run at the timestamp in BEGIN_TIMESTAMP– If POINT_IN_TIME contains a non-null value, the task begins to run at the next point in time that is defined at or after the timestamp in BEGIN_TIMESTAMP– If TRIGGER_TASK_NAME is a non-null value, the task begins to run at the next time that the task identified in TRIGGER_TASK_NAME completes or after the timestamp in BEGIN_TIMESTAMP• If BEGIN_TIMESTAMP contains a null value:<ul style="list-style-type: none">– If POINT_IN_TIME and TRIGGER_TASK_NAME contain null values, the task begins to run immediately– If POINT_IN_TIME contains a non-null value, the task begins to run at the next point in time that is defined– If TRIGGER_TASK_NAME is a non-null value, the task begins to run at the next time that the task identified in TRIGGER_TASK_NAME completes
END_ TIMESTAMP	TIMESTAMP	<p>Contains the timestamp of when the task is last able to run. If this column is NULL, there are no restrictions as to when the task must not run.</p>

Table 68. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
MAX_ INVOCATIONS	INTEGER	<p>Contains the maximum number of times the task can run. The maximum number applies to all types of schedules: triggered by events, scheduled by time interval, or by point in time. If this column is null, the task has no limit on the number of times it can be run.</p> <p>If both END_TIMESTAMP and MAX_INVOCATIONS contain values, the value in END_TIMESTAMP takes precedence over the value for MAX_INVOCATIONS. That is, if the value in END_TIMESTAMP is reached, even though the number of times the task has run has not reached the value for MAX_INVOCATIONS, the task will not run again</p>
INTERVAL	INTEGER	<p>Contains an integer that indicates the duration between the start of one instance of a task and the start of the next instance of the same task. If the value of this column is NULL, the task is not scheduled to run at a regular interval.</p>
POINT_IN_ TIME	VARCHAR(400)	<p>Contains one or more points in time (in UNIX cron format) for which the task is scheduled to run. If the value of this column is NULL, the task is not scheduled to run at a specific point in time.</p> <p>The format contains the following pieces of information separated by blanks: given hour, given minute, given day of the week, given day of the month, given month of the year.</p>
TRIGGER_ TASK_NAME	VARCHAR(128)	<p>Contains the task name of the task that, when its execution is complete, will trigger the running of the task that is described in the row.</p> <p>Task name DB2STOP is reserved for DB2 stop events and task name DB2START is reserved for DB2 start events. Those events are handled by the administrative scheduler that is associated with the DB2 subsystem that is starting or stopping.</p> <p>If the value of this column is NULL, the task that is described in this row will not be triggered to run by another task.</p>

Table 68. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
TRIGGER_TASK_COND	CHAR(2)	<p>Contains the type of comparison that is to be made to the return code after the running of task that is indicated in TRIGGER_TASK_NAME. The following values are possible:</p> <p>GT Greater than</p> <p>GE Greater than or equal to</p> <p>EQ Equal to</p> <p>LT Less than</p> <p>LE Less tan or equal to</p> <p>NE Not equal to</p> <p>If this column contains NULL, the task is triggered to run without consideration of the return code of the task that is indicated in TRIGGER_TASK_NAME.</p>
TRIGGER_TASK_CODE	INTEGER	<p>Contains the return code from running the task indicated in TRIGGER_TASK_NAME.</p> <p>If the running of this task is triggered by a stored procedure, TRIGGER_TASK_CODE contains the SQLCODE that must be returned by the stored procedure in order for this task to run.</p> <p>If the running of this task is triggered by a JCL job, TRIGGER_TASK_CODE contains the MAXRC that must be returned by the job in order for this task to run.</p> <p>“ADMIN_TASK_STATUS” on page 606 returns the SQLCODE or MAXRC value in the SQLCODE or MAXRC column.</p> <p>If TRIGGER_TASK_COND is NULL, this column will also be NULL.</p>

Table 68. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
DB2_SSID	VARCHAR(4)	<p>Contains the DB2 subsystem ID of the DB2 subsystem that is associated with the administrative scheduler that should run this task.</p> <p>The value in this column is used in a data sharing environment where, for example different DB2 members have different configurations and running the task relies on a certain environment. A value in DB2_SSID will prevent an administrative scheduler of other members to run this task, so that the task can only be run as long as the administrative scheduler of the subsystem indicated in DB2_SSID is running.</p> <p>For a task that is being triggered by a DB2 start or DB2 stop event as indicated in the TRIGGER_TASK_NAME column, a value in DB2_SSID will allow the task to be run only when the indicated subsystem is starting or stopping. If no value is indicated in DB2_SSID, each subsystem that starts or stops will trigger a the task to be run locally, provided that the triggered task is run serially.</p> <p>If this column is NULL, any administrative scheduler can run this task.</p>
PROCEDURE_SCHEMA	VARCHAR(128)	Contains the schema of the DB2 stored procedure that this task will run. If the value of this column is null, DB2 uses a default schema.
PROCEDURE_NAME	VARCHAR(128)	Contains the name of the DB2 stored procedure that this task will run. If the value of this column is NULL, no stored procedure will be called when this task is run.
PROCEDURE_INPUT	VARCHAR(4096)	Contains a statement that returns one row of data. The returned value will be used as the input parameter of the stored procedure that this task will run. If this column contains the null value, no parameters are passed to the stored procedure when this task is run.
JCL_LIBRARY	VARCHAR(44)	Contains the name of the data set that contains the JCL job that is run when this task is run. If the value of this column is the null value, no JCL job will be run when this task is run.
JCL_MEMBER	VARCHAR(8)	Contains the name of the library member that contains the JCL job that is run when this task is run. If the value of this column is the null value, the data set that is specified in JCL_LIBRARY is sequential and contains the JCL job that is run when this task is run.

Table 68. Format of the resulting table for ADMIN_TASK_LIST (continued)

Column name	Data type	Contains
JOB_WAIT	VARCHAR(8)	Contains one of the following values, which indicates whether the JCL job can be run synchronously. If the value in the column is not null, this column contains one of the following values: NO Runs asynchronously YES Runs synchronously PURGE Runs synchronously and then the job status in z/OS is purged
TASK_NAME	VARCHAR(128)	Contains the unique name that is assigned to this task.
DESCRIPTION	VARCHAR(128)	Contains a description of the task if one exists.
USERID	VARCHAR(128)	Contains the authorization ID of the user under which the task will be invoked. If this column is NULL, the task is invoked by the default authorization ID that is associated with the administrative scheduler.
CREATOR	VARCHAR(128)	Contains the authorization ID that added the task to the administrative scheduler task list.
LAST_MODIFIED	TIMESTAMP	Timestamp of when the task was added or last modified.

Example 1: Retrieve information about all of the tasks that are defined in the administrative scheduler task list:

```
SELECT *
FROM TABLE (DSNADM.ADMIN_TASK_LIST()) AS T;
```

ADMIN_TASK_STATUS

The ADMIN_TASK_STATUS function returns a table with one row for each task that is defined in the administrative scheduler task list. Each row indicates the status of the task for the last time it was run.

»—ADMIN_TASK_STATUS()—«

The schema is DSNADM.

The result of the function is a table with the format shown in the following table.

Table 69. Format of the resulting table for ADMIN_TASK_STATUS

Column name	Data type	Contains
TASK_NAME	VARCHAR(128)	Contains the name of the task that has run, is running, or has been bypassed.
STATUS	VARCHAR(10)	Contains one of the following values that indicates task status: RUNNING The task is currently running COMPLETED The task has finished running. For asynchronous tasks (JCL jobs), this column contains COMPLETED whenever the job is submitted to be run. Otherwise, this column contains COMPLETED only after the task has finished running. NOTRUN The task was not run at the scheduled invocation time. The MSG column contains the error or warning message that indicates why the task was not run. UNKNOWN The scheduler shut down while the task was running. The scheduler is started again but cannot know the execution status of this interrupted task.
NUM_ INVOCATIONS	INTEGER	Contains the number of times the administrative scheduler attempted to run the task, including the current time if the task is currently running. The values in this column does not indicate if the task was successfully run.
START_ TIMESTAMP	TIMESTAMP	Contains the time when the task started running if the STATUS column contains COMPLETED, RUNNING, or UNKNOWN. Otherwise, this column contains the time that the task should have started to run but could not.
END_ TIMESTAMP	TIMESTAMP	Contains the time when the task finished running.

Table 69. Format of the resulting table for ADMIN_TASK_STATUS (continued)

Column name	Data type	Contains																		
JOB_ID	CHAR(8)	Contains the job ID that is assigned to the JCL job submitted by the administrative scheduler. This column contains NULL if the task is a stored procedure or if the STATUS column does not contain COMPLETED.																		
MAXRC	INTEGER	<p>Contains the highest return code from submitting a JCL job. If the task is synchronous, the value in this column is changed to the return code that is returned when the job finishes running.</p> <p>This column is set to NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if a synchronous task is finished and has run with JES3 in a z/OS 1.7 or earlier system.</p>																		
COMPLETION_ TYPE	INTEGER	<p>Contains one of the following values that indicates the completion type of the JCL job submitted by the administrative scheduler:</p> <table><tr><td>0</td><td>No completion information</td></tr><tr><td>1</td><td>Job ended normally</td></tr><tr><td>2</td><td>Job ended by completion code</td></tr><tr><td>3</td><td>Job had a JCL error</td></tr><tr><td>4</td><td>Job was canceled</td></tr><tr><td>5</td><td>Job abended</td></tr><tr><td>6</td><td>Converter abended while processing the job</td></tr><tr><td>7</td><td>Job failed security checks</td></tr><tr><td>8</td><td>Job failed in end-of-memory</td></tr></table> <p>This column contains NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if the JCL job is run with JES3 in a z/OS 1.7 or earlier system.</p>	0	No completion information	1	Job ended normally	2	Job ended by completion code	3	Job had a JCL error	4	Job was canceled	5	Job abended	6	Converter abended while processing the job	7	Job failed security checks	8	Job failed in end-of-memory
0	No completion information																			
1	Job ended normally																			
2	Job ended by completion code																			
3	Job had a JCL error																			
4	Job was canceled																			
5	Job abended																			
6	Converter abended while processing the job																			
7	Job failed security checks																			
8	Job failed in end-of-memory																			
SYSTEM_ ABENDCD	INTEGER	<p>Contains the system abend code returned by a failed JCL job that was submitted by the administrative scheduler.</p> <p>This column contains NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if the JCL job is run with JES3 in a z/OS 1.7 or earlier system.</p>																		
USER_ABENDCD	INTEGER	<p>Contains the user abend code returned by a failed JCL job that was submitted by the administrative scheduler.</p> <p>This column contains NULL if the task is a stored procedure, if the STATUS column does not contain COMPLETED, or if the JCL job is run with JES3 in a z/OS 1.7 or earlier system.</p>																		
MSG	VARCHAR(128)	Contains the error or warning message from the last time the task was run.																		

Table 69. Format of the resulting table for ADMIN_TASK_STATUS (continued)

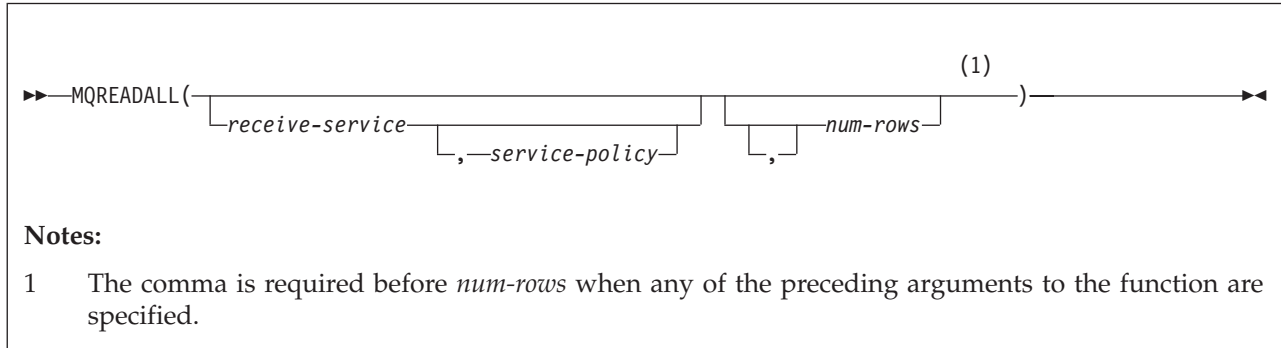
Column name	Data type	Contains
SQLCODE	INTEGER	Contains the SQLCODE set by DB2 when a stored procedure was called by the administrative scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.
SQLSTATE	CHAR(5)	Contains the SQLSTATE set by DB2 when a stored procedure was called by the administrative scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.
SQLERRP	VARCHAR(8)	Contains the SQLERRP set by DB2 when a stored procedure was called by the administrative scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.
SQLERRMC	VARCHAR(70)	Contains the SQLERRMC set by DB2 when a stored procedure was called by the administrative scheduler. This column contains NULL if the task is a JCL job or if the STATUS column does not contain COMPLETED.
DB2_SSID	VARCHAR(4)	Contains the DB2 subsystem ID that is associated with the administrative scheduler that ran the task or should have run the task.
USERID	VARCHAR(128)	Contain the user ID that the task ran under.

Example 1: Retrieve status information about all of the tasks that have run in the administrative scheduler task list:

```
SELECT *
FROM TABLE (DSNADM.ADMIN_TASK_STATUS()) AS T;
```

MQREADALL

The MQREADALL function returns a table that contains the messages and message metadata from a specified MQSeries location without removing the messages from the queue.



The schema is DB2MQ1N or DB2MQ2N.

The MQREADALL function returns a table containing the messages and message meta-data from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the messages from the queue that is associated with *receive-service*.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is read. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

num-rows

An expression that specifies the maximum number of messages to return. It must be an expression that returns a built-in integer data type.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns are nullable.

Table 70. Format of the resulting table for MQREADALL

Column name	Data type	Contains
MSG	VARCHAR(4000)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *  
  FROM TABLE (MQREADALL()) T;
```

The messages and all the metadata are returned as a table.

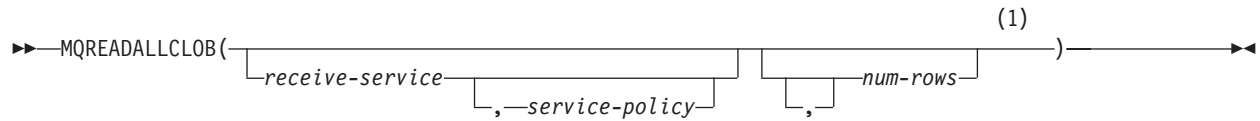
Example 2: Retrieve the first 10 messages from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *  
  FROM TABLE (MQREADALL(10)) T;
```

The first 10 messages and all the columns are returned as a table.

MQREADALLCLOB

The MQREADALLCLOB function returns a table that contains the messages and message metadata from a specified MQSeries location without removing the messages from the queue.



Notes:

- 1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The schema is DB2MQ1N or DB2MQ2N.

The MQREADALLCLOB function returns a table containing the messages and message meta-data from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the messages from the queue that is associated with *receive-service*.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is read. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

num-rows

An expression that specifies the maximum number of messages to return. It must be an expression that returns a built-in integer data type.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns in the table are nullable.

Table 71. Format of the resulting table for MQREADALLCLOB

Column name	Data type	Contains
MSG	CLOB(1M)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *  
FROM TABLE (MQREADALLCLOB()) T;
```

The messages and all the metadata are returned as a table.

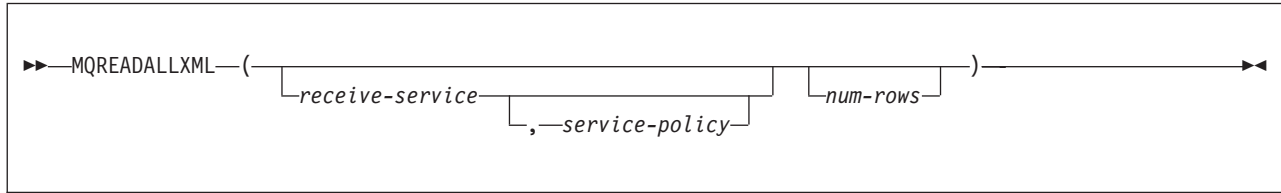
Example 2: Retrieve all the messages from the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY).

```
SELECT T.MSG, T.CORRELID  
FROM TABLE (MQREADALLCLOB('MYSERVICE')) T;
```

Only the MSG and CORRELID columns are returned as a table.

MQREADALLXML

The MQREADALLXML function returns a table that contains the messages and message metadata from a specified MQSeries location without removing the messages from the queue.



The schema is DMQXML1C or DMQXML2C.

The MQREADALLXML function returns a table containing the messages and message meta-data from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation does not remove the messages from the queue that is associated with *receive-service*.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is read. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

num-rows

An expression that specifies the maximum number of messages to return. It must be an integer that is greater than or equal to zero.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All the columns in the table are nullable.

Table 72. Format of the resulting table for MQREADALLXML

Column name	Data type	Contains
MSG	DB2XML.XMLVARCHAR	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), and read using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
FROM TABLE (MQREADALLXML()) T;
```

The messages and all the metadata are returned as a table.

Example 2: Retrieve all the messages from the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY). Have only the MSG and CORRELID columns of the table returned.

```
SELECT T.MSG, T.CORRELID
FROM table (MQREADALLXML('MYSERVICE')) T;
```

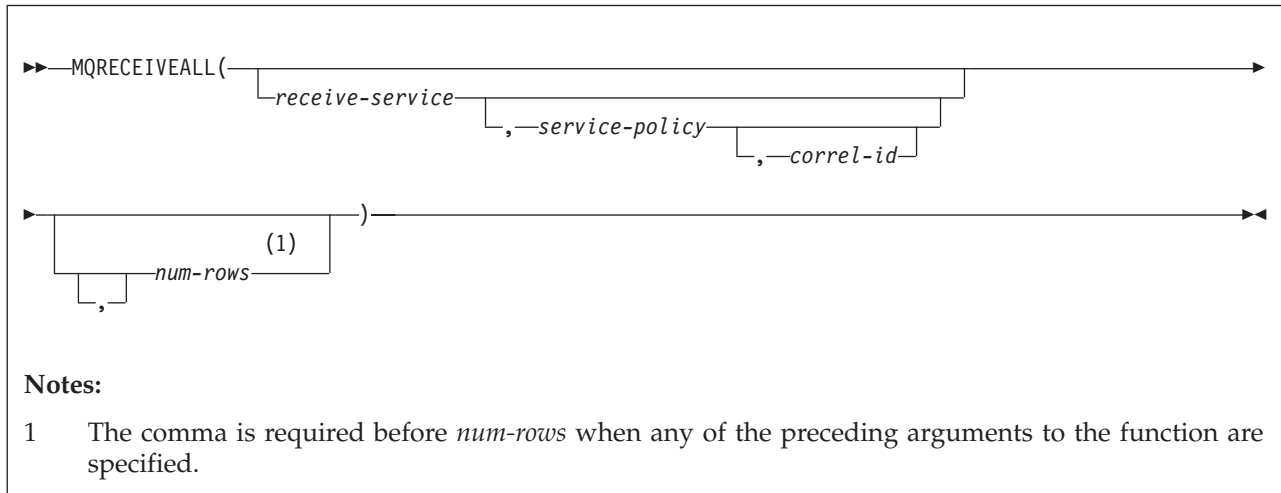
Example 3: Retrieve the first 10 messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
FROM TABLE (MQREADALLXML(10)) T;
```

The first 10 messages and all the columns are returned as a table.

MQRECEIVEALL

The MQRECEIVEALL function returns a table that contains the messages and message metadata from a specified MQSeries location and removes the messages from the queue.



The schema is DB2MQ1N or DB2MQ2N.

The MQRECEIVEALL function returns a table containing the messages and message meta-data from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the messages from the queue that is associated with *receive-service*.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is read. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. Only those messages with a matching correlation identifier are returned.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEALL does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, a correlation identifier is not used, and the message at the beginning of the queue is returned.

num-rows

An expression that specifies the maximum number of messages to return. It must be an expression that returns a built-in integer data type.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All of the columns are nullable.

Table 73. Format of resulting table for MQRECEIVEALL

Column name	Data type	Contains
MSG	VARCHAR(4000)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *
FROM TABLE (MQRECEIVEALL()) T;
```

The messages and all the metadata are returned as a table and deleted from the queue.

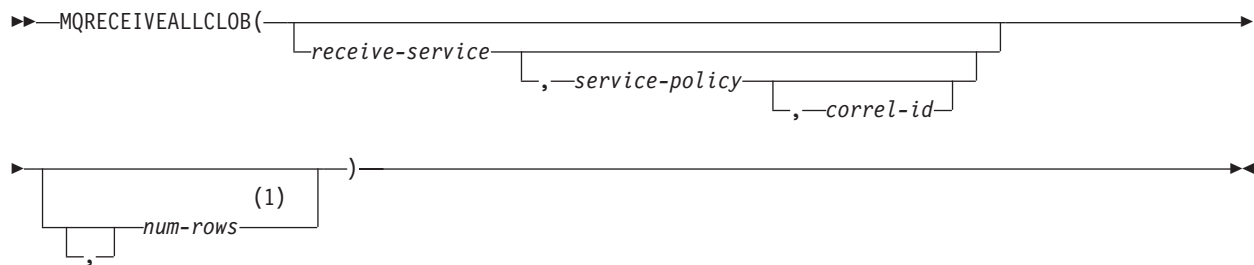
Example 2: Retrieve all the messages from the beginning of the queue specified by the service MYSERVICE, using the policy MYPOLICY.

```
SELECT T.MSG, T.CORRELID  
FROM TABLE (MQRECEIVEALL('MYSERVICE','MYPOLICY','1234')) T;
```

Only messages with CORRELID of '1234' and only the MSG and CORRELID columns are returned as a table and removed from the queue.

MQRECEIVEALLCLOB

The MQRECEIVEALLCLOB function returns a table that contains the messages and message metadata from a specified MQSeries location and removes the messages from the queue.



Notes:

- 1 The comma is required before *num-rows* when any of the preceding arguments to the function are specified.

The schema is DB2MQ1N or DB2MQ2N.

The MQRECEIVEALLCLOB function returns a table containing the messages and message metadata from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the messages from the queue that is associated with *receive-service*.

receive-service

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to a service point that is the logical MQSeries destination from which the message is read. A service point is defined in the DSNAMT repository file, and it represents a logical end-point from which a message is sent or received. A service point definition includes the name of the MQSeries queue manager and the name of the queue. See *MQSeries Application Messaging Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The value of the expression must not be the null value, an empty string, or a string with trailing blanks. The expression must have an actual length that is no greater than 48 bytes. The value of the expression refers to an MQSeries AMI service policy that is used in handling this message. A service policy is defined in the DSNAMT repository file, and it specifies a set of quality-of-service options that are to be applied to this messaging operation. These options include message priority and message persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

correl-id

An expression that returns a value that is a built-in character string or graphic string data type that is not a LOB. The expression must have an actual length that is no greater than 24 bytes. The value of the expression specifies the correlation identifier that is associated with this message. A correlation identifier is often specified in request-and-reply scenarios to associate requests with replies. Only those messages with a matching correlation identifier are returned.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQSEND, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEALLCLOB does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSEND request.

If *correl-id* is not specified, a correlation identifier is not used, and the message at the beginning of the queue is returned.

num-rows

An expression that specifies the maximum number of messages to return. It must be an expression that returns a built-in integer data type.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All of the columns are nullable.

Table 74. Format of resulting table for MQRECEIVEALLCLOB

Column name	Data type	Contains
MSG	CLOB(1M)	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

The CCSID of the result is the system CCSID that was in effect at the time that the MQSeries function was installed into DB2.

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *  
FROM TABLE (MQRECEIVEALLCLOB()) T;
```

The messages and all the metadata are returned as a table and deleted from the queue.

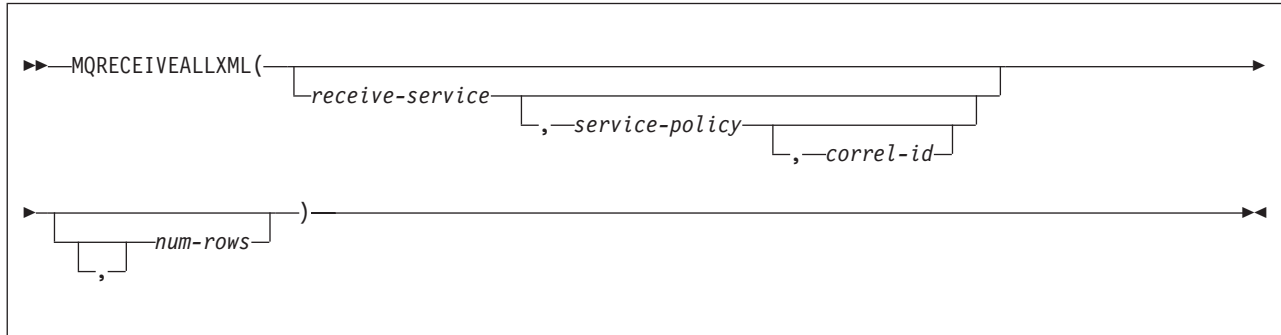
Example 2: Retrieve all the messages from the beginning of the queue specified by the service MYSERVICE, using the policy (DB2.DEFAULT.POLICY).

```
SELECT T.MSG, T.CORRELID  
FROM TABLE (MQRECEIVEALLCLOB('MYSERVICE')) T;
```

Only the MSG and CORRELID columns are returned as a table and removed from the queue.

MQRECEIVEALLXML

The MQRECEIVEALLXML function returns a table that contains the messages and message metadata from a specified MQSeries location and removes the messages from the queue.



The schema is DMQXML1C or DMQXML2C.

The MQRECEIVEALLXML function returns a table containing the messages and message metadata from the MQSeries location that is specified by *receive-service*, using the quality-of-service policy that is defined in *service-policy*. Performing this operation removes the messages from the queue that is associated with *receive-service*.

receive-service

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to a service point that is the logical MQSeries destination
| from which the message is read. A service point is defined in the DSNAMT
| repository file, and it represents a logical end-point from which a message is
| sent or received. A service point definition includes the name of the MQSeries
| queue manager and the name of the queue. See *MQSeries Application Messaging
| Interface* for more details.

If *receive-service* is not specified, DB2.DEFAULT.SERVICE is used.

service-policy

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The value of the expression must not be the
| null value, an empty string, or a string with trailing blanks. The expression
| must have an actual length that is no greater than 48 bytes. The value of the
| expression refers to an MQSeries AMI service policy that is used in handling
| this message. A service policy is defined in the DSNAMT repository file, and it
| specifies a set of quality-of-service options that are to be applied to this
| messaging operation. These options include message priority and message
| persistence. See *MQSeries Application Messaging Interface* for more details.

If *service-policy* is not specified, DB2.DEFAULT.POLICY is used.

correl-id

| An expression that returns a value that is a built-in character string or graphic
| string data type that is not a LOB. The expression must have an actual length
| that is no greater than 24 bytes. The value of the expression specifies the

correlation identifier to be associated with this message. The *correl-id* is often specified in request-and-reply scenarios to associate requests with replies.

A null value, an empty string, and a fixed length string with trailing blanks are all considered valid values. However, when the *correl-id* is specified on another request such as MQSENDXML, the *correl-id* must be specified the same to be recognized as a match. For example, specifying a value of 'test' for *correl-id* on MQRECEIVEALLXML does not match a *correl-id* value of 'test ' (with trailing blanks) specified earlier on an MQSENDXML request.

If *correl-id* is not specified, a correlation identifier is not used, and the message at the beginning of the queue is returned.

num-rows

An expression that specifies the maximum number of messages to return. It must be an integer that is greater than or equal to zero.

If *num-rows* is not specified or the value of expression is zero, all available messages are returned.

The result of the function is a table with the format shown in the following table. All of the columns are nullable.

Table 75. Format of resulting table for MQRECEIVEALLXML

Column name	Data type	Contains
MSG	DB2XML.XMLVARCHAR	The contents of the MQSeries message
CORRELID	VARCHAR(24)	The correlation ID that is used to relate messages
TOPIC	VARCHAR(40)	The topic that the message was published with, if available
QNAME	VARCHAR(48)	The name of the queue from which the message was received
MSGID	CHAR(24)	The unique, MQSeries-assigned identifier for the message
MSGFORMAT	VARCHAR(8)	The format of the message, as defined by MQSeries

Example 1: Retrieve all the messages from the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *  
FROM TABLE (MQRECEIVEALLXML()) T;
```

The messages and all the metadata are returned as a table. The messages are deleted from the queue.

Example 2: Retrieve all the messages from the beginning of the queue specified by the service MYSERVICE, using the default policy (DB2.DEFAULT.POLICY). Have only the MSG and CORRELID columns returned.

```
SELECT T.MSG, T.CORRELID  
FROM TABLE (MQRECEIVEALLXML('MYSERVICE')) T;
```

The messages and the data for the CORRELID column are returned as a table. The messages are deleted from the queue.

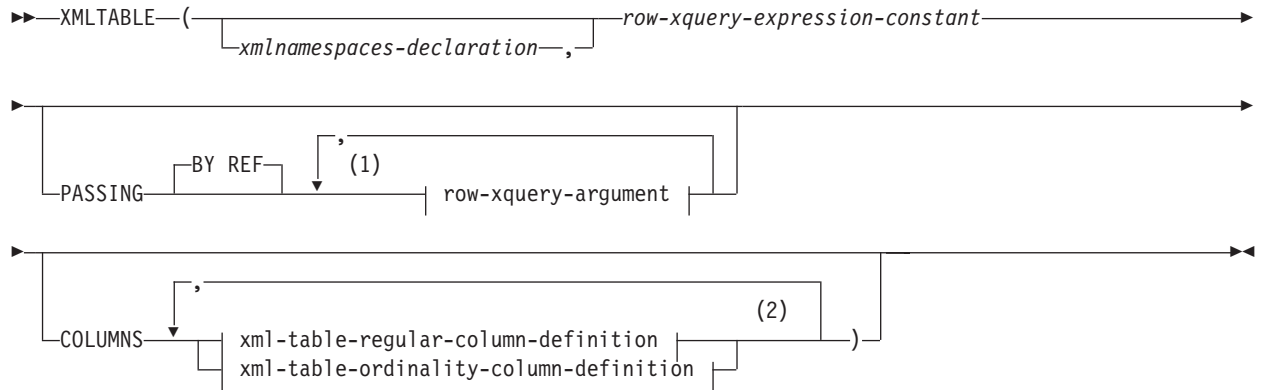
Example 3: Retrieve the first 10 the messages from the beginning of the queue specified by the default service (DB2.DEFAULT.SERVICE), using the default policy (DB2.DEFAULT.POLICY).

```
SELECT *  
FROM table (MQRECEIVEALLXML(10)) T;
```

The messages and all the metadata for the first 10 messages are returned as a table. The messages are deleted from the queue.

XMLTABLE

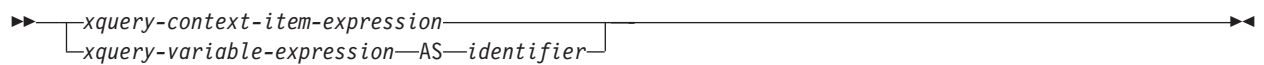
The XMLTABLE function returns a result table from the evaluation of XPath expressions, possibly by using specified input arguments as XPath variables. Each item in the result sequence of the row XPath expression represents one row of the result table.



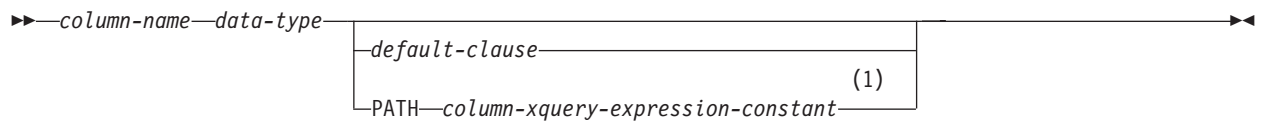
Notes:

- 1 *xquery-context-item-expression* must not be specified more than one time.
- 2 The *xml-table-ordinality-column-definition* clause must not be specified more than one time.

row-xquery-argument



xml-table-regular-column-definition



Notes:

- 1 Neither the *default-clause* or the **PATH** clause can be specified more than one time.

xml-table-ordinality-column-definition

► *column-name* — FOR ORDINALITY — ►

The schema is SYSIBM. The function name cannot be specified as a qualified name.

xmlnamespaces-declaration

Specifies one or more XML namespace declarations, using the XMLNAMESPACES function, that become part of the static context of the *row-xquery-expression-constant* and the *column-xquery-expression-constant*. The set of statically known namespaces for XPath expressions which are arguments of XMLTABLE is the combination of the pre-established set of statically known namespaces and the namespace declarations specified in this clause. The XPath prolog within an XPath expression can override these namespaces.

If *xmlnamespaces-declaration* is not specified, only the pre-established set of statically known namespaces apply to the XPath expressions.

row-xquery-expression-constant

Specifies an SQL character string constant that is interpreted as an XPath expression using supported XPath language syntax. This expression determines the number of rows in the result table. The expression is evaluated using the optional set of input XML values that is specified in *row-xquery-argument*, and returns an output XPath sequence where one row is generated for each item in the sequence. If the sequence is empty, the result of XMLTABLE is an empty table. *row-xquery-expression-constant* must not contain an empty string or a string of all blanks.

PASSING

Specifies input values and the manner in which these values are passed to *row-xquery-expression-constant*.

BY REF

Specifies that any XML input arguments are, by default, passed by reference. When XML values are passed by reference, the XPath evaluation uses the input node trees, if any exist, directly from the specified input expressions and preserves all properties, including the original node identities and document order.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

row-xquery-argument

Specifies an argument that is to be passed to the XPath expression specified by *row-xquery-expression-constant*. *row-xquery-argument* is an SQL expression that returns a value that is not a ROWID, LOB, DATE, TIME, TIMESTAMP, BINARY, VARBINARY, REAL, DECFLOAT, or character string with FOR BIT DATA attribute.

How *row-xquery-argument* is used in the XPath expression depends on whether the argument is specified as an *xquery-context-item-expression* or an *xquery-variable-expression*.

If the data type of *row-xquery-argument* is not XML, the result of the expression for the argument is implicitly cast to XML. A null value is converted to an XML empty sequence if the argument is *xquery-variable-expression*.

row-xquery-argument must not contain NEXT VALUE or PREVIOUS VALUE expressions or OLAP specifications.

xquery-context-item-expression

An expression that returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-context-item-expression* must not be a character string that is bit data.

xquery-context-item-expression specifies the initial context item for the *row-xquery-expression*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-variable-expression

Specifies an SQL expression whose value is available to the XPath expression specified by *row-xquery-expression-constant* during execution. The expression must return a value that is XML, integer, decimal, or a character or graphic string that is not a LOB.

xquery-variable-expression specifies an argument that will be passed to *row-xquery-expression-constant* as an XPath variable. If *xquery-variable-expression* is a null value, the XPath variable is set to an XML empty sequence. The scope of the XPath variables that are created from the PASSING clause is the XPath expression specified by *row-xquery-expression-constant*.

AS *identifier*

Specifies that the value generated by *xquery-variable-expression* will be passed to *row-xquery-expression-constant* as an XPath variable. The variable name will be *identifier*. The leading dollar sign (\$) that precedes variable names in the XPath language is not included in *identifier*. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier.

COLUMNS

Specifies the output columns of the result table including the column name, data type, and how the column value is computed for each row. If this clause is not specified, a single unnamed column of data type XML is returned, with the value based on the sequence item from evaluating the XPath expression in the *row-xquery-expression-constant* (equivalent to specifying PATH '.'). To reference the result column, a *column-name* must be specified in the *correlation-clause* following the function.

xml-table-regular-column-definition

Specifies one output column of the result table including the column name, data type, and an XPath expression to extract the value from the sequence item for the row.

column-name

Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table.

data-type

Specifies the data type of the column. See CREATE TABLE for the syntax and a description of types available. A *data-type* can be used in XMLTABLE if there is a supported XMLCAST from the XML data type to the specified *data-type*.

default-clause

Specifies a default value for the column. See CREATE TABLE for the syntax and a description of the *default-clause*. For XMLTABLE result columns, the default is applied when the processing of the XPath expression contained in *column-xquery-expression-constant* returns an empty sequence. This default value will not be inherited by declared global temporary tables even when the INCLUDING COLUMN DEFAULTS clause is specified in the definition of the declared global temporary table.

PATH *column-xquery-expression-constant*

Specifies an SQL character string constant that is interpreted as an XPath expression using supported XPath language syntax. The *column-xquery-expression-constant* specifies an XPath expression that determines the column value with respect to an item that is the result of evaluating the XPath expression in *row-xquery-expression-constant*. Given an item from the result of processing the *row-xquery-expression-constant* as the externally provided context item, the *column-xquery-expression-constant* is evaluated and returns an output sequence. The column value is determined based on this output sequence as follows.

- If the output sequence contains zero items, the *default-clause* provides the value of the column.
- If an empty sequence is returned and no *default-clause* was specified, a null value is assigned to the column.
- If a non-empty sequence is returned, the value is cast to the *data-type* specified for the column using the XMLCAST expression. An error could be returned from processing this XMLCAST.

The value for *column-xquery-expression-constant* must not be an empty string or a string of all blanks. If this clause is not specified, the default XPath expression is simply the *column-name*.

xml-table-ordinality-column-definition

Specifies the ordinality column of the result table.

column-name

Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table.

FOR ORDINALITY

Specifies that *column-name* is the ordinality column of the result table. The data type of this column is BIGINT. The value of this column in the result table is the sequential number of the item for the row in the resulting sequence from evaluating the XPath expression in *row-xquery-expression-constant*.

The result of the function is a table. The encoding scheme of the table is Unicode. If the evaluation of any of the XPath expressions results in an error, the XMLTABLE function returns the XPath error.

Example: List as a table result the purchase order items for orders with a status of 'NEW':

```
SELECT U."PO ID", U."Part #", U."Product Name",  
       U."Quantity", U."Price", U."Order Date"  
FROM PURCHASEORDER P,  
     XMLTABLE(XMLNAMESPACES('http://podemo.org' AS "pod"),
```

```

|                                     '$po/PurchaseOrder/itemlist/item' PASSING P.PORDER as "po"
|
|      COLUMNS "PO ID"              INTEGER      PATH '../../@POid',
|              "Part #"             CHAR(6)       PATH 'product/@pid',
|              "Product Name"       CHAR(50)      PATH 'product/pod:name',
|              "Quantity"           INTEGER      PATH 'quantity',
|              "Price"              DECIMAL(9,2)  PATH 'product/pod:price',
|              "Order Date"         TIMESTAMP    PATH '../..//dateTime'
|
|      ) AS U
|
| WHERE P.STATUS = 'NEW'

```

Chapter 4. Queries

A *query* specifies a result table or an intermediate table. A query is a component of certain SQL statements. There are three forms of a query.

A “subselect” on page 632

A “fullselect” on page 662

A “select-statement” on page 669

A subselect is a subset of a fullselect, and a fullselect is a subset of a *select-statement*.

“Authorization” on page 630 describes the privilege set that is required to use any form of a query.

Another SQL statement that can be used to retrieve at most a single row is described in “SELECT INTO” on page 1471. SELECT INTO is not a subselect, fullselect, or a *select-statement*.

Authorization

For any form of a query, the privilege set that is defined below must include one of the following:

- For each table or view identified in the statement, the privilege set must include one of the following:

- Ownership of the table or view
- The SELECT privilege on the table or view
- DBADM authority for the database (tables only)

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

- SYSADM authority
- SYSCTRL authority (catalog tables only)

If a query includes a user-defined function, the privileges that are held by the authorization ID of the statement must include at least one of the following:

- For each user-defined function that is identified in the statement, one of the following:
 - The EXECUTE privilege on the function
 - Ownership of the function
- SYSADM authority

If the *select-statement* is part of a DECLARE CURSOR statement, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package.

If the *select-statement* contains an SQL data change statement, the privilege set must include the SELECT privilege and the appropriate privileges for the SQL data change statement (insert, update, or delete privileges) on the target table or view.

For dynamically prepared statements, the privilege set depends on the dynamic SQL statement behavior, which is specified by option DYNAMICRULES:

Run behavior

The privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Bind behavior

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

Define behavior

The privilege set is the privileges that are held by the authorization ID of the owner of the stored procedure or user-defined function.

Invoke behavior

The privilege set is the privileges that are held by the authorization ID of the invoker of the stored procedure or user-defined function.

For a list of the DYNAMICRULES values that specify run, bind, define, or invoke behavior, see Table 6 on page 64.

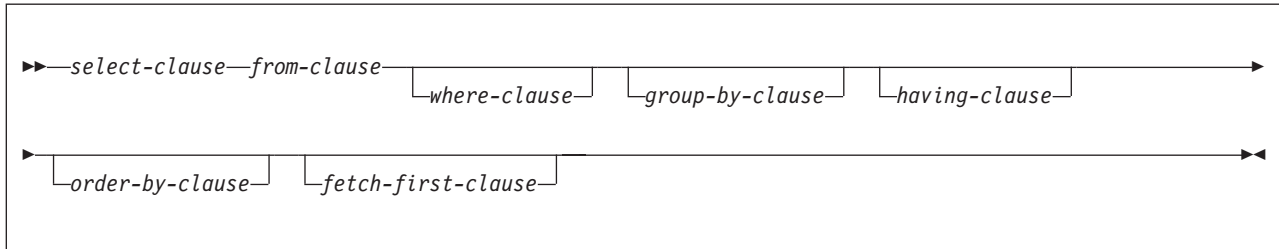
When any form of a query is used as a component of another statement, the authorization rules that apply to the query are specified in the description of that

statement. For example, see “CREATE VIEW” on page 1184 for the authorization rules that apply to the subselect component of CREATE VIEW.

If your installation uses the access control authorization exit (DSNX@XAC), that exit might be controlling the authorization rules instead of the rules that are listed here.

subselect

The *subselect* is a component of the fullselect. A subselect specifies a result table that is derived from the tables or views that are identified in the FROM clause.



The derivation of the result table can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method that is used to perform the derivation might be quite different from this description. If portions of the subselect do not actually need to be executed for the correct result to be obtained, they might not be executed.)

A *scalar-subselect* is a subselect, enclosed in parentheses, that returns a single result row and a single result column. If the result of the subselect is no rows, the null value is returned. An error is returned if the result contains more than one row.

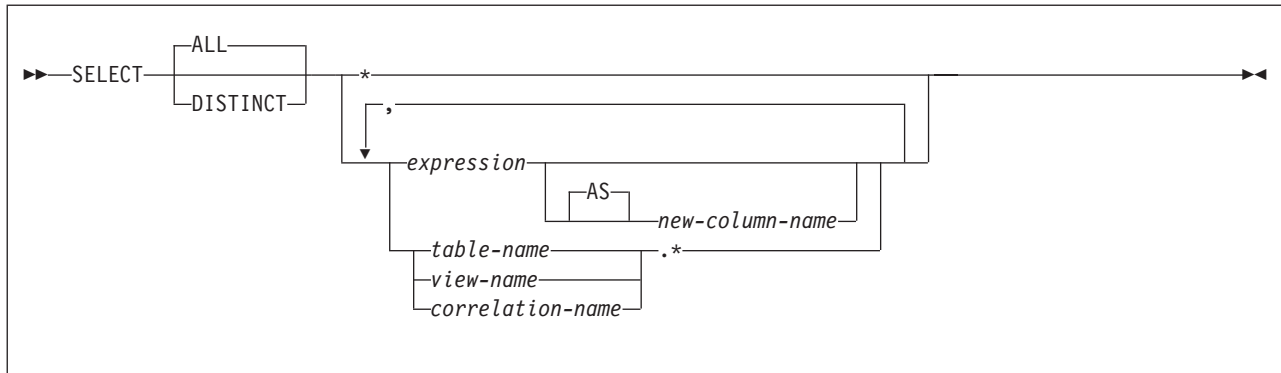
The clauses of the subselect are processed in the following sequence:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause
7. FETCH FIRST clause

select-clause

The SELECT clause specifies the columns of the final result table. The column values are produced by the application of the *select list* to R. The select list is a list of names and expressions specified in the SELECT clause, and R is the result of the previous operation of the subselect. For example, if SELECT, FROM, and WHERE are the only clauses specified, then R is the result of that WHERE clause.

select-clause



ALL

Retains all rows of the final result table and does not eliminate redundant duplicates. This is the default.

DISTINCT

Eliminates all but one of each set of duplicate rows of the final result table.

Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value in the second row. For determining duplicate rows, two null values are considered equal.

When SELECT DISTINCT is specified, no column or expression in the implicit or explicit list can return a value that is a LOB or XML data type. When a column or expression in the list returns a value that is a DECFLOAT data type and multiple bit representations of the same number exists in the intermediate result, the value that is returned is unpredictable. See “Numeric comparisons” on page 114 for additional information.

Select list notation:

- * Represents a list of columns of table R, excluding any columns that are defined as implicitly hidden. The list of names is established when the statement containing the SELECT clause is prepared. Therefore, * does not identify any columns that have been added to a table after the statement has been prepared.

A column that is defined as implicitly hidden can be explicitly referenced in the select list.

expression

Specifies the values of a result column. Each *column-name* in the expression must unambiguously identify a column of the intermediate result table.

AS *new-column-name*

Names or renames the result column. The name must not be qualified and does not have to be unique. *new-column-name* is an SQL identifier of 128 UTF-8 bytes or less.

*name.**

Represents a list of columns of *name*, excluding any columns that are defined as implicitly hidden, in the order the columns are produced by the FROM clause. *name* can be a table name, view name, or correlation name, and must designate an exposed table, view, or correlation name in the FROM clause that immediately follows the SELECT clause. The first name in the list identifies the first column of the table or view, the second name in the list identifies the second column of the table or view, and so on.

The list of names is established when the statement that contains the SELECT clause is prepared. Therefore, * does not identify any columns that have been added to a table after the statement has been prepared.

SQL statements can be implicitly or explicitly prepared again. The effect of another prepare on statements that include * or *name.** is that the list of names is re-established. Therefore, the number of columns returned by the statement might change.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established at the time the statement is prepared), and cannot exceed 750. The result of a subquery must be a single column unless the subquery is used in an EXISTS predicate.

If the FROM clause contains a MERGE statement: The SELECT list must not implicitly or explicitly refer to a column that has a LOB data type, a ROWID data type (or a distinct type that is based on a LOB, or ROWID), or an XML data type.

Implicitly hidden ROWID columns in the select list: The result for SELECT * does not include any implicitly hidden ROWID columns. To be included in the result, implicitly hidden ROWID columns must be explicitly specified in the select list.

VARBINARY data: If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To query the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.

Applying the select list: Some of the results of applying the select list to R depend on whether GROUP BY or HAVING is used. The following three lists describe the results.

IF neither GROUP BY nor HAVING is used:

- The select list can include aggregate functions only if it includes other aggregate functions, constants, or expressions that only involve constants.
- If the select list does not include aggregate functions, it is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list includes aggregate functions, R is the source of the arguments of the functions and the result of applying the select list is one row, even when R has no rows.

If HAVING is used and GROUP BY is not used:

Each expression or *column-name* in an expression in the select list must be specified within an aggregate function. Constants or expressions that involve only constants can also be in the select list.

If GROUP BY is used:

- Each expression in the select list must use one or more grouping expressions. Or, each expression or *column-name* in an expression must:
 - Unambiguously identify a grouping column of R.
 - Be specified within an aggregate function.
 - Be a correlated reference. (A column-name is a correlated reference if it identifies a column of a table or view identified in an outer subselect.)
- If an expression in the select list is a scalar fullselect, a correlated reference from the scalar fullselect to a group R must either identify a grouping column or be contained within an aggregate function. For example, the following query fails because the correlated reference T1.C1 || T1.C2 in the select list of the scalar fullselect does not match a grouping column from the outer subselect. (Matching the grouping expression T1.C1 || T1.C2 is not supported.)

```
SELECT MAX(T1.C2) AS X1,  
       (SELECT T1.C1 || T1.C2 FROM T2 GROUP BY T2.C1) AS Y1  
FROM T1  
GROUP BY T1.C1, T1.C1 || T1.C2;
```

- You cannot use GROUP BY with a name defined using the AS clause unless the name is defined in a nested table expression. Example 6 demonstrates the valid use of AS and GROUP BY in a SELECT statement.

In either case, the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

Null attributes of result columns: Result columns allow null values if they are derived from one of the following:

- Any aggregate function except COUNT or COUNT_BIG
- A column that allows null values
- A view column in an outer select list that is derived from an arithmetic expression
- An arithmetic expression in an outer select list
- An arithmetic expression that allows nulls
- A scalar function or string expression that allows null values
- A host variable that has an indicator variable, or in the case of Java, a host variable or expression whose type is able to represent a Java null value
- A result of a set operator if at least one of the corresponding items in the select list is nullable

Names of result columns: In the following cases a result column is considered a named column:

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.

- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), the result column name is the unqualified name of that column.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single SQL variable or SQL parameter (without any functions or operators), the result column name is the unqualified name of that SQL variable or SQL parameter.

In all other cases, a result column is an unnamed column.

Names of result columns are placed into the SQL descriptor area (SQLDA) when the DESCRIBE statement is executed. This allows an interactive SQL processor such as SPUFI, the command line processor, or DB2 QMF to use the column names when displaying the results. The names in the SQLDA include those specified by the AS clause.

Data types of result columns: Each column of the result of SELECT acquires a data type from the expression from which it is derived. Table 76 shows the data types of result columns.

Table 76. Data types of result columns

When the expression is...	The data type of the result column is...
The name of any numeric column	The same as the data type of the column, with the same precision and scale for decimal columns.
An integer constant	INTEGER.
A decimal or floating-point constant	The same as the data type of the constant, with the same precision and scale for decimal constants. For floating-point constants, the data type is DOUBLE PRECISION.
A decimal floating point constant	DECFLOAT(34)
The name of any numeric host variable	The same as the data type of the variable, with the same precision and scale for decimal variables. The result is decimal if the data type of the host variable is not an SQL data type; for example, DISPLAY SIGN LEADING SEPARATE in COBOL.
An arithmetic or string expression	The same as the data type of the result, with the same precision and scale for decimal results as described in “Expressions” on page 180.
Any function	The data type of the result of the function. For a built-in function, see Chapter 3, “Functions,” on page 259 to determine the data type of the result. For a user-defined function, the data type of the result is what was defined in the CREATE FUNCTION statement for the function.
The name of any string column	The same as the data type of the column, with the same length attribute.
The name of any string host variable	The same as the data type of the variable, with a length attribute equal to the length of the variable. The result is a varying-length character string if the data type of the host variable is not an SQL data type; for example, a NUL-terminated string in C.
A character string constant of length <i>n</i>	VARCHAR(<i>n</i>).
A binary string constant of length <i>n</i>	VARBINARY(<i>n</i>)
A graphic string constant of length <i>n</i>	VARGRAPHIC(<i>n</i>).
The name of a datetime column	The same as the data type of the column.
The name of a ROWID column	Row ID.

Table 76. Data types of result columns (continued)

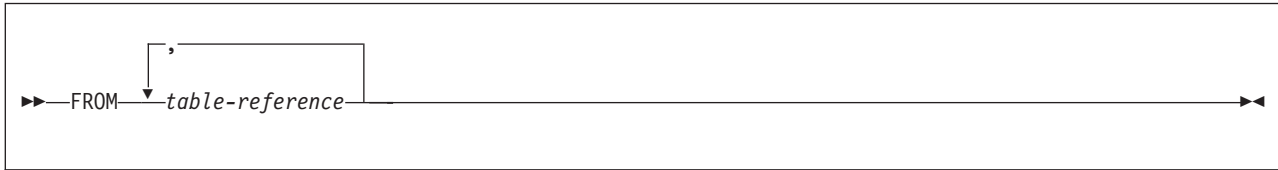
When the expression is...	The data type of the result column is...
The name of a distinct type column	The same as the distinct type of the column, with the same length, precision, and scale attributes, if any.

For information about the CCSID of the result column, see “Rules for result data types” on page 119.

from-clause

The FROM clause specifies an intermediate result table.

from-clause



If only one *table-reference* is specified, the intermediate result table is simply the result of that *table-reference*. If more than one *table-reference* is specified, the intermediate result table consists of all possible combinations of the rows of the result of each specified *table-reference*.

Each row of the result is a row from the result of the first *table-reference* concatenated with a row from the result of the second *table-reference*, concatenated with a row from the result of the third *table-reference*, and so on. The number of rows in the result is the product of the number of rows in the result of each *table-reference*. Thus, if the result of any *table-reference* is empty, the result is empty.

If a *table-reference* contains a security label column, DB2 compares the security label of the user to the security label of each row. Results are returned according to the following rules:

- If the security label of the user dominates the security label of the row, DB2 returns the row.
- If the security label of the user does not dominate the security label of the row, DB2 does not return the data from that row, and DB2 does not generate an error report.

table-reference

A *table-reference* specifies a result table as either a table or view, or an intermediate table.

table-reference:



single-table:

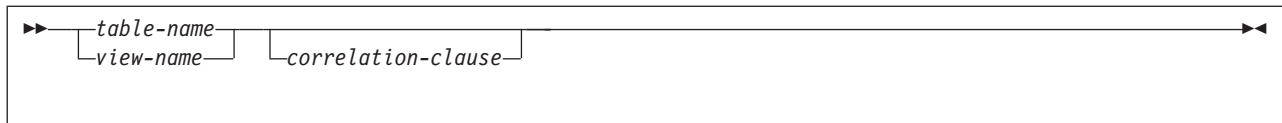
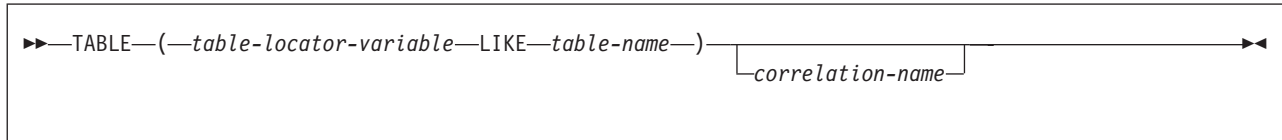


table-locator-reference:



nested-table-expression:

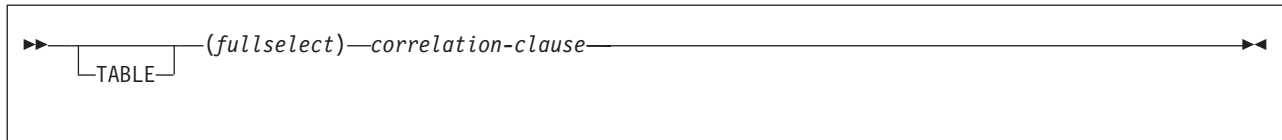


table-function-reference:

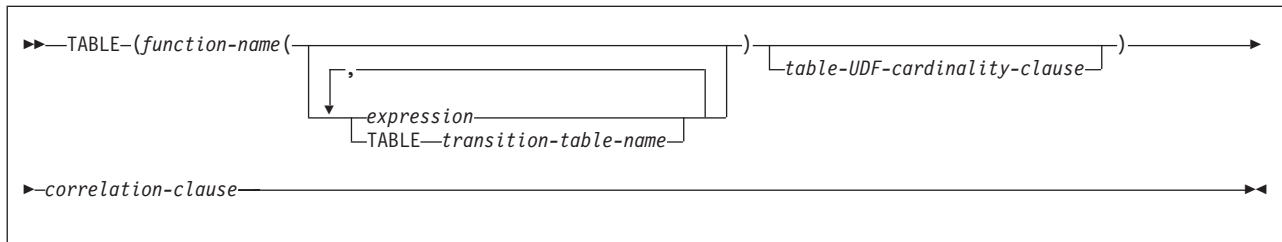
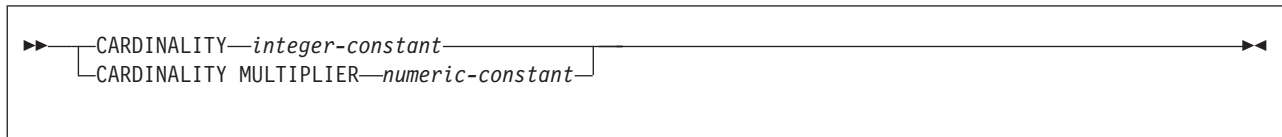
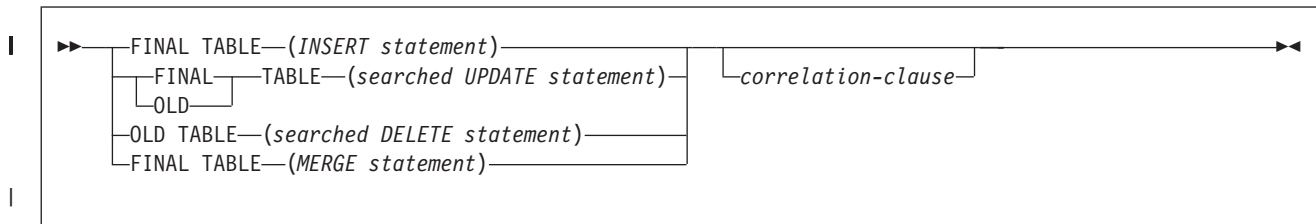


table-UDF-cardinality-clause:



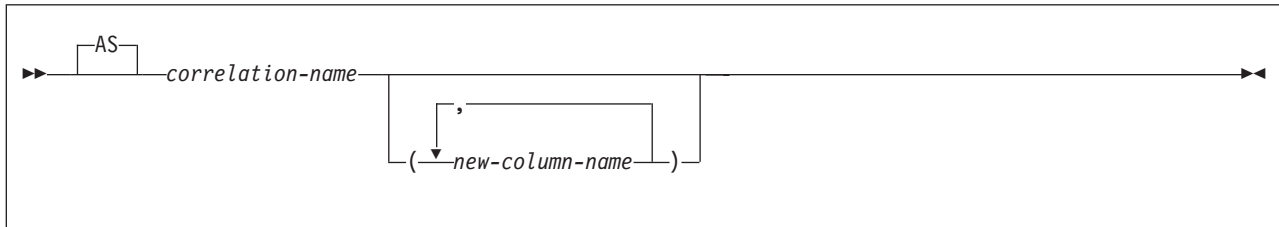
data-change-table-reference:



xmltable-expression:

►► *xmltable-function-correlation clause* ◀◀

correlation-clause:



- If a single table or view is identified, the result table is simply that table or view.
- If a table locator is identified, the host variable represents the intermediate table. The intermediate table has the same structure as the table identified in *table-name*.

Each *table-name* or *view-name* specified in every **FROM** clause of the same SQL statement must identify a table or view that exists at the same DB2 subsystem. If a **FROM** clause is specified in a subquery of a basic predicate, a view that includes **GROUP BY** or **HAVING** must not be identified.

A *table-reference* must not identify a table that was implicitly created for an XML column.

table-locator-reference

Each *table-locator-variable* must specify a host variable with a table locator type. The only way to assign a value to a table locator is to pass the old or new transition table of a trigger to a user-defined function or stored procedure. A table locator host variable must not have a null indicator and must not be a parameter marker. In addition, a table locator can be used only in a manipulative SQL statement.

nested-table-expression

A *fullselect* in parentheses is called a *nested table expression*. If a nested table expression is specified, the result table is the result of that *nested-table-expression*. The columns of the result do not need unique names, but a column with a non-unique name cannot be referenced. At any time, the table consists of the rows that would result if the fullselect were executed.

table-function-reference

If a *function-name* is specified, the result table is the set of rows returned by the table function.

Each *function-name*, together with the types of its arguments, must resolve to a table function that exists at the same DB2 subsystem. An algorithm called function resolution, which is described in "Function resolution" on page 175, uses the function name and the arguments to determine the exact function to use. Unless given column names in the *correlation-clause*, the column names for a table function are those specified on the RETURNS clause of the CREATE FUNCTION statement. This is analogous to the column names of a table, which are defined in the CREATE TABLE.

table-UDF-cardinality-clause

The *table-UDF-cardinality clause* can be specified to each user-defined table function reference within the table spec of the FROM clause in a subselect.

This option indicates the expected number of rows to be returned only for the SELECT statement that contains it.

CARDINALITY *integer-constant* specifies an estimate of the expected number of rows returned by the reference to the user-defined function. The value of *integer-constant* must range from 0 to 2147483647.

The value set in the CARDINALITY field of SYSIBM.SYSROUTINES for the table function name is used as the reference cardinality value. The product of the specified **CARDINALITY MULTIPLIER** *numeric-constant* and the reference cardinality value are used by DB2 as the expected number of rows returned by the table function reference.

In this case, the *numeric-constant* can be in the integer, decimal, or floating-point format. The value must be greater than or equal to zero. If the decimal number notation is used, the number of digits can be up to 31. An integer value is treated as a decimal number with no fraction. The maximum value allowed for a floating-point number is about 7.237E + 75. If no value has been set in the CARDINALITY field of SYSIBM.SYSROUTINES, its default value is used as the reference cardinality value. If zero is specified or the computed cardinality is less than 1, DB2 assumes that the cardinality of the reference to the user-defined table function is 1.

Only a numeric constant can follow the keyword **CARDINALITY** or **CARDINALITY MULTIPLIER**. No host variable or parameter marker is allowed in a cardinality option. Specifying a cardinality option in a table function reference does not change the corresponding CARDINALITY field in SYSIBM.SYSROUTINES. The CARDINALITY field value in SYSIBM.SYSROUTINES can be initialized by the **CARDINALITY** option in the CREATE FUNCTION (external table) statement when a user-defined table function is created. It can be changed by the **CARDINALITY** option in the ALTER FUNCTION statement or by a direct update operation to SYSIBM.SYSROUTINES.

data-change-table-reference

A *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the SQL data change statement that is included in the clause. A *data-change-table-reference* can only be specified as the only *table-reference* in the FROM clause of the outer fullselect that is used in a select-statement and that fullselect must be in a subselect, or a SELECT INTO statement. A *data-change-table-reference* in a SELECT statement of a cursor makes the cursor read only. The target table or view of the SQL data change statement is a table or view that is reference in the query. The privileges that are held by the authorization ID of the statement must include the SELECT privilege on that target table or view.

Expressions in the select list of a view in a table reference can only be selected if OLD TABLE is specified or if the expression does not include any of the following objects:

- a function that is defined to read or modify SQL data
- a function that is defined as not deterministic or has an external action
- a NEXT VALUE expression for a sequence

FINAL TABLE

Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they appear at the completion of the SQL data change statement. If there are AFTER triggers that result in further operations on the table that is the target of

the SQL data change statement, an error is returned. If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned.

OLD TABLE

The rows of the intermediate result table represent the set of affected rows as they exist prior to the application of the SQL data change statement. If OLD TABLE is specified, the select list cannot contain an XML column.

INSERT statement

Specifies an INSERT statement as described in “INSERT” on page 1367. A fullselect in the INSERT statement cannot contain correlated references to columns that are outside of the fullselect of the INSERT statement. The target of the INSERT statement must be a base table, a view that is defined with the **WITH CASCADED CHECK** clause, or a view where the view definition has no **WHERE** clause. If there are input variables elsewhere in the fullselect, the INSERT statement cannot be a multiple row not atomic insert, or a multiple row atomic insert that specifies the **USING DESCRIPTOR** clause.

MERGE statement

Specifies a MERGE statement as described in “MERGE” on page 1388. A table reference in the MERGE statement must not contain correlated references to columns that are outside of the table reference in the MERGE statement.

If the MERGE statement is used in the SELECT statement and the MERGE statement references a view, the view must be defined using the **WITH CASCADED CHECK OPTION** clause.

The target table or view of the MERGE statement must not have a column with a ROWID, LOB, or XML data type.

AFTER triggers that result in further operations on the target table cannot exist on the target table.

searched UPDATE statement

Specifies a searched UPDATE statement as described in “UPDATE” on page 1521. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated referenced to columns that are outside of the UPDATE statement. The target of the UPDATE statement must be a base table, a symmetric view, or a view where the view definition has no WHERE clause.

If the searched UPDATE statement is used in the SELECT statement and the UPDATE statement references a view, the view must be defined using the **WITH CASCADED CHECK OPTION** clause.

AFTER triggers that result in further operations on the target table cannot exist on the target table.

searched DELETE statement

Specifies a searched DELETE statement as described in “DELETE” on page 1224. A WHERE clause in the DELETE statement cannot contain correlated references to columns that are outside of the DELETE statement. The target of the DELETE statement must be a base table, a symmetric view, or a view where the view definition has no WHERE clause.

If the searched DELETE statement is used in the SELECT statement and the DELETE statement references a view, the view must be defined using the **WITH CASCADED CHECK OPTION** clause.

AFTER triggers that result in further operations on the target table cannot exist on the target table.

The content of the intermediate result table for a table reference that contains an SQL data change statement is determined when the cursor is opened. The intermediate result table includes a column for each of the columns of the target table (including implicitly hidden columns) or view. All of the columns of the target table or view of an SQL data change statement are accessible by using the names of the columns from the target table or view unless the columns are renamed by using the correlation clause. If a *correlation-name* is not specified, the column names can be qualified by the target table or view name of the SQL data change statement. If an INCLUDE clause is specified as part of the SQL data change statement, the intermediate result table will contain these additional columns.

joined-table

If a *joined-table* is specified, the result table is the result of one or more join operations as explained in “joined-table” on page 645.

correlation-clause

Each *correlation-name* in a *correlation-clause* defines a designator for the immediately preceding result table (*table-name*, *view-name*, *nested-table-expression*, or *function-name* reference), which can be used to qualify references to the columns of the table. *new-column-names* is an SQL identifier of 128 UTF-8 bytes or less. Using *new-column-names* to list and rename the columns is optional. A correlation name must be specified for nested table expressions and references to table functions.

If no *correlation-name* is specified for *data-change-table-references*, the *correlation-name* is the name of the target table or view of the SQL data change statement. Otherwise, the correlation name is the *correlation-name*.

If a list of *new-column-names* is specified in a *correlation-clause*, the number of names must be the same as the number of columns in the corresponding table, view, nested table expression, or table function. Each name must be unique and unqualified. If columns are added to an underlying table of a *table-reference*, the number of columns in the result of the *table-reference* no longer matches the number of names in its *correlation-clause*. Therefore, when a rebind of a package containing the query in question is attempted, DB2 returns an error and the rebind fails. At that point, change the *correlation-clause* of the embedded SQL statement in the application program so that the number of names matches the number of columns. Then, precompile, compile, bind, and link-edit the modified program.

An exposed name is a *correlation-name* or a *table-name* or view name that is not followed by a *correlation-name*. The exposed names in a FROM clause should be unique, and only exposed names should be used as qualifiers of column names. Thus, if the same table name is specified twice, at least one specification of the table name should be followed by a unique correlation name. That correlation name should be used to qualify references to columns of that instance of the table. In addition, if column names are listed for the correlation name in the FROM clause, those column names should be used to reference the columns. For more information, see “Column name qualifiers in correlated references” on page 157.

xmltable-function

Specifies an invocation of the built-in XMLTABLE function. See “XMLTABLE” on page 624 for more information.

Correlated references in *table-references*: In general, nested table expressions and table functions can be specified in any FROM clause. Columns from the nested table expressions and table functions can be referenced in the select list and in the rest of the fullselect using the correlation name. The scope of this correlation name is the same as correlation names for other table or view names in the FROM clause. The basic rule that applies for both these cases is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries.

Nested table expressions can be used in place of a view to avoid creating a view when general use of the view is not required. They can also be used when the desired result table is based on host variables.

For table functions, an additional capability exists. A table function can contain one or more correlated references to other tables in the same FROM clause if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause. The same capability exists for nested table expressions if the optional keyword TABLE is specified; otherwise, only references to higher levels in the hierarchy of subqueries is allowed.

A nested table expression or table function that contains correlated references to other tables in the same FROM clause:

- Cannot participate in a FULL OUTER JOIN or a RIGHT OUTER JOIN
- Can participate in LEFT OUTER JOIN or an INNER JOIN if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause

The following table shows some examples of valid and invalid correlated references. TABF1 and TABF2 represent table functions.

Table 77. Examples of correlated references

Subselect	Valid	Reason
SELECT T.C1, Z.C5 FROM TABLE(TABF1(T.C2)) AS Z, T WHERE T.C3 = Z.C4;	No	T.C2 cannot be resolved because T does not precede TABF1 in FROM
SELECT T.C1, Z.C5 FROM T, TABLE(TABF1(T.C2)) AS Z WHERE T.C3 = Z.C4;	Yes	T precedes TABF1 in FROM, making T.C2 known
SELECT A.C1, B.C5 FROM TABLE(TABF2(B.C2)) AS A, TABLE(TABF1(A.C6)) AS B WHERE A.C3 = B.C4;	No	B in B.C2 cannot be resolved because the table function that would resolve it, TABF1, follows its reference in TABF2 in FROM
SELECT D.DEPTNO, D.DEPTNAME, EMPINFO.AVGSAL, EMPINFO.EMPCOUNT FROM DEPT D, (SELECT AVG(E.SALARY) AS AVGSAL, COUNT(*) AS EMPCOUNT FROM EMP E WHERE E.WORKDEPT = D.DEPTNO) AS EMPINFO;	No	DEPT precedes nested table expression, but keyword TABLE is not specified, making D.DEPTNO unknown

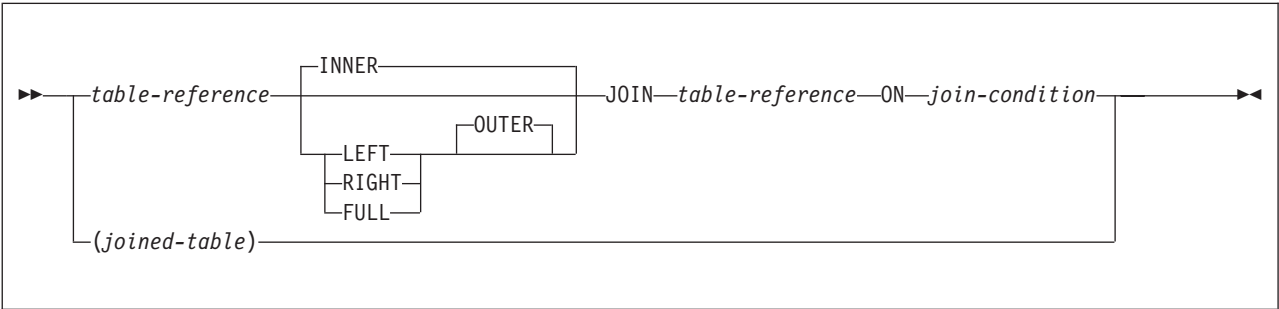
Table 77. Examples of correlated references (continued)

Subselect	Valid	Reason
SELECT D.DEPTNO, D.DEPTNAME, EMPINFO.AVGSAL, EMPINFO.EMPCOUNT FROM DEPT D, TABLE (SELECT AVG(E.SALARY) AS AVGSAL, COUNT(*) AS EMPCOUNT FROM EMP E WHERE E.WORKDEPT = D.DEPTNO) AS EMPINFO;	Yes	DEPT precedes nested table expression and keyword TABLE is specified, making D.DEPTNO known

joined-table

A *joined-table* specifies a result table that is the result of either an inner equi-join or an outer join. The table is derived by applying one of the join-operators: INNER, RIGHT OUTER, LEFT OUTER, or FULL OUTER to its operands. If a join-operator is not specified, INNER is implicit. The order in which a LEFT OUTER JOIN or RIGHT OUTER JOIN is performed can affect the result.

joined-table



As described in more detail under “Join operations” on page 647 an inner join combines each row of the left table with every row of the right table keeping only the rows where the join-condition is true. Thus, the result table might be missing rows from either or both of the joined tables. Outer joins include the rows produced by the inner join as well as the missing rows, depending on the type of outer join as follows:

- Left outer.* Includes the rows from the left table that were missing from the inner join.
- Right Outer.* Includes the rows from the right table that were missing from the inner join.
- Full Outer.* Includes the rows from both tables that were missing from the inner join.

A *joined-table* can be used in any context in which any form of the SELECT statement is used. Both a view and a cursor is read-only if its SELECT statement includes a *joined-table*. If LEFT OUTER JOIN, RIGHT OUTER JOIN, or FULL OUTER JOIN is specified, the RID built-in function and the ROW CHANGE expression must not be specified in the subselect that contains the FROM clause.

join-condition specifies the conditions of a join that is used in a query.

join-condition

►►—*search-condition*—◄◄

```

graph LR
    Start(( )) --> E1[full-join-expression]
    E1 -- AND --> E2[full-join-expression]
    E2 --> End(( ))
  
```

```

graph LR
    Start(( )) --> COALESCE[COALESCE]
    COALESCE --> LParen[(]
    LParen --> Arg1Box[ ]
    Arg1Box --> Arg1Top[column-name]
    Arg1Box --> Arg1Bot[cast-function]
    Arg1Box --> P1((1))
    Arg1Box --> RParen1[)]
    Arg1Box --> Comma[,]
    Comma --> Arg2Box[ ]
    Arg2Box --> Arg2Top[,column-name]
    Arg2Box --> Arg2Bot[,cast-function]
    Arg2Box --> P2((1))
    Arg2Box --> RParen2[)]
    Arg2Box --> RParen1
    RParen1 --> End(( ))
  
```

Notes:

- 1 *cast-function* must only contain a column and the casting data type must be a distinct type or the data type upon which the distinct type was based.

- With one exception, It cannot contain any subqueries. If the *join-table* that contains the *join-condition* in the associated FROM clause is composed of only INNER joins, the *join-condition* can contain subqueries.
- Any column that is referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join operator (in the scope of the same *joined-table* clause).

For a FULL OUTER (or FULL) join, the *join-condition* is a search condition in which the predicates can only be combined with AND. In addition, each predicate must have the form 'expression = expression', where one expression references only columns of one of the operand tables of the associated join operator, and the other expression references only columns of the other operand table. The values of the expressions must be comparable.

Each *full-join-expression* in a FULL OUTER join must include a column name or a cast function that references a column. The COALESCE function is allowed.

For any type of join, column references in an expression of the *join-condition* are resolved using the rules for resolution of column name qualifiers specified in "Resolution of column name qualifiers and column names" on page 158 before any rules about which tables the columns must belong to are applied.

Join operations

A *join-condition* specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of its associated JOIN operator. For all possible combinations of rows T1 and T2, a row of T1 is paired with a row of T2 if the join-condition is true. When a row of T1 is joined with a row of T2, a row in the result consists of the values of that row of T1 concatenated with the values of that row of T2. The execution might involve the generation of a "null row". The null row of a table consists of a null value for each column of the table, regardless of whether the columns allow null values.

The following summarizes the results of the join operations:

- The result of T1 INNER JOIN T2 consists of their paired rows.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.
- The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.
- The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T1, the concatenation of that row with the null row of T2, and for each unpaired row of T2, the concatenation of that row with the null row in T1. All columns of the result table allow null values.

A join operation is part of a FROM clause; therefore, for the purpose of predicting which rows will be returned from a SELECT statement containing a join operation, assume that the join operation is performed before the other clauses in the statement.

where-clause

The WHERE clause specifies a result table that consists of those rows of R for which the search condition is true. R is the result of the FROM clause of the subselect.

where-clause

►►—WHERE—*search-condition*—◄◄

The search condition must conform to the following rules:

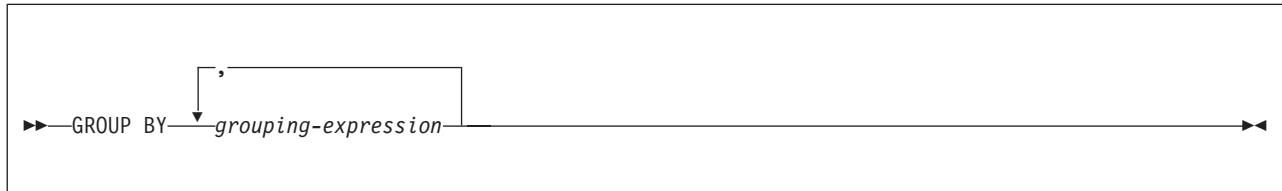
- Each column name must unambiguously identify a column of R or be a correlated reference. A column name is a correlated reference if it identifies a column of a table or view that is identified in an outer subselect.
- An aggregate function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the *search-condition* is effectively executed for each row of R and the results are used in the application of the *search-condition* to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references is executed just one time, whereas a subquery with a correlated reference might have to be executed one time for each row.

group-by-clause

The GROUP BY clause specifies a result table that consists of a grouping of the rows of intermediate result table that is the result of the previous clause.

group-by-clause



grouping-expression is an expression that defines the grouping of R. The following restrictions apply to *grouping-expression*:

- If *grouping-expression* is a single column, the column name must unambiguously identify a column of R.
- The result of *grouping-expression* cannot be a LOB data type (or a distinct type that is based on a LOB) or an XML data type.
- *grouping-expression* cannot include any of the following items:
 - A correlated column
 - A host variable
 - An aggregate function
 - Any function or expression that is not deterministic or that is defined to have an external action
 - A scalar fullselect
 - A CASE expression whose *searched-when-clause* contains a quantified predicate, an IN predicate using a fullselect, or an EXISTS predicate

The result of GROUP BY is a set of groups of rows. In each group of more than one row, all values of each *grouping-expression* are equal, and all rows with the same set of values of the *grouping-expression* are in the same group. For grouping, all null values for a *grouping-expression* are considered equal.

If a *grouping-expression* contains DECFLOAT values, the DECFLOAT values with the same value will be in the same group. But the number of digits returned for each group is unpredictable.

Because every row of a group contains the same value of any *grouping-expression*, a *grouping-expression* can be used in a search condition in a HAVING clause or an expression in a SELECT clause, or in a *sort-key-expression* of an ORDER BY clause. In each case, the reference specifies only one value for each group. For example, if *grouping-expression* is col1+col2, col1+col2+3 would be an allowed expression in the select list. Associative rules for expressions do not allow the similar expression of 3+col1+col2, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, 3+(col1+col2) would also be allowed in the select list. If the concatenation operator is used, *grouping-expression* must be used exactly as the expression was specified in the select list.

If a *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and might not all

have the same length. In that case, a reference to *grouping-expression* still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

having-clause

The HAVING clause specifies a result table that consists of those groups of the intermediate result table for which the search-condition is true. The intermediate result table is the result of the previous clause. If this clause is not GROUP BY, the intermediate result table is considered a single group with no grouping columns of the previous clause of the subselect.

having-clause

►►—HAVING—*search-condition*—◄◄

Each *column-name* in *search-condition* must be one of the following:

- Unambiguously identify a grouping column of the intermediate result table
- Be specified within an aggregate function²²
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table, view, *common-table-expression*, or *nested-table-expression* that is identified in an outer subselect

A group of the intermediate result table to which the search condition is applied supplies the argument for each function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of the intermediate result table, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see Example 4 and Example 5 in “Examples of subselects” on page 657 below.

A correlated reference to a group of the intermediate result table must either identify a grouping column or be contained within an aggregate function.

When HAVING is used without GROUP BY, any expression or column name in the select list must appear within an aggregate function.

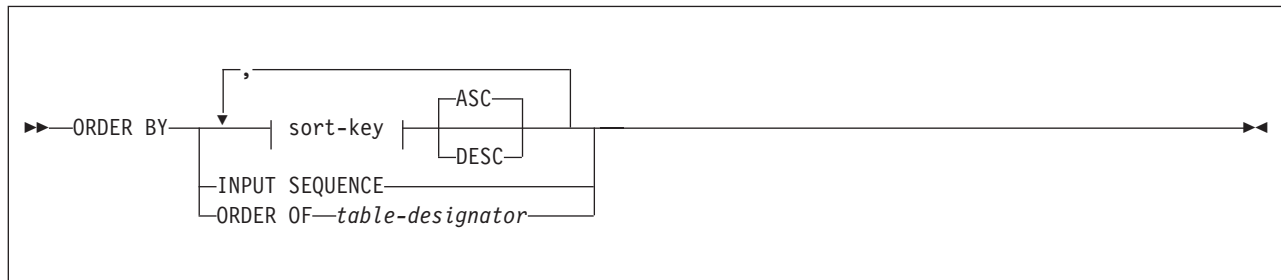
The RID built-in function and the ROW CHANGE expression cannot be specified in a HAVING clause unless they are within an aggregate function.

22. See Chapter 3, “Functions,” on page 259 for restrictions that apply to the use of aggregate functions.

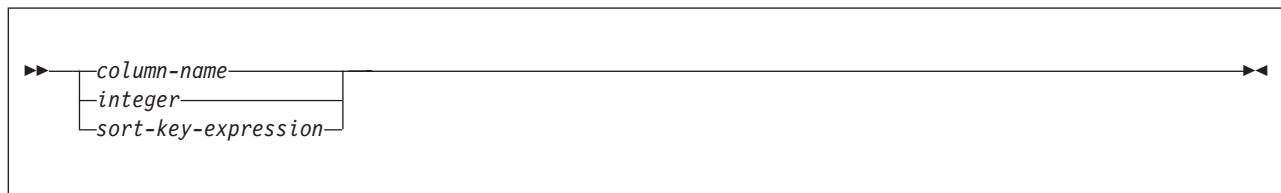
order-by-clause

The ORDER BY clause specifies an ordering of the rows of the result table.

order-by-clause



sort-key:



A subselect that contains an ORDER BY clause cannot be specified in the outermost fullselect of a view

If the subselect is not enclosed within parentheses and is not the outermost fullselect, the **ORDER BY** clause cannot be specified. The **ORDER BY** clause cannot be used in an outermost fullselect that contains a **FOR UPDATE** clause.

An **ORDER BY** clause that is specified in a subselect only effects the order of the rows that returned by the query if the subselect is the outermost fullselect, except when a nested subselect includes an **ORDER BY** clause and the outermost fullselect specifies that the ordering of the rows should be retained (by using the **ORDER OF** *table-designator* clause).

Multiple **ORDER BY** clauses can be specified in the same subselect if each clause is separated with parentheses.

INPUT SEQUENCE

Indicates that the result table reflects the input order of the rows specified in the VALUES clause of an INSERT statement. INPUT SEQUENCE ordering can be specified only when an INSERT statement is specified in a *from-clause*.

ORDER OF *table-designator*

Specifies that the same ordering of the rows for the result table that is designated by *table-designator* should be applied to the result table of the subselect (or fullselect) that contains the ORDER OF specification. There must be a table reference in the FROM clause of the subselect (or fullselect) that specifies this clause and matches *table-designator*.

sort-key

A *column-name*, *integer*, or *sort-key-expression* that specifies the value that is to be used to order the rows of the result of the subselect.

If a single *sort-key* is identified, the rows are ordered by the values of that *sort-key*. If more than one *sort-key* is identified, the rows are ordered by the values of the first *sort-key*, then by the values of the second *sort-key*, and so on. A *sort-key* cannot be a LOB or XML expression.

The result table can be ordered by a named column in the select list by specifying a *sort-key* that is an integer or the column name. The result table can be ordered by an unnamed column in the select list by specifying a *sort-key* that is an integer or, in some cases, by a *sort-key-expression* that matches the expression in the select list.

column-name

An identifier that usually identifies a column of the result table. In this case, *column-name* must be the name of a named column in the select list. If the *fullselect* includes a set operator, the column name cannot be qualified.

If the query is a *subselect*, the *column-name* can also identify a column name of a table, view, or nested table expression identified in the FROM clause, including a column that is defined as implicitly hidden. The subselect must not include any of the following:

- DISTINCT in the select list
- Aggregate functions in the select list
- GROUP BY clause

integer

Must be greater than 0 and not greater than the number of columns in the result table. The integer *n* identifies the *n*th column of the result table.

sort-key-expression

Specifies an expression with operators (that is, not simply a *column-name* or *integer*). The query to which ordering is applied must be a *subselect* to use this form of the *sort-key*.

The *sort-key-expression* cannot include an expression that is not deterministic or a function that is defined to have an external action except for the RID built-in function and the ROW CHANGE expression. Any column name in the expression must conform to the rules described Column names in sort keys. If *sort-key-expression* includes an aggregate function, the input arguments to that function must not reference a named column in the select list that is derived from an aggregate function. An expression cannot be specified if DISTINCT is used in the select list of the *subselect*.

If the *subselect* is grouped, the *sort-key-expression* might or might not be in the select list of the *subselect*. When *sort-key-expression* is not in the select list the following rules apply:

- Each expression in the ORDER BY clause must either:
 - Use one or more grouping expressions
 - Use a column name that either unambiguously identifies a grouping column of R or is specified within a aggregate function.
- Each expression in the ORDER BY clause must not contain a scalar fullselect.

ASC

Uses the values of the *sort-key* in ascending order.

ASC is the default.

DESC

Uses the values of the *sort-key* in descending order.

Ordering is performed in accordance with the comparison rules described in Chapter 2, “Language elements,” on page 47, beginning on page “Numeric comparisons” on page 114. The null value is higher than all other values. If your ordering specification does not determine a complete ordering, rows with duplicate values of the last identified *sort-key* have an arbitrary order. If you do not specify ORDER BY, the rows of the result table have an arbitrary order.

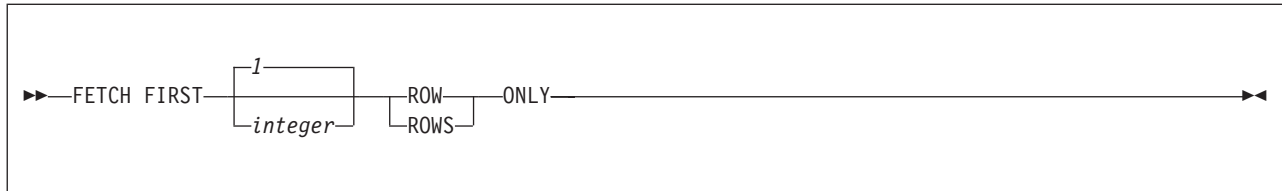
Column names in sort keys: A column name in a *sort-key* must conform to the following rules:

- If the column name is qualified, the query must be a *subselect*. The column name must unambiguously identify a column of a table, view, or nested table expression in the FROM clause of the subselect; its value is used to compute the value of the sort specification.
- If the column name is unqualified and the query is a *subselect*:
 - If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view, or nested table expression in the FROM clause of the ordering subselect.
 - If the column name is identical to one column of the result table, its value is used to compute the value of the sort specification.
 - If the column name is not identical to a column in the result table, it must unambiguously identify a column of a table, view, or nested table expression in the FROM clause of the subselect. If the column name is identical to one column of a table, view, or nested table expression in the FROM clause of the subselect, its value is used to compute the value of the sort specification.

fetch-first-clause

The FETCH FIRST clause limits the number of rows that can be fetched. It improves the performance of queries with potentially large result tables when only a limited number of rows are needed.

fetch-first-clause



The FETCH FIRST clause sets a maximum number of rows that can be retrieved. FETCH FIRST specifies that only *integer* rows should be made available to be retrieved, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond *integer* rows is handled the same way as normal end of data. The value of *integer* must be a positive integer (not zero). The default is 1.

The FETCH FIRST clause specifies an ordering of the rows of the result table. A subselect that contains a FETCH FIRST clause cannot be specified in the following objects:

- The outermost fullselect of a view
- The definition of a materialized query table

Limiting the result table to the first *n* rows can improve performance. The DB2 system will cease processing the query when it has determined the first *n* rows. If both the FETCH FIRST clause and the OPTIMIZE FOR clause are specified, the lower of the *integer* values from these clause will be used to influence the buffer size. The values are considered independently for optimization purposes. If the OPTIMIZE FOR clause is not specified, a default of OPTIMIZE FOR *integer* ROWS, where *integer* is the value that is specified in the FETCH FIRST clause, is assumed. The DB2 system uses this value for access path optimization.

Specification of the **FETCH FIRST** clause in an outermost fullselect makes the result table read-only. A read-only result table must not be referenced in an UPDATE, MERGE, or DELETE statement. The **FETCH FIRST** clause cannot be used in an outermost fullselect that contains a **FOR UPDATE** clause.

If the FETCH FIRST clause is specified in a subselect, and the subselect is not the outermost fullselect, the subselect must be enclosed in parentheses.

If both the FETCH FIRST clause and the ORDER BY clause are specified, the ordering is performed on the entire result table prior to returning the first *n* rows.

Multiple **FETCH FIRST** clauses can be specified in the same subselect if each clause is separated with parentheses.

If the FETCH FIRST clause is specified in the outermost fullselect of a SELECT statement that contains a data change statement (an INSERT, DELETE, UPDATE,

| or MERGE statement), all rows are processed by the specified data change
| statement, but only the number of rows that is specified in the FETCH FIRST
| clause are returned in the final result table.

Examples of subselects

Examples of subselects can illustrate how to use the various clauses of the subselect to construct queries.

Example 1: Show all rows of the table DSN8910.EMP.

```
SELECT * FROM DSN8910.EMP;
```

Example 2: Show the job code, maximum salary, and minimum salary for each group of rows of DSN8910.EMP with the same job code, but only for groups with more than one row and with a maximum salary greater than 50000.

```
SELECT JOB, MAX(SALARY), MIN(SALARY)
FROM DSN8910.EMP
GROUP BY JOB
HAVING COUNT(*) > 1 AND MAX(SALARY) > 50000;
```

Example 3: For each employee in department E11, get the following information from the table DSN8910.EMPPROJACT: employee number, activity number, activity start date, and activity end date. Using the CHAR function, convert the start and end dates to their USA formats. Get the needed department information from the table DSN8910.EMP.

```
SELECT EMPNO, ACTNO, CHAR(EMSTDATE,USA), CHAR(EMENDATE,USA)
FROM DSN8910.EMPPROJACT
WHERE EMPNO IN (SELECT EMPNO FROM DSN8910.EMP
                WHERE WORKDEPT = 'E11');
```

Example 4: Show the department number and maximum departmental salary for all departments whose maximum salary is less than the average salary for all employees. (In this example, the subquery would be executed only one time.)

```
SELECT WORKDEPT, MAX(SALARY)
FROM DSN8910.EMP
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                     FROM DSN8910.EMP);
```

Example 5: Show the department number and maximum departmental salary for all departments whose maximum salary is less than the average salary for employees in all other departments. (In contrast to Example 4, the subquery in this statement, containing a correlated reference, would need to be executed for each group.)

```
SELECT WORKDEPT, MAX(SALARY)
FROM DSN8910.EMP Q
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                     FROM DSN8910.EMP
                     WHERE NOT WORKDEPT = Q.WORKDEPT);
```

Example 6: For each group of employees hired during the same year, show the year-of-hire and current average salary. (This example demonstrates how to use the AS clause in a FROM clause to name a derived column that you want to refer to in a GROUP BY clause.)

```
SELECT HIREYEAR, AVG(SALARY)
FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, SALARY
      FROM DSN8910.EMP) AS NEWEMP
GROUP BY HIREYEAR;
```

Example 7: For an example of how to group the results of a query by an expression in the SELECT clause without having to retype the expression, see Example 4 for CASE expressions.

Example 8: Get the employee number and employee name for all the employees in DSN8910.EMP. Order the results by the date of hire.

```
SELECT EMPNO, FIRSTNAME, LASTNAME
FROM DSN8910.EMP
ORDER BY HIREDATE;
```

Example 9: Select all the rows from tables T1 and T2 and order the rows such that the rows from table T1 are first and are ordered by column C1, followed by the rows from T2, which are ordered by column C2. The rows of T1 are retrieved by one subselect which is connected to the results of another subselect that retrieves the rows from T2. Each subselect specifies the ordering for the rows from the referenced table. Note that both subselects need to be enclosed in parenthesis because each subselect is not the outermost fullselect.

```
(SELECT * FROM T1 ORDER BY C1)
UNION
(SELECT * FROM T2 ORDER BY C2);
```

Example 10: Specify the ORDER BY clause to order the results of a union using the second column of the result table if the union. In this example, the second ORDER BY clause applies to the results of the outermost fullselect (the result of the union) rather than to the second subselect. If the intent is to apply the second ORDER BY clause to the second subselect, the second subselect should be enclosed within parentheses as shown in Example 9.

```
(SELECT * FROM T1 ORDER BY C1)
UNION
SELECT * FROM T2 ORDER BY C2
```

Example 11: Retrieve all rows of table T1 with no specific ordering) and connect the result table to the rows of table T2, which have been ordered by the first column of table T2. The ORDER BY ORDER OF clause in the fullselect specifies that the order of the rows in the result table of the union is to be inherited by the final result.

```
SELECT *
FROM (SELECT * FROM T1
      UNION ALL
      (SELECT * FROM T2 ORDER BY 1)
     ) AS UTABLE
ORDER BY ORDER OF UTABLE;
```

Example 12: The following example uses a query to join data from a table to the result table of a nested table expression. The query uses the ORDER BY ORDER OF clause to order the rows of the result table using the order of the rows of the nested table expression.

```
SELECT T1.C1, T1.C2, TEMP.Cy, TEMP.Cx
FROM T1,
     (SELECT T2.C1, T2.C2 FROM T2 ORDER BY 2) AS TEMP(Cx, Cy)
WHERE Cy = T1.C1
ORDER BY ORDER OF TEMP;
```

Example 13: Using the EMP_ACT table, find the project numbers that have an employee whose salary is in the top three salaries for all employees.

```
SELECT EMP_ACT.EMPNO, PROJNO
FROM EMP_ACT
WHERE EMP_ACT.EMPNO IN
     (SELECT EMPLOYEE.EMPNO
      FROM EMPLOYEE
      ORDER BY SALARY DESC
      FETCH FIRST 3 ROWS ONLY);
```

Example 14: Assume that an external function named ADDYEARS exists. For a given date, the function adds a given number of years and returns a new date. (The data types of the two input parameters to the function are DATE and INTEGER.) Get the employee number and employee name for all employees who have been hired within the last 5 years.

```
SELECT EMPNO, FIRSTNAME, LASTNAME
FROM DSN8910.EMP
WHERE ADDYEARS(HIREDATE, 5) > CURRENT DATE;
```

To distinguish the different types of joins, to show nested table expressions, and to demonstrate how to combine join columns, the remaining examples use these two tables:

The PARTS table			The PRODUCTS table		
PART	PROD#	SUPPLIER	PROD#	PRODUCT	PRICE
=====	=====	=====	=====	=====	=====
WIRE	10	ACWF	505	SCREWDRIVER	3.70
OIL	160	WESTERN_CHEM	30	RELAY	7.55
MAGNETS	10	BATEMAN	205	SAW	18.90
PLASTIC	30	PLASTIK_CORP	10	GENERATOR	45.75
BLADES	205	ACE_STEEL			

Example 15: Join the tables on the PROD# column to get a table of parts with their suppliers and the products that use the parts:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS, PRODUCTS
WHERE PARTS.PROD# = PRODUCTS.PROD#;
```

or

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS INNER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

Either one of these two statements give this result:

PART	SUPPLIER	PROD#	PRODUCT
=====	=====	=====	=====
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW

Notice two things about the example:

- There is a part in the parts table (OIL) whose product (#160) is not listed in the products table. There is a product (SCREWDRIVER, #505) that has no parts listed in the parts table. Neither OIL nor SCREWDRIVER appears in the result of the join.

An outer join, however, includes rows where the values in the joined columns do not match.

- There is explicit syntax to express that this familiar join is not an outer join but an inner join. You can use INNER JOIN in the FROM clause instead of the comma. Use ON when you explicitly join tables in the FROM clause.

You can specify more complicated join conditions to obtain different sets of results. For example, eliminate the suppliers that begin with the letter A from the table of parts, suppliers, product numbers and products:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS INNER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#
AND SUPPLIER NOT LIKE 'A%';
```

The result of the query is all rows that do not have a supplier that begins with A:

PART	SUPPLIER	PROD#	PRODUCT
=====	=====	=====	=====
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY

Example 16: Join the tables on the PROD# column to get a table of all parts and products, showing the supplier information, if any.

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS FULL OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result is:

PART	SUPPLIER	PROD#	PRODUCT
=====	=====	=====	=====
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW
OIL	WESTERN_CHEM	160	(null)
(null)	(null)	(null)	SCREWDRIVER

The clause FULL OUTER JOIN includes unmatched rows from both tables. Missing values in a row of the result table are filled with nulls.

Example 17: Join the tables on the PROD# column to get a table of all parts, showing what products, if any, the parts are used in:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS LEFT OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result is:

PART	SUPPLIER	PROD#	PRODUCT
=====	=====	=====	=====
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW
OIL	WESTERN_CHEM	160	(null)

The clause LEFT OUTER JOIN includes rows from the table identified before it where the values in the joined columns are not matched by values in the joined columns of the table identified after it.

Example 18: Join the tables on the PROD# column to get a table of all products, showing the parts used in that product, if any, and the supplier.

```
SELECT PART, SUPPLIER, PRODUCTS.PROD#, PRODUCT
FROM PARTS RIGHT OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result is:

PART	SUPPLIER	PROD#	PRODUCT
=====	=====	=====	=====
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW
(null)	(null)	505	SCREWDRIVER

The clause RIGHT OUTER JOIN includes rows from the table identified after it where the values in the joined columns are not matched by values in the joined columns of the table identified before it.

Example 19: The result of Example 16 (a full outer join) shows the product number for SCREWDRIVER as null, even though the PRODUCTS table contains a product number for it. This is because PRODUCTS.PROD# was not listed in the SELECT list of the query. Revise the query using COALESCE so that all part numbers from both tables are shown.

```
SELECT PART, SUPPLIER,
       COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM, PRODUCT
FROM PARTS FULL OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

In the result, notice that the AS clause (AS PRODNUM), provides a name for the result of the COALESCE function:

PART	SUPPLIER	PRODNUM	PRODUCT
=====	=====	=====	=====
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW
OIL	WESTERN_CHEM	160	(null)
(null)	(null)	505	SCREWDRIVER

Example 20: For all parts that are used in product numbers less than 200, show the part, the part supplier, the product number, and the product name. Use a nested table expression.

```
SELECT PART, SUPPLIER, PRODNUM, PRODUCT
FROM (SELECT PART, PROD# AS PRODNUM, SUPPLIER
      FROM PARTS
      WHERE PROD# < 200) AS PARTX
LEFT OUTER JOIN PRODUCTS
ON PRODNUM = PROD#;
```

The result is:

PART	SUPPLIER	PRODNUM	PRODUCT
=====	=====	=====	=====
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
OIL	WESTERN_CHEM	160	(null)

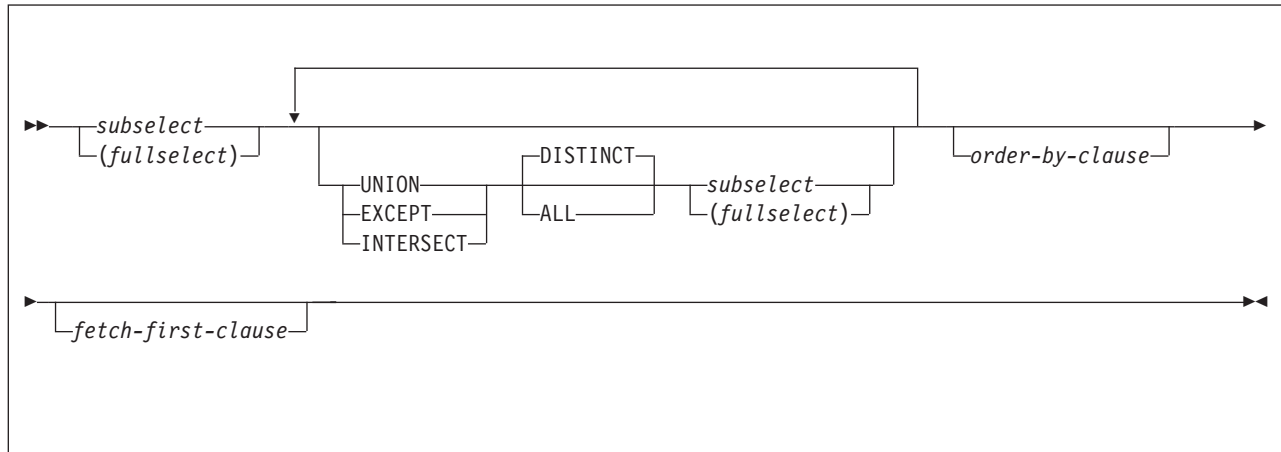
Example 21: Examples of statements with DISTINCT specified more than once in a subselect:

```
SELECT DISTINCT COUNT(DISTINCT A1), COUNT(A2)
FROM T1;

SELECT COUNT(DISTINCT A))
FROM T1
WHERE A3 > 0
HAVING AVG(DISTINCT A4) >1;
```

fullselect

The fullselect is a component of the *select-statement*, ALTER TABLE statement for the definition of a materialized query table, CREATE TABLE statement, CREATE VIEW statement, DECLARE GLOBAL TEMPORARY TABLE statement, and INSERT statement.



A fullselect that is enclosed in parentheses is called a *subquery*. For example, a subquery can be used in a search condition.

A *scalar-fullselect* is a fullselect, enclosed in parentheses, that returns a single result row and a single result column. If the result of the fullselect is no rows, then the null value is returned. An error is returned if there is more than one row in the result. For example, a scalar-fullselect can be used in the assignment clause of the DELETE, UPDATE and MERGE statements.

A *row-fullselect* is a fullselect that returns a single row. An error is returned if there is more than one row in the result. For example, a row-fullselect can be used in the assignment clause of the DELETE and UPDATE statements.

UNION, EXCEPT, or INTERSECT

The set operators, UNION, EXCEPT, and INTERSECT, correspond to the relational operators union, difference, and intersection. A *fullselect* specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect. Otherwise, the result table is derived by combining the two other result tables (R1 and R2) subject to the specified set operator.

UNION DISTINCT or UNION ALL

If UNION ALL is specified, the result consists of all rows in R1 and R2.

With UNION DISTINCT, the result is the set of all rows in either R1 or R2 with the redundant duplicate rows eliminated. In either case, each row of the result table of the union is either a row from R1 or a row from R2.

EXCEPT DISTINCT or EXCEPT ALL

If EXCEPT ALL is specified, the result consists of all rows from only R1, including significant redundant duplicate rows. With EXCEPT DISTINCT, the result consists of all rows that are only in R1, with redundant duplicate rows eliminated. In either case, each row in the result table of the difference is a row from R1 that does not have a matching row in R2.

INTERSECT DISTINCT or INTERSECT ALL

If INTERSECT ALL is specified, the result consists of all rows that are both in R1 and R2, including significant redundant duplicate rows. With INTERSECT DISTINCT, the result consists of all rows that are in both R1 and R2, with redundant duplicate rows eliminated. In either case each row of the result table of the intersection is a row that exists in both R1 and R2.

Rules for columns:

- R1 and R2 must have the same number of columns, and the data type of the *n*th column of R1 must be compatible with the data type of the *n*th column of R2.
- R1 and R2 must not include columns having a data type of CLOB, BLOB, DBCLOB, XML, or a distinct type that is based on any of these types. However, this rule is not applicable when UNION ALL is used with the set operator.
- If the *n*th column of R1 and the *n*th column of R2 have the same result column name, the *n*th column of the result table of the set operation has the same result column name. Otherwise, the *n*th column of the result table of the set operation is unnamed.
- Qualified column names cannot be used in the ORDER BY clause when the set operators are specified.

For information on the valid combinations of operand columns and the data type of the result column, see “Rules for result data types” on page 119.

Duplicate rows: Two rows are duplicates if the value in each column in the first row is equal to the corresponding value of the second row. For determining duplicates, two null values are considered equal.

The DECFLOAT data type allows for multiple bit representations of the same number. For example 2.00 and 2.0 are two numbers with the same coefficient, but different exponent values. See “Numeric comparisons” on page 114 section for more information. So if the result table of UNION contains a DECFLOAT column and multiple bit representations of the same number exist, the one returned is unpredictable.

Operator precedence: When multiple set operations are combined in an expression, set operations within parentheses are performed first. If the order is not specified by parentheses, set operations are performed from left to right with the exception that all INTERSECT operations are performed before any UNION or any EXCEPT operations.

Results of set operators: The following table illustrates the results of all set operations, with rows from result table R1 and R2 as the first two columns and the result of each operation on R1 and R2 under the corresponding column heading.

Table 78. Example of UNION, EXCEPT, and INTERSECT set operations on result tables R1 and R2.

Rows in R1	Rows in R2	Result of UNION ALL	Result of UNION DISTINCT	Result of EXCEPT ALL	Result of EXCEPT DISTINCT	Result of INTERSECT ALL	Result of INTERSECT DISTINCT
1	1	1	1	1	2	1	1
1	1	1	2	2	5	1	3
1	3	1	3	2		3	4
2	3	1	4	2		4	
2	3	1	5	4			

| Table 78. Example of UNION, EXCEPT, and INTERSECT set operations on result tables R1 and R2. (continued)

Rows in R1	Rows in R2	Result of UNION ALL	Result of UNION DISTINCT	Result of EXCEPT ALL	Result of EXCEPT DISTINCT	Result of INTERSECT ALL	Result of INTERSECT DISTINCT
2	3	2		5			
3	4	2					
4		2					
4		3					
5		3					
		3					
		3					
		3					
		4					
		4					
		4					
		5					

| **Examples of fullselects**

Example 1: A query specifies the union of result tables R1 and R2. A column in R1 has the data type CHAR(10) and the subtype BIT. The corresponding column in R2 has the data type CHAR(15) and the subtype SBCS. Hence, the column in the union has the data type CHAR(15) and the subtype BIT. Values from the first column are converted to CHAR(15) by adding five trailing blanks.

Example 2: Show all the rows from DSN8910.EMP.

```
SELECT * FROM DSN8910.EMP;
```

Example 3: Using sample tables DSN8910.EMP and DSN8910.EMPPROJACT, list the employee numbers of all employees for which either of the following statements are true:

- Their department numbers begin with 'D'.
- They are assigned to projects whose project numbers begin with 'AD'.

```
SELECT EMPNO FROM DSN8910.EMP
WHERE WORKDEPT LIKE 'D%'
UNION
SELECT EMPNO FROM DSN8910.EMPPROJACT
WHERE PROJNO LIKE 'AD'
```

The result is the union of two result tables, one formed from the sample table DSN8910.EMP, the other formed from the sample table DSN8910.EMPPROJACT. The result—a one-column table—is a list of employee numbers. Because UNION, rather than UNION ALL, was used, the entries in the list are distinct. If instead UNION ALL were used, certain employee numbers would appear in the list more than once. These would be the numbers for employees in departments that begin with 'D' while their projects begin with 'AD'.

| *Example 4:* Specify a series of unions and order the results by the first column of the final result table.

```

SELECT * FROM T1
UNION
SELECT * FROM T2
UNION
SELECT * FROM T3
ORDER BY 1;

```

Example 5: Specify a series of unions and order the results by the first column of the final result table. The first ORDER BY clause order the rows of the result of the first union by the first column of that result table. The second ORDER BY clause is applied as part of the outer fullselect and it causes the rows of the final result table to be ordered by the first column of the final result table.

```

(SELECT * FROM T1
 UNION
 SELECT * FROM T2
 ORDER BY 1)
UNION
SELECT * FROM T3
ORDER BY 1;

```

Example 6: Assume that tables T1 and T2 exist and each contain the same number of columns named C1, C2, and so on. This example of the EXCEPT operator produces all rows that are in T1 but not in T2, with duplicate rows removed:

```

(SELECT * FROM T1)
EXCEPT DISTINCT
(SELECT * FROM T2)

```

Example 7: Assume that tables T1 and T2 exist and each contain the same number of columns named C1, C2, and so on. This example of the INTERSECT operator produces all rows that are in both table T1 and table T2, with duplicate rows removed:

```

(SELECT * FROM T1)
INTERSECT DISTINCT
(SELECT * FROM T2)

```

Character conversion in set operations and concatenations

The SQL operations that combine strings include concatenation, set operators, and the IN list of an IN predicate. Within an SQL statement, concatenation combines two or more strings into a new string. Within a fullselect, set operation, or the IN list of an IN predicate combine two or more string columns resulting from the subselects into results column.

All such operations have the following in common:

- The choice of a result CCSID for the string or column
- The possible conversion of one or more of the component strings or columns to the result CCSID

For all such operations, the rules for those two actions are the same, as described in “Selecting the result CCSID” on page 667. These rules also apply to the COALESCE scalar function.

Selecting the result CCSID

The result CCSID is selected at package prepare time. The result CCSID is the CCSID of one of the operands.

Two operands: When two operands are used, the result CCSID is determined by the operand types, their CCSIDs, and their relative positions in the operation. When a CCSID is X'FFFF', the result CCSID is always X'FFFF', and no character conversions take place. When neither CCSID is X'FFFF', the rules for selecting the result CCSID are identical to the ones for string comparison. See "String comparisons" in Chapter 3, "Functions," on page 259.

Three or more operands:

If all the operands have the same CCSID, the result CCSID is the common CCSID.

If at least one of the CCSIDs has the value X'FFFF', the result CCSID also has the value X'FFFF'.

Otherwise, selection proceeds as follows:

1. The rules for a pair of operands are applied to the first two operands. This picks a "candidate" for the second step. The candidate is the operand that would furnish the result CCSID if just the first two operands were involved in the operation.
2. The rules are applied to the Step 1 candidate and the third operand, thereby selecting a second candidate.
3. If a fourth operand is involved, the rules are applied to the second candidate and fourth operand, to select a third candidate, and so on.

The process continues until all operands have been used. The remaining candidate is the one that furnishes the result CCSID. Whenever the rules for a pair are applied to a candidate and an operand, the candidate is considered to be the first operand.

Consider, for example, the following concatenation:

A CONCAT B CONCAT C

Here, the rules are first applied to the strings A and B. Suppose that the string selected as candidate is A. Then the rules are applied to A and C. If the string selected is again A, then A furnishes the result CCSID. Otherwise, C furnishes the result CCSID.

Character conversion of components: An operand of concatenation or the selected argument of the COALESCE scalar function is converted, if necessary, to the coded character set of the result string. Each string of an operand of a set operation is converted, if necessary, to the coded character set of the result column. In either case, the coded character set is the one identified by the result CCSID. Character conversion is necessary only if all of the following are true:

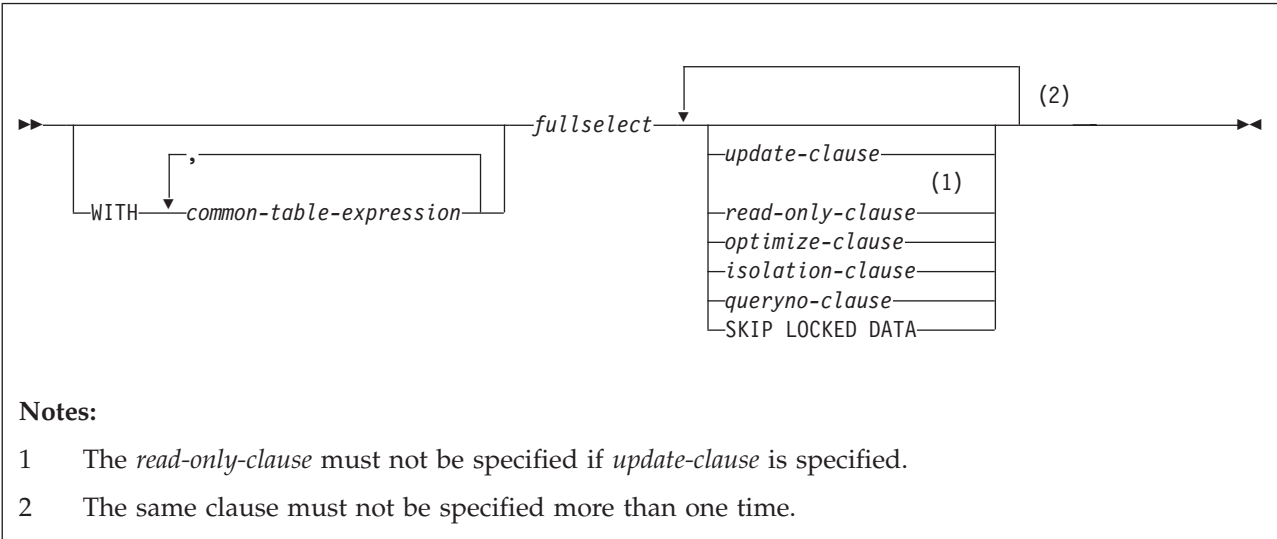
- The result and operand CCSIDs are different.
- Neither CCSID is X'FFFF' (neither string is defined as BIT data).
- The string is neither null nor empty.
- The SYSSTRINGS catalog table indicates that conversion is necessary.

An error occurs if a character of a string cannot be converted, SYSSTRINGS is used but contains no information about the CCSID pair, or DB2 cannot do the

conversion through z/OS support for Unicode. A warning occurs if a character of a string is converted to the substitution character.

select-statement

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR statement or FOR statement, prepared and then referenced in a DECLARE CURSOR statement, or directly specified in an SQLJ assignment clause. It can also be issued using SPUFI or the command line processor which causes a result table to be displayed at your terminal. In any case, the result table specified by a *select-statement* is the result of the *fullselect*.



The tables and view identified in a select statement can be at the current server or any DB2 subsystem with which the current server can establish a connection.

For local queries on DB2 for z/OS or remote queries in which the server and requester are DB2 for z/OS, if a table is encoded as ASCII or Unicode, the retrieved data is encoded in EBCDIC. For information on retrieving data encoded in ASCII or Unicode, see *DB2 Application Programming and SQL Guide*.

A select statement can implicitly or explicitly invoke user-defined functions or implicitly invoke stored procedures. This technique is known as *nesting* of SQL statements. A function or procedure is implicitly invoked in a select statement when it is invoked at a lower level. For instance, if you invoke a user-defined function from a select statement and the user-defined function invokes a stored procedure, you are implicitly invoking the stored procedure. When a SELECT statement refers to a table, any SQL statements that are implicitly invoked (as a result of nested functions or procedures) must not result in an SQL data change statement that modifies the same table.

For example, suppose that you execute this SQL statement at level 1 of nesting:
SELECT UDF1(C1) FROM T1;

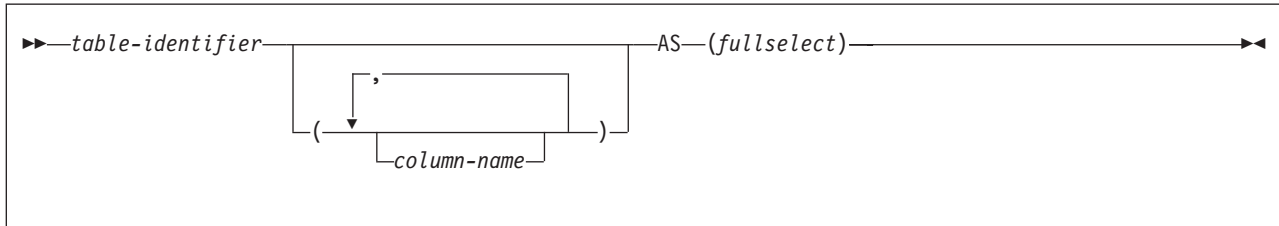
You cannot execute this SQL statement at a lower level of nesting:
INSERT INTO T1 VALUES(...);

common-table-expression

A *common table expression* defines a result table with *table-identifier* that can be referenced in any FROM clause of the *fullselect* that follows.

Multiple common table expressions can be specified following the single WITH keyword. Each specified common table expression can also be referenced by name in the FROM clause of subsequent common table expressions.

common-table-expression



If a list of columns is specified, it must consist of as many names as there are columns in the result table of the *fullselect*. Each *column-name* must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the *fullselect* used to define the common table expression.

table-identifier must be an unqualified SQL identifier, and it must be different from any other *table-identifier* in the same statement. If the common table expression is specified in an INSERT statement, the *table-identifier* must not be the same as the table or view name that is the object of the insert. If the common table expression is specified in a CREATE VIEW statement, the *table-identifier* must not be the same as the view name that is created. A common table expression *table-identifier* can be specified as a table name in any FROM clause throughout the *fullselect*.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted. A *cyclic reference* occurs when two common table expressions *dt1* and *dt2* are created such that *dt1* refers to *dt2* and *dt2* refers to *dt1*. Furthermore, a common table expression defined before cannot refer to subsequent common table expressions.

A common table expression name can only be referenced in the *select-statement*, SELECT INTO statement, INSERT statement, CREATE VIEW statement, or RETURN statement that defines it.

If a *select-statement*, SELECT INTO statement, INSERT statement, or CREATE VIEW statement that is not contained in a trigger definition refers to a unqualified table name, the following rules are applied to determine which table is actually being referenced:

- If the unqualified name corresponds to one or more common table expression names that are specified in the *select-statement*, the name identifies the common table expression that is in the innermost scope.
- Otherwise, the name identifies a persistent table, a temporary table, or a view that is present in the default schema.

If a *select-statement*, SELECT INTO statement, INSERT statement, or CREATE VIEW statement that is contained in a trigger definition refers to a unqualified table name, the following rules are applied to determine which table is actually being referenced:

- If the unqualified name corresponds to one or more common table expression names that are specified in the *select-statement*, the name identifies the common table expression that is in the innermost scope.
- If the unqualified name corresponds to a transition table name, the name identifies that transition table.
- Otherwise, the name identifies a persistent table, a temporary table, or a view that is present in the default schema.

The common table expression is also optional prior to the *fullselect* in the CREATE VIEW and INSERT statements. However, the use of common table expressions is not allowed in a INSERT within SELECT statement.

A common table expression can be used:

- In place of a view to avoid creating the view (when general use of the view is not required and positioned updates or deletes are not used)
- When the desired result table is based on host variables
- When the same result table needs to be shared in a *fullselect*
- When the result needs to be derived using recursion

If a *fullselect* of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following must be true of a recursive common table expression:

- Each *fullselect* that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed. Furthermore, the set operators must use the ALL keyword.
- The column names must be specified following the *table-name* of the common table expression.
- The first *fullselect* of the first set operator (the initialization *fullselect*) must not include a reference to the common table expression itself in any FROM clause).
- If a column name of the common table expression is referred to in the iterative *fullselect*, the data type, length, and CCSID for the column are determined based on the initialization *fullselect*. The corresponding column in the iterative *fullselect* must have the same data type and length as the data type and length determined based on the initialization *fullselect* and the CCSID must match. However, for character string types, the length of the two data types can differ. In this case, the column in the iterative *fullselect* must have a length that would always be assignable to the length determined from the initialization *fullselect*. If a column of a recursive common table expression is not used recursively in its definition, the data type, length, and CCSID for the column are determined by applying rules associated with non-recursive queries.
- Each *fullselect* that is part of the recursion cycle must not include any aggregate functions, GROUP BY clauses, or HAVING clauses. The FROM clauses of these *fullselects* can include at most one reference to a common table expression that is part of a recursion cycle.
- Subqueries (scalar or quantified) must not be part of any recursion cycles.

- Outer join must not be part of any recursion cycles.

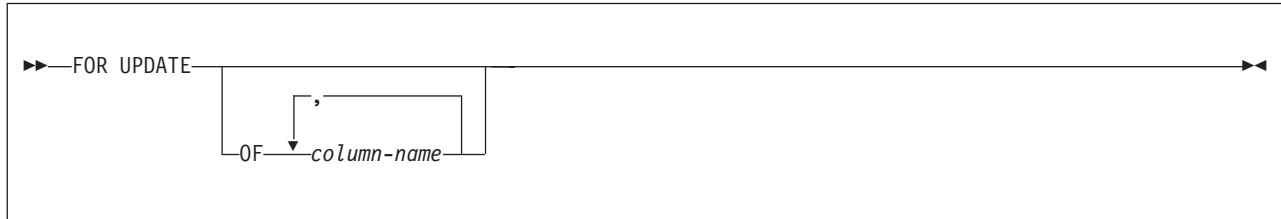
When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will terminate. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:

- In the iterative *fullselect*, an integer column incremented by a constant.
- A predicate in the WHERE clause of the iterative *fullselect* in the form of "*counter_col* < *constant*" or "*counter_col* < :*hostvar*". A warning is issued if this syntax is not found.

update-clause

The optional **FOR UPDATE** clause identifies the columns that can be updated in a later positioned **UPDATE** statement.

update-clause



Each column name must be unqualified and must identify a column of the table or view identified in the first **FROM** clause of the fullselect. The clause must not be specified if the result table of the fullselect is read-only. For a discussion of read-only result tables, see “**DECLARE CURSOR**” on page 1191. The clause must also not be specified if a created temporary table is referenced in the first **FROM** clause of the *select-statement*.

If the **FOR UPDATE** clause is specified without a *column-name* list, the columns that can be updated will include all the updatable columns of the table or view that is identified in the first **FROM** clause of the fullselect.

The declaration of a cursor referred to in a positioned **UPDATE** statement need not include an **UPDATE** clause if the **STDSQL(YES)** or **NOFOR** option is specified when the program is precompiled. For more on the subject, see “Positioned updates of columns” on page 256.

When **FOR UPDATE** is used, **FETCH** operations referencing the cursor acquire U or X locks rather than S locks when:

- The isolation level of the statement is cursor stability.
- The isolation level of the statement is repeatable read or read stability and field **U LOCK FOR RR/RS** on installation panel **DSNTIPI** is set to get U locks.
- The isolation level of the statement is repeatable read or read stability and **USE AND KEEP EXCLUSIVE LOCKS** or **USE AND KEEP UPDATE LOCKS** is specified in the SQL statement, an X lock or a U lock, respectively, is acquired at fetch time.

No locks are acquired on declared temporary tables. For a discussion of U locks and S locks, see *DB2 Performance Monitoring and Tuning Guide*.

read-only-clause

The read-only clause specifies that the result table is read-only. Therefore, the cursor cannot be referred to in positioned UPDATE or DELETE statements.

read-only-clause



►►—FOR READ ONLY—◄◄

Some result tables are read-only by nature (for example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For tables in which updates and deletes are allowed, specifying FOR READ ONLY can possibly improve the performance of FETCH operations as DB2 can do blocking and avoid exclusive locks. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, DB2 might open cursors as if the UPDATE clause was specified.

A read-only result table must not be referred to in an UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY.

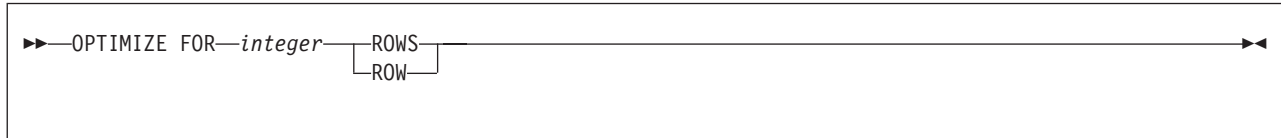
To take advantage of the possibly improved performance of FETCH operations while guaranteeing that selected data is not modified and preventing some types of deadlocks, you can specify FOR READ ONLY in combination with the optional syntax of USE AND KEEP ... LOCKS on the *isolation-clause*.

Alternative syntax and synonyms: FOR FETCH ONLY can be specified as a synonym for FOR READ ONLY.

optimize-clause

The OPTIMIZE clause requests special optimization of the *select-statement*.

optimize-clause



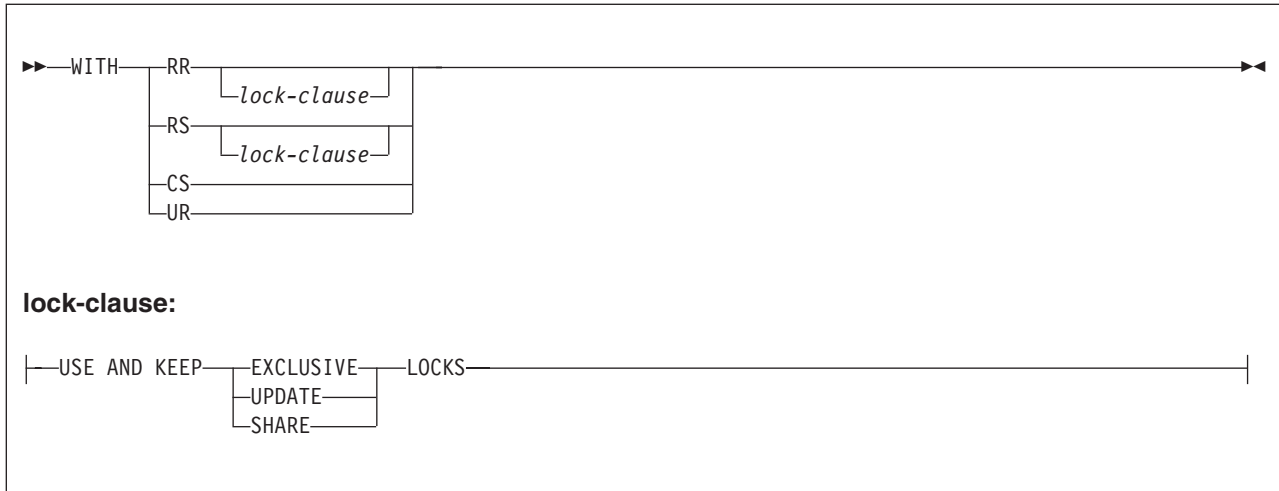
The *optimize-clause* tells DB2 to assume that the program does not intend to retrieve more than *integer* rows from the result table. Without this clause, DB2 assumes that all rows of the result table will be retrieved, unless the FETCH FIRST clause is specified. Optimizing for *integer* rows can improve performance. If this clause is omitted and the FETCH FIRST is specified, OPTIMIZE FOR *integer* ROWS is assumed, where *integer* is the value that is specified in the FETCH FIRST clause. DB2 will optimize the query based on the specified number of rows.

The clause does not limit the number of rows that can be fetched, change the result table, or change the order in which the rows are fetched. Any number of rows can be fetched, but performance can possibly degrade after *integer* fetches. In general, if you are retrieving only a few rows, specify OPTIMIZE FOR 1 ROW to influence the access path that DB2 selects. For more information about using this clause, see *DB2 Application Programming and SQL Guide*.

The value of *integer* must be a positive integer (not zero).

isolation-clause

The *isolation-clause* specifies the isolation level at which the statement is executed. (Isolation level does not apply to declared temporary tables because no locks are acquired.)

isolation-clause

RR Repeatable read

RR *lock-clause*

Repeatable read, using and keeping the type of lock that is specified in *lock-clause* on all accessed pages and rows

RS	Read stability
-----------	----------------

RS *lock-clause*

Read stability, using and keeping the type of lock that is specified in *lock-clause* on all accessed pages and rows

CS Cursor stability

UR Uncommitted read

lock-clause

Specifies the type of lock.

USE AND KEEP EXCLUSIVE LOCKS

USE AND KEEP UPDATE LOCKS

USE AND KEEP SHARE LOCKS

Specifies that DB2 is to acquire and hold X, U, or S locks, respectively.

WITH UR can be specified only if the result table of the fullselect or the SELECT INTO statement is read-only.

In an ODBC application, the `SQLSetStmtAttr` function can be used to set statement attributes that interact with the *lock-clause*. If `SQLSetStmtAttr` is invoked with a cursor's statement handle and specifying that its `SQL_ATTR_CLOSE_BEHAVIOR` is `SQL_CC_RELEASE` (locks are to be released when the cursor is closed), then irrespective of any *lock-clause*, lock used by the cursor that are not needed to protect the integrity of changed data are released. For more information on `SQLSetStmtAttr`, see *DB2 ODBC Guide and Reference*.

Although requesting an UPDATE or EXCLUSIVE LOCK can reduce concurrency, it can prevent some types of deadlocks.

The **default** isolation level of the statement depends on:

- The isolation of the package or plan that the statement is bound in
- Whether the result table is read-only

Table 79 shows the default isolation level of the statement.

Table 79. Default isolation level based on the isolation level of the package or plan and whether the result table is read-only

If package isolation is:	And plan isolation is:	And the result table is:	Then the default isolation is:
RR	Any	Any	RR
RS	Any	Any	RS
CS	Any	Any	CS
UR	Any	Read-only	UR
		Not read-only	CS
Not specified	Not specified	Any	RR
	RR	Any	RR
	RS	Any	RS
	CS	Any	CS
	UR	Read-only	UR
		Not read-only	CS

A simple way to ensure that a result table is read-only is to specify FOR READ ONLY in the SQL statement.

Alternative syntax and synonyms: KEEP UPDATE LOCKS can be specified as a synonym for USE AND KEEP EXCLUSIVE LOCKS. However, KEEP UPDATE LOCKS can be specified only if FOR UPDATE OF is specified, and it is not supported in the SELECT INTO statement.

queryno-clause

The QUERYNO clause specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO columns of the plan tables for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

queryno-clause

►►—QUERYNO—*integer*—◄◄

integer is the value to be used to identify this SQL statement in EXPLAIN output and trace records.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:

- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

SKIP LOCKED DATA

The SKIP LOCKED DATA clause specifies that rows are skipped when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only with isolation CS or RS and applies only to row level or page level locks.



Diagram illustrating the SKIP LOCKED DATA clause in a query statement. The clause is shown as a horizontal line with arrows at both ends, indicating it is a clause that can be applied to a query.

SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is repeatable read (WITH RR) or uncommitted read (WITH UR). The default isolation level of the statement depends on the isolation level of the package or plan with which the statement is bound, and whether the result table is read-only.

Examples of select statements

Examples of SELECT statements.

Example 1: Select all the rows from DSN8910.EMP.

```
SELECT * FROM DSN8910.EMP;
```

Example 2: Select all the rows from DSN8910.EMP, arranging the result table in chronological order by date of hiring.

```
SELECT * FROM DSN8910.EMP  
ORDER BY HIREDATE;
```

Example 3: Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the table DSN8910.EMP. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)  
FROM DSN8910.EMP  
GROUP BY WORKDEPT  
ORDER BY 2;
```

Example 4: Change various salaries, bonuses, and commissions in the table DSN8910.EMP. Confine the changes to employees in departments D11 and D21. Use positioned updates to do this with a cursor named UP_CUR. Use a FOR UPDATE clause in the cursor declaration to indicate that all updatable columns are updated. Below is the declaration for a PL/I program.

```
EXEC SQL DECLARE UP_CUR CURSOR FOR  
SELECT WORKDEPT, EMPNO, SALARY, BONUS, COMM  
FROM DSN8910.EMP  
WHERE WORKDEPT IN ('D11','D21')  
FOR UPDATE;
```

Beginning where the cursor is declared, all updatable columns would be updated. If only specific columns needed to be updated, such as only the salary column, the FOR UPDATE clause could be used to specify the salary column (FOR UPDATE OF SALARY).

Example 5: Find the maximum, minimum, and average bonus in the table DSN8910.EMP. Execute the statement with uncommitted read isolation, regardless of the value of ISOLATION with which the plan or package containing the statement is bound. Assign 13 as the query number for the SELECT statement.

```
EXEC SQL  
SELECT MAX(BONUS), MIN(BONUS), AVG(BONUS)  
INTO :MAX, :MIN, :AVG  
FROM DSN8910.EMP  
WITH UR  
QUERYNO 13;
```

If bind option EXPLAIN(YES) is specified, rows are inserted into the plan table. The value used for the QUERYNO column for these rows is 13.

Example 6: The cursor declaration shown below is in a PL/I program. In the query within the declaration, X.RMT_TAB is an alias for a table at some other DB2. Hence, when the query is used, it is processed using DRDA access. See "Distributed data" on page 31.

The declaration indicates that no positioned updates or deletes will be done with the query's cursor. It also specifies that the access path for the query be optimized for the retrieval of at most 50 rows. Even so, the program can retrieve more than

50 rows from the result table, which consists of the entire table identified by the alias. However, when more than 50 rows are retrieved, performance could possibly degrade.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT * FROM X.RMT_TAB
  OPTIMIZE FOR 50 ROWS
  FOR READ ONLY;
```

The `FETCH FIRST` clause could be used instead of the `OPTIMIZE FOR` clause to ensure that only 50 rows are retrieved as in the following example:

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT * FROM X.RMT_TAB
  FETCH FIRST 50 ROWS ONLY;
```

Example 7: Assume that table `DSN8810.EMP` has 1000 rows and you wish to see the first five `EMP_ROWID` values that were inserted into `DSN8810.EMP_PHOTO_RESUME`.

```
EXEC SQL DECLARE CS1 CURSOR FOR
  SELECT EMP_ROWID
  FROM FINAL TABLE (INSERT INTO DSN8910.EMP_PHOTO_RESUME (EMPNO)
                     SELECT EMPNO FROM DSN8810.EMP)
  FETCH FIRST 5 ROWS ONLY;
```

All 1000 rows are inserted into `DSN8810.EMP_PHOTO_RESUME`, but only the first five are returned.

Chapter 5. Statements

This section contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements.

Table 80. SQL statements

SQL statement	Function	Topic
ALLOCATE CURSOR	Defines and associates a cursor with a result set locator variable	"ALLOCATE CURSOR" on page 696
ALTER DATABASE	Changes the description of a database	"ALTER DATABASE" on page 698
ALTER FUNCTION (external)	Changes the description of a user-defined external scalar or table function	"ALTER FUNCTION (external)" on page 701
ALTER FUNCTION (SQL scalar)	Changes the description of an SQL scalar function	"ALTER FUNCTION (SQL scalar)" on page 719
ALTER INDEX	Changes the description of an index	"ALTER INDEX" on page 725
ALTER PROCEDURE (external)	Changes the description of an external procedure	"ALTER PROCEDURE (external)" on page 741
ALTER PROCEDURE (SQL - external)	Changes the description of an external SQL procedure	"ALTER PROCEDURE (SQL - external)" on page 752
ALTER PROCEDURE (SQL - native)	Changes the description of or defines additional versions for a native SQL procedure	"ALTER PROCEDURE (SQL - native)" on page 758
ALTER SEQUENCE	Changes the description of a sequence	"ALTER SEQUENCE" on page 781
ALTER STOGROUP	Changes the description of a storage group	"ALTER STOGROUP" on page 786
ALTER TABLE	Changes the description of a table	"ALTER TABLE" on page 789
ALTER TABLESPACE	Changes the description of a table space	"ALTER TABLESPACE" on page 841
ALTER TRUSTED CONTEXT	Changes the description of a trusted context	"ALTER TRUSTED CONTEXT" on page 855
ALTER VIEW	Regenerates a view	"ALTER VIEW" on page 867
ASSOCIATE LOCATORS	Gets the result set locator value for each result set returned by a stored procedure	"ASSOCIATE LOCATORS" on page 868
BEGIN DECLARE SECTION	Marks the beginning of a host variable declaration section	"BEGIN DECLARE SECTION" on page 872
CALL	Calls a stored procedure	"CALL" on page 874
CLOSE	Closes a cursor	"CLOSE" on page 885
COMMENT	Replaces or adds a comment to the description of an object	"COMMENT" on page 887
COMMIT	Ends a unit of recovery and commits the database changes made by that unit of recovery	"COMMIT" on page 896
CONNECT	Connects the process to a server	"CONNECT" on page 899
CREATE ALIAS	Defines an alias	"CREATE ALIAS" on page 905

Table 80. SQL statements (continued)

SQL statement	Function	Topic
CREATE AUXILIARY TABLE	Defines an auxiliary table for storing LOB data	"CREATE AUXILIARY TABLE" on page 908
CREATE DATABASE	Defines a database	"CREATE DATABASE" on page 912
CREATE FUNCTION (external scalar)	Defines a user-defined external scalar function	"CREATE FUNCTION (external scalar)" on page 916
CREATE FUNCTION (external table)	Defines a user-defined external table function	"CREATE FUNCTION (external table)" on page 940
CREATE FUNCTION (sourced)	Defines a user-defined function that is based on an existing scalar or aggregate function	"CREATE FUNCTION (sourced)" on page 958
CREATE FUNCTION (SQL scalar)	Defines a user-defined SQL scalar function	"CREATE FUNCTION (SQL scalar)" on page 972
CREATE GLOBAL TEMPORARY TABLE	Defines a created temporary table	"CREATE GLOBAL TEMPORARY TABLE" on page 982
CREATE INDEX	Defines an index on a table	"CREATE INDEX" on page 988
CREATE PROCEDURE (external)	Defines an external stored procedure	"CREATE PROCEDURE (external)" on page 1016
CREATE PROCEDURE (SQL - external)	Defines an external SQL procedure	"CREATE PROCEDURE (SQL - external)" on page 1035
CREATE PROCEDURE (SQL - native)	Defines a native SQL procedure	"CREATE PROCEDURE (SQL - native)" on page 1046
CREATE ROLE	Defines a role	"CREATE ROLE" on page 1065
CREATE SEQUENCE	Defines a sequence	"CREATE SEQUENCE" on page 1066
CREATE STOGROUP	Defines a storage group	"CREATE STOGROUP" on page 1074
CREATE SYNONYM	Defines an alternate name for a table or view	"CREATE SYNONYM" on page 1077
CREATE TABLE	Defines a table	"CREATE TABLE" on page 1079
CREATE TABLESPACE	Defines a table space, which includes allocating and formatting the table space	"CREATE TABLESPACE" on page 1128
CREATE TRIGGER	Defines a trigger	"CREATE TRIGGER" on page 1151
CREATE TRUSTED CONTEXT	Defines a trusted context	"CREATE TRUSTED CONTEXT" on page 1167
CREATE TYPE	Defines a type (user-defined data type)	"CREATE TYPE" on page 1177
CREATE VIEW	Defines a view of one or more tables or views	"CREATE VIEW" on page 1184
DECLARE CURSOR	Defines an SQL cursor	"DECLARE CURSOR" on page 1191

Table 80. SQL statements (continued)

SQL statement	Function	Topic
DECLARE GLOBAL TEMPORARY TABLE	Defines a declared temporary table	"DECLARE GLOBAL TEMPORARY TABLE" on page 1202
DECLARE STATEMENT	Declares names used to identify prepared SQL statements	"DECLARE STATEMENT" on page 1216
DECLARE TABLE	Provides the programmer and the precompiler with a description of a table or view	"DECLARE TABLE" on page 1217
DECLARE VARIABLE	Defines a CCSID for a host variable	"DECLARE VARIABLE" on page 1221
DELETE	Deletes one or more rows from a table	"DELETE" on page 1224
DESCRIBE CURSOR	Puts information about the result set associated with a cursor into a descriptor	"DESCRIBE CURSOR" on page 1238
DESCRIBE INPUT	Puts information about the input parameter markers of a prepared statement into a descriptor	"DESCRIBE INPUT" on page 1240
DESCRIBE OUTPUT	Describes the result columns of a prepared statement	"DESCRIBE OUTPUT" on page 1243
DESCRIBE PROCEDURE	Puts information about the result sets returned by a stored procedure into a descriptor	"DESCRIBE PROCEDURE" on page 1250
DESCRIBE TABLE	Describes the columns of a table or view	"DESCRIBE TABLE" on page 1253
DROP	Deletes objects	"DROP" on page 1256
END DECLARE SECTION	Marks the end of a host variable declaration section	"END DECLARE SECTION" on page 1273
EXCHANGE	Exchanges data between the specified base table and an associated clone table	"EXCHANGE" on page 1274
EXECUTE	Executes a prepared SQL statement	the EXECUTE statement
EXECUTE IMMEDIATE	Prepares and executes an SQL statement	"EXECUTE IMMEDIATE" on page 1280
EXPLAIN	Obtains information about how an SQL statement would be executed	"EXPLAIN" on page 1283
FETCH	Positions the cursor, returns data, or both positions the cursor and returns data	"FETCH" on page 1290
FREE LOCATOR	Removes the association between a LOB locator variable and its value	"FREE LOCATOR" on page 1316
GET DIAGNOSTICS	Provides diagnostic information about the last SQL statement that was executed	"GET DIAGNOSTICS" on page 1317
GRANT	The GRANT statement grants privileges to authorization IDs. There is a separate form of the statement for each of the classes of privilege.	"GRANT" on page 1333
GRANT (collection privileges)	Grants authority to create a package in a collection	"GRANT (collection privileges)" on page 1336
GRANT (database privileges)	Grants privileges on a database	"GRANT (database privileges)" on page 1337
GRANT (type or JAR file privileges)	Grants the usage privilege on a type (user-defined data type) or a JAR file	"GRANT (type or JAR file privileges)" on page 1359

Table 80. SQL statements (continued)

SQL statement	Function	Topic
GRANT (function or procedure privileges)	Grants privileges on a user-defined function or a stored procedure	"GRANT (function or procedure privileges)" on page 1340
GRANT (package privileges)	Grants authority to bind, execute, or copy a package	"GRANT (package privileges)" on page 1345
GRANT (plan privileges)	Grants authority to bind or execute an application plan	"GRANT (plan privileges)" on page 1348
GRANT (schema privileges)	Grants privileges on a schema	"GRANT (schema privileges)" on page 1349
GRANT (sequence privileges)	Grants privileges on a user-defined sequence	"GRANT (sequence privileges)" on page 1351
GRANT (system privileges)	Grants system privileges	"GRANT (system privileges)" on page 1352
GRANT (table or view privileges)	Grants privileges on a table or view	"GRANT (table or view privileges)" on page 1355
GRANT (use privileges)	Grants authority to use specified buffer pools, storage groups, or table spaces	"GRANT (use privileges)" on page 1361
HOLD LOCATOR	Allows a LOB locator variable to retain its association with its value beyond a unit of work	"HOLD LOCATOR" on page 1363
INCLUDE	Inserts declarations into a source program	"INCLUDE" on page 1365
INSERT	Inserts one or more rows into a table	"INSERT" on page 1367
LABEL	Replaces or adds a label on the description of a table, view, alias, or column	"LABEL" on page 1384
LOCK TABLE	Locks a table or table space partition in shared or exclusive mode	"LOCK TABLE" on page 1386
I MERGE	Updates and/or inserts one or more rows of a table	"MERGE" on page 1388
OPEN	Opens a cursor	"OPEN" on page 1400
PREPARE	Prepares an SQL statement (with optional parameters) for execution	"PREPARE" on page 1405
REFRESH TABLE	Refreshes the data in a materialized query table	"REFRESH TABLE" on page 1422
RELEASE (connection)	Places one or more connections in the release pending status	"RELEASE (connection)" on page 1424
RELEASE SAVEPOINT	Releases a savepoint and any subsequently set savepoints within a unit of recovery	"RELEASE SAVEPOINT" on page 1427
I RENAME	Renames an existing table or index	"RENAME" on page 1428
REVOKE	Revokes privileges from authorization IDs. There is a separate form of the statement for each of the classes of privilege	"REVOKE" on page 1432
REVOKE (collection privileges)	Revokes authority to create a package in a collection	"REVOKE (collection privileges)" on page 1437
REVOKE (database privileges)	Revokes privileges on a database	"REVOKE (database privileges)" on page 1439
REVOKE (type or JAR file privileges)	Revokes the usage privilege on a type (user-defined data type) or a JAR file	"REVOKE (type or JAR file privileges)" on page 1461

Table 80. SQL statements (continued)

SQL statement	Function	Topic
REVOKE (function or procedure privileges)	Revokes privileges on a user-defined function or a stored procedure	"REVOKE (function or procedure privileges)" on page 1442
REVOKE (package privileges)	Revokes authority to bind, execute, or copy a package	"REVOKE (package privileges)" on page 1447
REVOKE (plan privileges)	Revokes authority to bind or execute an application plan	"REVOKE (plan privileges)" on page 1449
REVOKE (schema privileges)	Revokes privileges on a schema	"REVOKE (schema privileges)" on page 1451
REVOKE (sequence privileges)	Revokes privileges on a user-defined sequence	"REVOKE (sequence privileges)" on page 1453
REVOKE (system privileges)	Revokes system privileges	"REVOKE (system privileges)" on page 1455
REVOKE (table or view privileges)	Revokes privileges on a table or view	"REVOKE (table or view privileges)" on page 1458
REVOKE (use privileges)	Revokes authority to use specified buffer pools, storage groups, or table spaces	"REVOKE (use privileges)" on page 1463
ROLLBACK	Ends a unit of recovery and backs out the changes to the database made by that unit of recovery, or partially rolls back the changes to a savepoint within the unit of recovery	"ROLLBACK" on page 1465
SAVEPOINT	Sets a savepoint within a unit of recovery	"SAVEPOINT" on page 1468
SELECT	Specifies the SELECT statement of the cursor	"SELECT" on page 1470
SELECT INTO	Specifies a result table of no more than one row and assigns the values to host variables	"SELECT INTO" on page 1471
SET CONNECTION	Establishes the database server of the process by identifying one of its existing connections	"SET CONNECTION" on page 1475
SET CURRENT APPLICATION ENCODING SCHEME	Assigns a value to the CURRENT APPLICATION ENCODING SCHEME special register	"SET CURRENT APPLICATION ENCODING SCHEME" on page 1477
SET CURRENT DEBUG MODE	Assigns a value to the CURRENT DEBUG MODE special register	"SET CURRENT DEBUG MODE" on page 1478
SET CURRENT DECFLOAT ROUNDING MODE	Assigns a value to the CURRENT DECFLOAT ROUNDING MODE special register	"SET CURRENT DECFLOAT ROUNDING MODE" on page 1480
SET CURRENT DEGREE	Assigns a value to the CURRENT DEGREE special register	"SET CURRENT DEGREE" on page 1482
SET CURRENT LOCALE LC_CTYPE	Assigns a value to the CURRENT LOCALE LC_CTYPE special register	"SET CURRENT LOCALE LC_CTYPE" on page 1483
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Assigns a value to the CURRENT MAINTAINED TABLE TYPES FOR MAINTAINED TABLE TYPES special register	"SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION" on page 1485
SET CURRENT OPTIMIZATION HINT	Assigns a value to the CURRENT OPTIMIZATION HINT special register	"SET CURRENT OPTIMIZATION HINT" on page 1487
SET CURRENT PACKAGE PATH	Assigns a value to the CURRENT PACKAGE PATH special register	"SET CURRENT PACKAGE PATH" on page 1488

Table 80. SQL statements (continued)

SQL statement	Function	Topic
SET CURRENT PACKAGESET	Assigns a value to the CURRENT PACKAGESET special register	"SET CURRENT PACKAGESET" on page 1492
SET CURRENT PRECISION	Assigns a value to the CURRENT PRECISION special register	"SET CURRENT PRECISION" on page 1494
SET CURRENT REFRESH AGE	Assigns a value to the CURRENT REFRESH AGE special register	"SET CURRENT REFRESH AGE" on page 1495
SET CURRENT ROUTINE VERSION	Assigns a value to the CURRENT ROUTINE VERSION special register	"SET CURRENT ROUTINE VERSION" on page 1497
SET CURRENT RULES	Assigns a value to the CURRENT RULES special register	"SET CURRENT RULES" on page 1499
SET CURRENT SQLID	Assigns a value to the CURRENT SQLID special register	"SET CURRENT SQLID" on page 1500
SET ENCRYPTION PASSWORD	Assign a value for the ENCRYPTION PASSWORD and, optionally, a hint for the password	"SET host-variable assignment" on page 1504
SET host-variable Assignment	Assigns values to host variables	"SET host-variable assignment" on page 1504
SET PATH	Assigns a value to the CURRENT PATH special register	"SET PATH" on page 1507
SET SCHEMA	Assigns a value to the CURRENT SCHEMA special register	"SET SCHEMA" on page 1510
SET transition-variable Assignment	Assigns values to transition variables	"SET transition-variable assignment" on page 1513
SIGNAL	Signals an error or warning condition and optionally returns the specified message text	"SIGNAL statement" on page 1582
TRUNCATE	Deletes all rows from a table	"TRUNCATE" on page 1517
UPDATE	Updates the values of one or more columns in one or more rows of a table	"UPDATE" on page 1521
VALUES	Provides a way to invoke a user-defined function from a trigger	"VALUES" on page 1536
VALUES INTO	Assigns values to host variables	"VALUES INTO" on page 1537
WHENEVER	Defines actions to be taken on the basis of SQL return codes	"WHENEVER" on page 1539

How SQL statements are invoked

SQL statements are invoked in different ways depending on whether the statement is an executable or nonexecutable statement or the *select-statement*.

The SQL statements are classified as *executable* or *nonexecutable*. The description of each statement includes a heading on invocation that indicates whether or not the statement is executable.

Executable statements can be invoked in the following ways:

- Embedded in an application program
- Dynamically prepared and executed
- Dynamically prepared and executed using DB2 ODBC function calls

- Issued interactively

Depending on the statement, you can use some or all of these methods. The section on invocation in the description of each statement tells you which methods can be used.

A *nonexecutable statement* can only be embedded in an application program.

The *select-statement* is an additional SQL statement construct. (See “select-statement” on page 669.) It is used in a different way from other statements.

A *select-statement* can be invoked in the following ways:

- Included in DECLARE CURSOR and implicitly executed by OPEN
- Dynamically prepared, referred to in DECLARE CURSOR, and implicitly executed by OPEN
- Dynamically executed (no PREPARE required) using a DB2 ODBC function call
- Issued interactively

The first two methods are called, respectively, the *static* and the *dynamic* invocation of *select-statement*.

Embedding a statement in an application program

You can include SQL statements in a source program that will be submitted to the precompiler. Such statements are said to be *embedded* in the application program. An embedded statement can be placed anywhere in the application program where a host language statement is allowed. Each embedded statement must be preceded by a keyword (or keywords) to indicate that the statement is an SQL statement.

- In C and COBOL, each embedded statement must be preceded by the keywords EXEC SQL.
- In Java, each embedded statement must be preceded by the keywords #sql.
- In REXX, each embedded statement must be preceded by the keyword EXECSQL.

Executable statements: An executable statement embedded in an application program is executed every time a statement of the host language would be executed if specified in the same place. (Thus, for example, a statement within a loop is executed every time the loop is executed, and a statement within a conditional construct is executed only when the condition is satisfied.)

An embedded statement can contain references to host variables. A host variable referred to in this way can be used in one of two ways:

As input

The current value of the host variable is used in the execution of the statement.

As output

The variable is assigned a new value as a result of executing the statement.

In particular, all references to host variables in expressions and predicates are effectively replaced by current values of the variables; that is, the variables are used as input. The treatment of other references is described individually for each statement.

The successful or unsuccessful execution of the statement is indicated by setting the SQLCODE and SQLSTATE fields in the SQLCA.²³ You must therefore follow all executable statements by a test of SQLCODE or SQLSTATE. Alternatively, you can use the WHENEVER statement (which is itself nonexecutable) to change the flow of control immediately after the execution of an embedded statement.

Nonexecutable statements: An embedded nonexecutable statement is processed only by the precompiler. The precompiler reports any errors encountered in the statement. The statement is never executed, and acts as a no-operation if placed among executable statements of the application program. Therefore, do not follow such statements with a test of an SQL return code.

Dynamic preparation and execution

Your application program can dynamically build an SQL statement in the form of a character string placed in a host variable. In general, the statement is built from some data available to the application program (for example, input from a workstation).

In non-Java languages, the statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE and executed by means of the (embedded) statement EXECUTE, as described in *DB2 Application Programming and SQL Guide*. Alternatively, you can use the (embedded) statement EXECUTE IMMEDIATE to prepare and execute a statement in one step. In Java, the statement can be prepared for execution by means of the Statement, PreparedStatement, and CallableStatement classes, and executed by means of their respective execute() methods.

The statement can also be prepared by calling the DB2 ODBC SQLPrepare function and then executed by calling the DB2 ODBC SQLExecute function. In both cases, the application does not contain an embedded PREPARE or EXECUTE statement. You can execute the statement, without preparation, by passing the statement to the DB2 ODBC SQLExecDirect function. *DB2 ODBC Guide and Reference* describes the APIs supported with this interface.

A statement that is going to be prepared must not contain references to host variables. It can instead contain parameter markers. (See Parameter markers in the description of the PREPARE statement for rules concerning parameter markers.) When the prepared statement is executed, the parameter markers are effectively replaced by current values of the host variables specified in the EXECUTE statement. (See the EXECUTE statement for rules concerning this replacement.) After it is prepared, a statement can be executed several times with different values of host variables. Parameter markers are not allowed in the SQL statement prepared and executed using EXECUTE IMMEDIATE.

In non-Java languages, the successful or unsuccessful execution of the statement is indicated by the values returned in the SQLCODE and SQLSTATE fields in the SQLCA after the EXECUTE (or EXECUTE IMMEDIATE) statement. You should check the fields as described above for embedded statements. In Java, the successful or unsuccessful execution of the statement is handled by Java Exceptions.

23. SQLCODE and SQLSTATE cannot be in the SQLCA when the precompiler option STDSQL(YES) is in effect. See "SQL standard language" on page 255.

As explained in “Authorization IDs and dynamic SQL” on page 64, the DYNAMICRULES behavior in effect determines the privilege set that is used for authorization checking when dynamic SQL statements are processed. The following table summarizes those privilege sets. (See Table 6 on page 64 for a list of the DYNAMICRULES bind option values that determine which behavior is in effect).

Table 81. DYNAMICRULES behaviors and authorization checking

DYNAMICRULES behavior	Privilege set
Run behavior	<p>The union of the set of privileges held by each authorization ID of the process if the dynamically prepared statement is other than an ALTER, CREATE, DROP, GRANT, RENAME, or REVOKE statement.</p> <p>The privileges that are held by the SQL authorization ID of the process or the role of the primary authorization ID (if the process is running in a trusted context that is defined with the ROLE AS OBJECT OWNER clause), if the dynamic SQL statement is a CREATE, GRANT, or REVOKE statement.</p>
Bind behavior	The privileges that are held by the primary authorization ID of the owner of the package or plan.
Define behavior	The privileges that are held by the authorization ID of the stored procedure or user-defined function owner (definer).
Invoke behavior	The privileges that are held by the authorization ID of the stored procedure or user-defined function invoker. However, if the invoker is the primary authorization ID of the process or the CURRENT SQLID value, secondary authorization IDs are also checked if they are needed for the required authorization. Therefore, in that case, the privilege set is the union of the set of privileges that are held by each authorization ID or role (if running in a trusted context).

Static invocation of a SELECT statement

A SELECT statement can be invoked statically in different ways.

You can include a SELECT statement as a part of the (nonexecutable) statement DECLARE CURSOR. Such a statement is executed every time you open the cursor by means of the (embedded) statement OPEN. After the cursor is open, you can retrieve the result table a row at a time by successive executions of the (embedded) SQL FETCH statement.

If the application is using DB2 ODBC, the SELECT statement is first prepared with the SQLPrepare function call. It is then executed with the SQLExecute function call. Data is then fetched with the SQLFetch function call. The application does not explicitly open the cursor.

The SELECT statement used in this way can contain references to host variables. These references are effectively replaced by the values that the variables have at the moment of executing OPEN.

The successful or unsuccessful execution of the SELECT statement is indicated by the values returned in the SQLCODE and SQLSTATE fields in the SQLCA after the OPEN. You should check the fields as described above for embedded statements.

If the application is using DB2 ODBC, the successful execution of the SELECT statement is indicated by the return code from the SQLExecute function call. If necessary, the application can retrieve the SQLCA by calling the SQLGetSQLCA function.

Dynamic invocation of a SELECT statement

Your application program can dynamically build a SELECT statement in the form of a character string placed in a host variable. In general, the statement is built from some data available to the application program (for example, a query obtained from a terminal).

The statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE, and referred to by a (nonexecutable) statement DECLARE CURSOR. The statement is then executed every time you open the cursor by means of the (embedded) statement OPEN. After the cursor is open, you can retrieve the result table a row at a time by successive executions of the (embedded) SQL FETCH statement.

The SELECT statement used in that way must not contain references to host variables. It can instead contain parameter markers. (See “Notes” in “PREPARE” on page 1405 for rules concerning parameter markers.) The parameter markers are effectively replaced by the values of the host variables specified in the OPEN statement. (See “OPEN” on page 1400 for rules concerning this replacement.)

The successful or unsuccessful execution of the SELECT statement is indicated by the values returned in the SQLCODE and SQLSTATE fields in the SQLCA after the OPEN. You should check the fields as described above for embedded statements.

Interactive invocation

An SQL statement submitted to DB2 from a terminal is said to be issued interactively.

IBM relational database management systems allow you to enter SQL statements from a terminal. DB2 for z/OS provides SPUFI to prepare and execute SQL statements. Other products are also available. A statement entered in this way is said to be issued interactively.

A statement issued interactively must not contain parameter markers or references to host variables, because these make sense only in the context of an application program. For the same reason, there is no SQLCA involved.

SQL diagnostics information

DB2 uses a diagnostics area to store status information and diagnostics information about the execution of an executable SQL statement.

When an SQL statement other than GET DIAGNOSTICS or *compound-statement* is processed, the current diagnostics area is cleared before processing the SQL statement. As each SQL statement is processed, information about the execution of that SQL statement is recorded in the current diagnostics area as one or more completion conditions or exception conditions.

A completion condition indicates that the SQL statement completed successfully, completed with a warning condition, or completed with a not found condition. An exception condition indicates that the statement was not successful. The GET

DIAGNOSTICS statement can be executed in most languages to return conditions and other information about the previously executed SQL statement from the diagnostics area. Additionally, the condition information is provided through language specific mechanisms:

- For SQL procedures, see “Notes” on page 1546 in “SQL-procedure-statement” on page 1546 for information about detecting and processing error and warning conditions.
- For host language applications, see “Detecting and processing error and warning conditions in host language applications.”

Related reference

“GET DIAGNOSTICS” on page 1317

Detecting and processing error and warning conditions in host language applications

Errors and warnings conditions in host language applications can be checked by using the SQLCODE or SQLSTATE host variables or by using the SQLCA.

Each host language provides a mechanism for handling diagnostic information.

- In Assembler, C, COBOL, Fortran, and PL/I, an application program that contains executable SQL statements must provide at least one of the following:
 - A structure named SQLCA, which can be provided by using the INCLUDE SQLCA statement
 - A stand-alone CHAR(5) (CHAR(6) in C) variable named *SQLSTATE* (*SQLSTT* in Fortran)
 - A stand-alone integer variable named *SQLCODE* (*SQLCOD* in Fortran)
- In Java, for error conditions, the *getSQLState* method of the JDBC *SQLException* class can be used to get the *SQLSTATE* and the *getErrorCode* method can be used to get the *SQLCODE*.
- In REXX, an SQLCA is provided automatically.

Whether you define stand-alone *SQLCODE* and *SQLSTATE* host variables or an SQLCA in your program depends on the DB2 precompiler option you choose.

If the application is using DB2 ODBC and it calls the *SQLGetSQLCA* function, it need only include an SQLCA. Otherwise, all notification of success or errors is specified with return codes for the various function calls.

When you specify *STDSQL(YES)*, which indicates conformance to the SQL standard, you should not define an SQLCA. The stand-alone variable for *SQLCODE* must be a valid host variable in the *DECLARE SECTION* of a program. It can also be declared outside of the *DECLARE SECTION* when no variable is defined for *SQLSTATE*. The stand-alone variable for *SQLSTATE* must be declared in the *DECLARE SECTION*. It must not be declared as an element of a structure.

Use a stand-alone *SQLSTATE* to conform with the SQL 2003 Core standard. When you specify *STDSQL(NO)*, which indicates conformance to DB2 rules, you must include an SQLCA explicitly to have access to the *SQLSTATE* and *SQLCODE* information.

SQLSTATE

DB2 sets *SQLSTATE* after each SQL statement (other than *GET DIAGNOSTICS* or a compound statement) is executed. DB2 returns values that conform to the error

specification in the SQL standard. Thus, application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE.

SQLSTATE provides application programs with common codes for common error conditions (the values of SQLSTATE are product-specific if the error or warning is product-specific). Furthermore, SQLSTATE is designed so that application programs can test for specific errors or classes of errors. The coding scheme is the same for all IBM implementations of SQL. The SQLSTATE values are based on the SQLSTATE specifications contained in the SQL standard. Error messages and the tokens that are substituted for variables in error messages are associated with SQLCODE values, not SQLSTATE values.

In the case of a LOOP statement, the SQLSTATE is set after the END LOOP portion of the LOOP statement completes. With the REPEAT statement, the SQLSTATE is set after the UNTIL and END REPEAT portions of the REPEAT statement completes.

If the application is using DB2 ODBC, the SQLSTATE returned conforms to the ODBC Version 2.0 specification.

SQLCODE

The SQLCODE is also set by DB2 after each SQL statement is executed as follows:

DB2 conforms to the SQL standard as follows:

- If SQLCODE = 0 and SQLWARN0 is blank, execution was successful.
- If SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.
- If SQLCODE > 0 and not = 100, execution was successful with a warning.
- If SQLCODE = 0 and SQLWARN0 = 'W', execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

In the case of a LOOP statement, the SQLSTATE is set after the END LOOP portion of the LOOP statement completes. With the REPEAT statement, the SQLSTATE is set after the UNTIL and END REPEAT portions of the REPEAT statement completes.

The SQL standard does not define the meaning of any other specific positive or negative values of SQLCODE, and the meaning of these values is not the same in all implementations of SQL.

If the application is using DB2 ODBC, an SQLCODE is only returned if the application issues the SQLGetSQLCA function.

SQL comments

Static SQL statements can include host language or SQL comments. Dynamic SQL statements can include SQL comments. There are two types of SQL comments, simple comments and bracketed comments.

simple comments

Simple comments are introduced with two consecutive hyphens (--) and end with the end of a line. The following rules apply to the use of simple comments:

- The two hyphens must be on the same line and must not be separated by a space.
- Simple comments can be started whenever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Simple comments cannot be continued to the next line.
- In COBOL, the hyphen must be preceded by a space.

bracketed comments

Bracketed comments are introduced with /* and end with */. The following rules apply to the use of bracketed comments:

- The /* must be on the same line and not separated by a space.
- The */ must be on the same line and not separated by a space.
- Bracketed comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Bracketed comments can be continued to the next line.
- Bracketed comments can be nested within other bracketed comments. However, nested bracketed comments are not supported by DSNTEP2, DSNTEP4, SPUFI, or the command line processor.
- Bracketed comments are not allowed in static SQL statements in a COBOL, Fortran, or Assembler program.

Example: This example shows how to include comments in an SQL statement within a C program. The example uses both simple and bracketed comments:

```
EXEC SQL
  CREATE VIEW PRJ_MAXPER          --projects with most support personnel
  /*
    * Returns number and name of the project
    */
  AS SELECT PROJNO, PROJNAME      -- number and name of project
     FROM DSN8910.PROJ
  /*
    * E21 is the systems support dept code
    */
  WHERE DEPTNO = 'E21'           -- systems support dept code
  AND PRSTAFF > 1;
```

For information about host language comments, refer to *DB2 Application Programming and SQL Guide*.

ALLOCATE CURSOR

The `ALLOCATE CURSOR` statement defines a cursor and associates it with a result set locator variable.

Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot be issued interactively.

Authorization

None required.

Syntax

►►—`ALLOCATE`—*cursor-name*—`CURSOR FOR RESULT SET`—*rs-locator-variable*—◄◄

Description

cursor-name

Names the cursor. The name must not identify a cursor that has already been declared in the source program.

CURSOR FOR RESULT SET *rs-locator-variable*

Specifies a result set locator variable that has been declared in the application program according to the rules for declaring result set locator variables.

The result set locator variable must contain a valid result set locator value, as returned by the `ASSOCIATE LOCATORS` or `DESCRIBE PROCEDURE SQL` statement. The value of the result set locator variable is used at the time the cursor is allocated. Subsequent changes to the value of the result set locator have no affect on the allocated cursor. The result set locator value must not be the same as a value used for another cursor allocated in the source program.

Notes

Dynamically prepared ALLOCATE CURSOR statements: The `EXECUTE` statement with the `USING` clause must be used to execute a dynamically prepared `ALLOCATE CURSOR` statement. In a dynamically prepared statement, references to host variables are represented by parameter markers (question marks). In the `ALLOCATE CURSOR` statement, *rs-locator-variable* is always a host variable. Thus, for a dynamically prepared `ALLOCATE CURSOR` statement, the `USING` clause of the `EXECUTE` statement must identify the host variable whose value is to be substituted for the parameter marker that represents *rs-locator-variable*.

You cannot prepare an `ALLOCATE CURSOR` statement with a statement identifier that has already been used in a `DECLARE CURSOR` statement. For example, the following SQL statements are invalid because the `PREPARE` statement uses `STMT1` as an identifier for the `ALLOCATE CURSOR` statement and `STMT1` has already been used for a `DECLARE CURSOR` statement.

```
DECLARE CURSOR C1 FOR STMT1;
PREPARE STMT1 FROM          INVALID
  'ALLOCATE C2 CURSOR FOR RESULT SET ?';
```

Rules for using an allocated cursor: The following rules apply when you use an allocated cursor:

- You cannot open an allocated cursor with the OPEN statement.
- You can close an allocated cursor with the CLOSE statement. Closing an allocated cursor closes the associated cursor defined in the stored procedure.
- You can allocate only one cursor to each result set.

The life of an allocated cursor: A rollback operation, an implicit close, or an explicit close destroy allocated cursors. A commit operation destroys allocated cursors that are not defined WITH HOLD by the stored procedure. Destroying an allocated cursor closes the associated cursor defined in the stored procedure.

Considerations for scrollable cursors: Following an ALLOCATE CURSOR statement, a GET DIAGNOSTICS statement can be used to get the attributes of the cursor such as the following information (for more information, see “GET DIAGNOSTICS” on page 1317):

- DB2_SQL_ATTR_CURSOR_HOLD. Whether the cursor was defined with the WITH HOLD attribute.
- DB2_SQL_ATTR_CURSOR_SCROLLABLE. Scrollability of the cursor.
- DB2_SQL_ATTR_CURSOR_SENSITIVITY. Effective sensitivity of the cursor.
The sensitivity information can be used by applications (such as an ODBC driver) to determine what type of FETCH (INSENSITIVE or SENSITIVE) to issue for a cursor defined as ASENSITIVE.
- DB2_SQL_ATTR_CURSOR_ROWSET. Whether the cursor can be used to access rowsets.
- DB2_SQL_ATTR_CURSOR_TYPE. Whether a cursor type is forward-only, static, or dynamic.

In addition, if subsystem parameter DISABSCL is set to NO, a subset of the above information is returned in the SQLCA:

- The scrollability of the cursor is in SQLWARN1.
- The sensitivity of the cursor is in SQLWARN4.
- The effective capability of the cursor is in SQLWARN5.

Example

The statement in the following example is assumed to be in a PL/I program.

Define and associate cursor C1 with the result set locator variable *LOC1* and the related result set returned by the stored procedure:

```
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :LOC1;
```

ALTER DATABASE

The ALTER DATABASE statement changes the description of a database at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

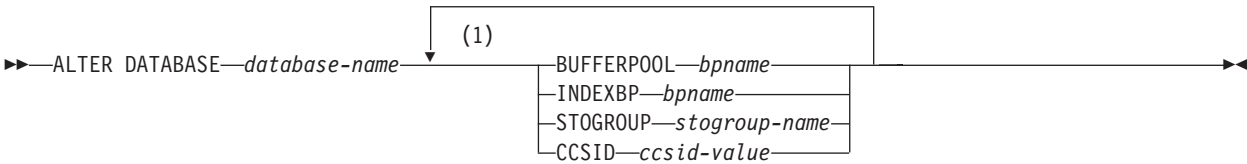
The privilege set that is defined below must include at least one of the following:

- The DROP privilege on the database
- Ownership of the database
- DBADM or DBCTRL authority for the database
- SYSADM or SYSCTRL authority

If the database is implicitly created, the privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

Syntax



Notes:

- 1 The same clause must not be specified more than one time.

Description

DATABASE *database-name*

Identifies the database that is to be altered. The name must identify a database that exists at the current server and must not identify an implicitly created system database.

BUFFERPOOL *bpname*

Identifies the default buffer pool for the table spaces within the database. It does not apply to table spaces that already exist within the database.

If the database is a work file database, 8 KB and 16 KB buffer pools cannot be specified.

See “Naming conventions” on page 51 for more details about *bpname*.

INDEXBP *bpname*

Identifies the default buffer pool for the indexes within the database. It does not apply to indexes that already exist within the database. The name can identify a 4 KB, 8 KB, 16 KB, or 32 KB buffer pool. See “Naming conventions” on page 51 for more details about *bpname*.

STOGROUP *stogroup-name*

Identifies the storage group to be used, as required, as a default storage group to support DASD space requirements for table spaces and indexes within the database. It does not apply to table spaces and indexes that already exist within the database.

CCSID *ccsid-value*

Identifies the default CCSID for table spaces within the database. It does not apply to existing table spaces in the database. *ccsid-value* must identify a CCSID value that is compatible with the current value of the CCSID for the database. “Notes” contains a list that shows the CCSID to which a given CCSID can be altered.

CCSID cannot be specified for a work file database.

Notes

Altering the CCSID: The ability to alter the default CCSID enables you to change to a CCSID that supports the Euro symbol. You can only convert between specific CCSIDs that do and do not define the Euro symbol. In most cases, the code point that supports the Euro symbol replaces an existing code point, such as the International Currency Symbol (ICS).

Changing a CCSID can be disruptive to the system and requires several steps. For each encoding scheme of a system (ASCII, EBCDIC, and Unicode), DB2 supports SBCS, DBCS, and mixed CCSIDs. Therefore, the CCSIDs for all databases and all table spaces within an encoding scheme should be altered at the same time. Otherwise, unpredictable results might occur.

The recommended method for changing the CCSID requires that the data be unloaded and reloaded. See *DB2 Installation Guide* for the steps needed to change the CCSID, such as running an installation CLIST to modify the CCSID data in DSNHDECP, when to drop and recreate views, and when to rebind invalidated plans and packages.

The following lists show the CCSIDs that can be converted. The second CCSID in each pair is the CCSID with the Euro symbol. The CCSID can be changed from the CCSID that does not support the Euro symbol to the CCSID that does, and vice versa. For example, if the current CCSID is 500, it can be changed to 1148.

EBCDIC CCSIDs

37	1140
273	1141
277	1142
278	1143
280	1144
284	1145
285	1146
297	1147
500	1148
871	1149

ASCII CCSIDs	

850	858
874	4970
1250	5346
1251	5347
1252	5348
1253	5349
1254	5350
1255	5351
1256	5352
1257	5353

Example

Change the default buffer pool for both table spaces and indexes within database ABCDE to BP2.

```
ALTER DATABASE ABCDE
  BUFFERPOOL BP2
  INDEXBP BP2;
```

ALTER FUNCTION (external)

The ALTER FUNCTION statement changes the description of a user-defined external scalar function or external table function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- Ownership of the function
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

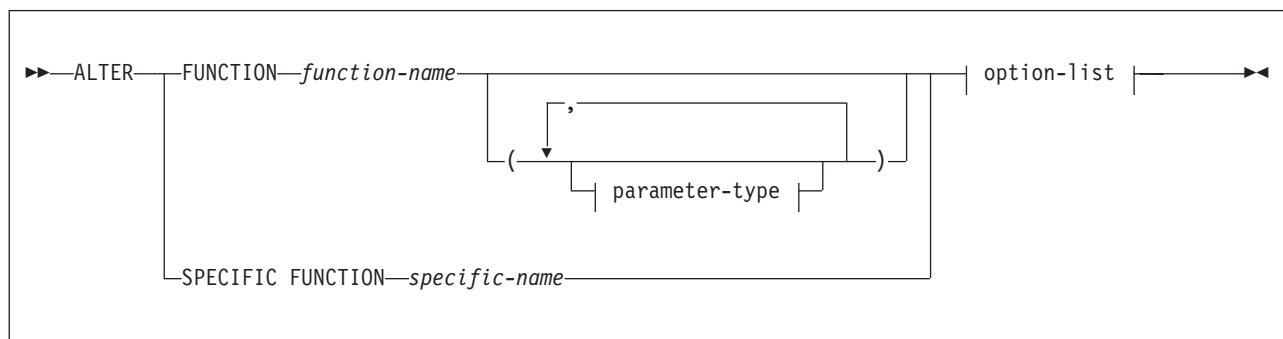
If the statement is dynamically prepared, the privilege set is the privileges that are held by the authorization IDs of the process. The specified function name can include a schema name (a qualifier). However, if the schema name is not the same as one of these authorization IDs, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- An authorization ID of the process has the ALTERIN privilege on the schema.

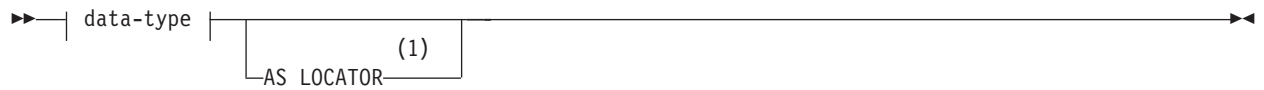
If the environment in which the function is to be run is being changed, the authorization ID must have authority to use the WLM environment specified. The required authorization is obtained from an external security product, such as RACF.

For *external scalar functions*, when **LANGUAGE** is **JAVA** and a *jar-name* is specified in the **EXTERNAL NAME** clause, the privilege set must include USAGE on the JAR file.

Syntax



parameter-type:



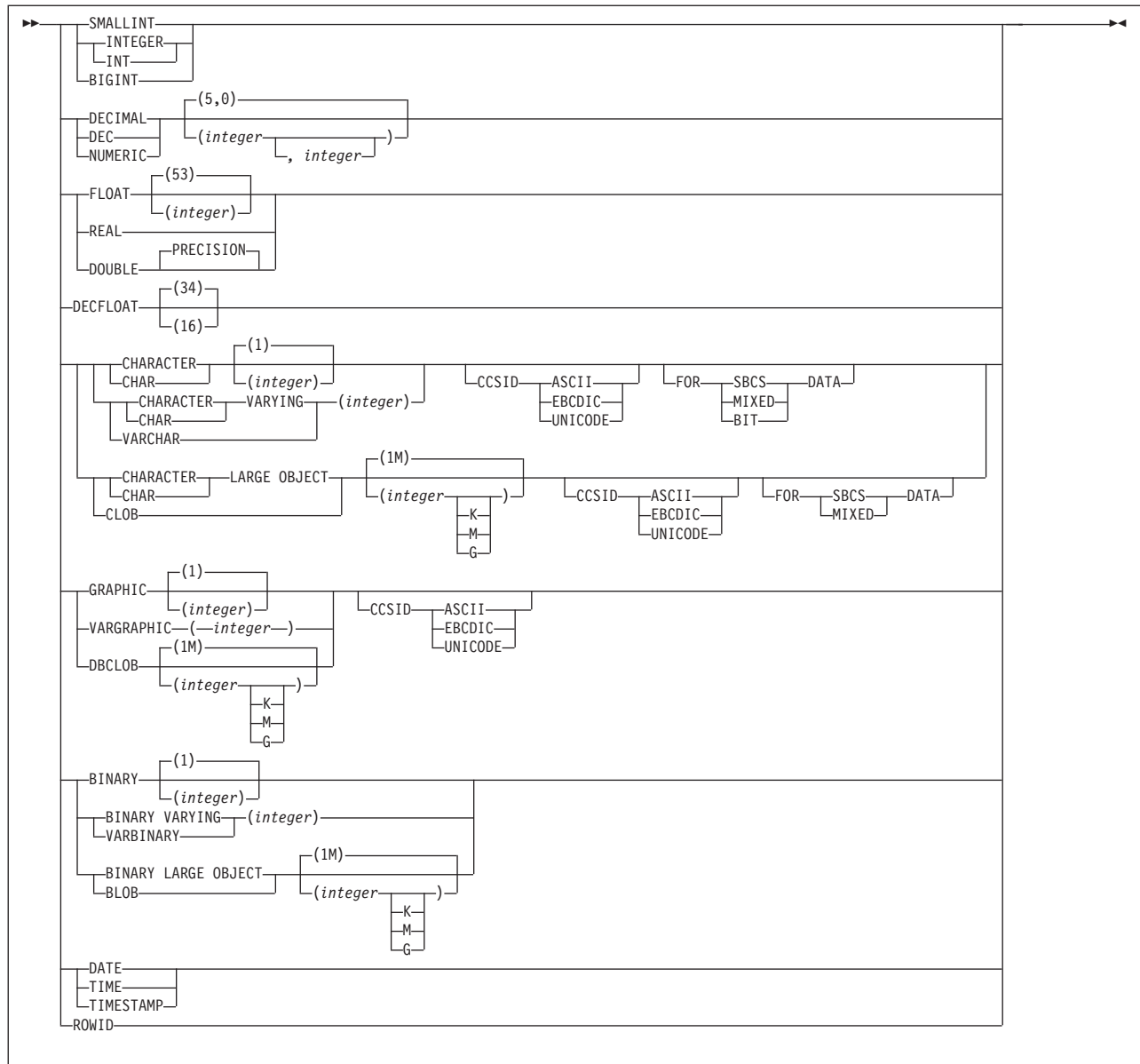
Notes:

- 1 **AS LOCATOR** can be specified only for a LOB data type or a distinct type based on a LOB data type.

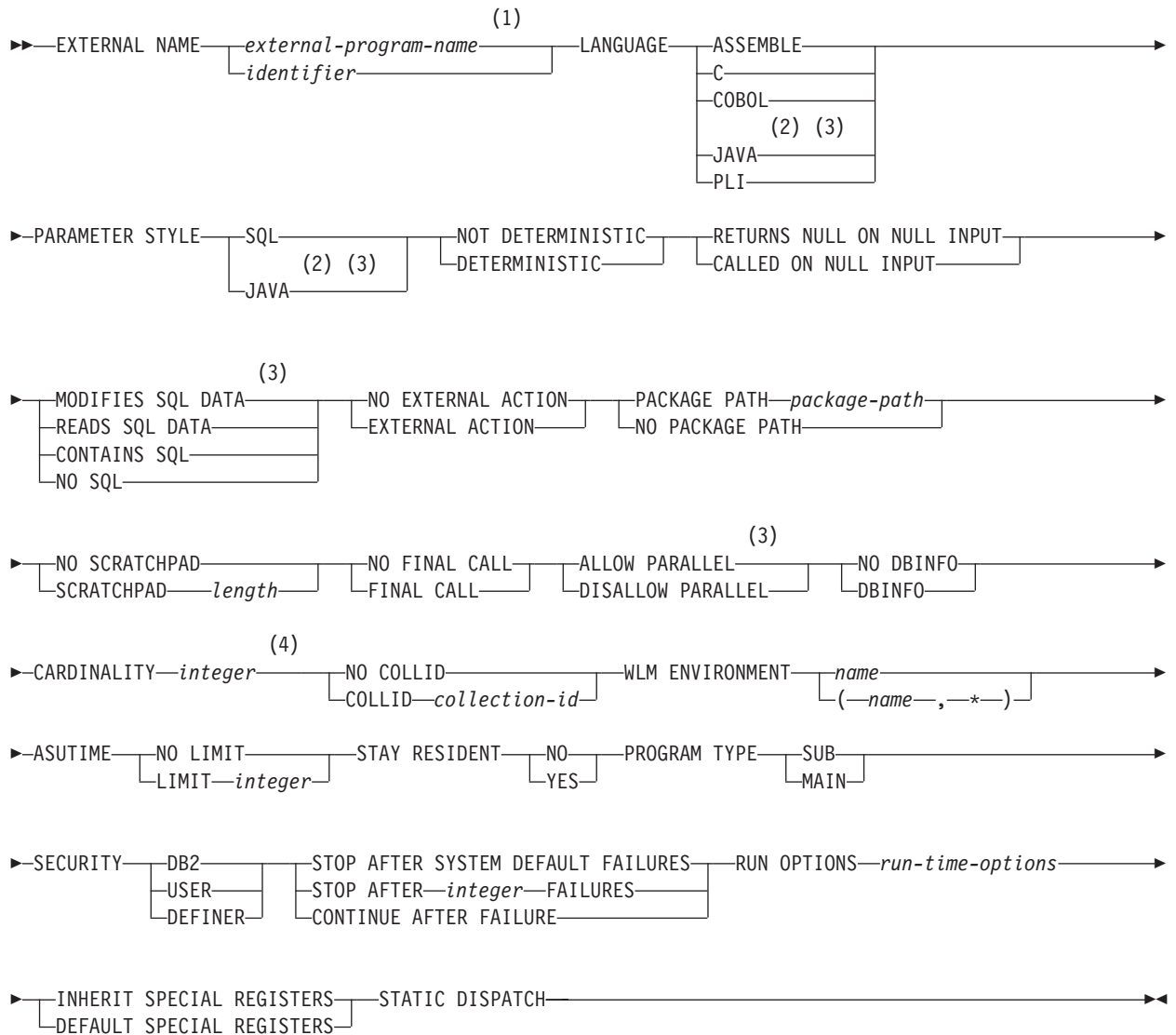
data-type:



built-in-type:



option-list: (Specify options in any order. Specify at least one option. Do not specify the same option more than once.)



Notes:

- 1 If **LANGUAGE** is **JAVA**, **EXTERNAL NAME** must be specified with a valid *external-java-routine-name*.
- 2 When **LANGUAGE JAVA** is specified, **PARAMETER STYLE JAVA** must also be specified. When **PARAMETER STYLE JAVA** is specified, **LANGUAGE JAVA** must also be specified.
- 3 **LANGUAGE JAVA**, **PARAMETER STYLE JAVA**, **MODIFIES SQL DATA**, and **ALLOW PARALLEL** are not supported for *external table functions*.
- 4 **CARDINALITY** is not supported for *external scalar functions*.

```

graph LR
    subgraph Command_Line_Format [ ]
        direction LR
        JarName[jar-name:] --- MethodName[method-name] --- MethodSignature[method-signature]
    end

```

The diagram illustrates the components of a `jar-id`. It consists of a long horizontal line. Underneath the left part of this line is a bracket labeled `schema-name.`. Underneath the right part of the line is the label `jar-id`.

Diagram illustrating the structure of a Java identifier. The identifier is composed of several parts: a package-id, a class-name, a method-id, and a version number. The package-id is enclosed in brackets and followed by a dot. The class-name is enclosed in brackets and followed by a dot. The method-id is enclosed in brackets and followed by a dot. The version number is enclosed in brackets and followed by a dot. The diagram shows the sequence of these components in a Java identifier.

- 1 The slash (/) is supported for compatibility with previous releases of DB2 for z/OS.
- 2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

One of the following three clauses identifies the function to be changed.

Identifies the external function by its function name. <i>function-name</i> must identify a function that exists at the current server. The function must be a user-defined external function, and there must be exactly one function with <i>function-name</i> in the schema.
--

The function can have any number of input parameters. If the schema does not contain a function with *function-name* or contains more than one function with this name, an error occurs.

Identifies the external function by its function signature, which uniquely identifies the function.

<i>function-name</i>	Identifies the function by its name.
----------------------	--------------------------------------

If *function-name()* is specified, the function that is identified must have zero parameters.

(parameter-type,...)

Identifies the number of input parameters of the function and their data types.

The data type of each parameter must match the data type that was specified in the CREATE FUNCTION statement for the parameter in the corresponding position. The number of data types and the logical concatenation of the data types are used to uniquely identify the function. Therefore, you cannot change the number of parameters or the data types of the parameters.

For data types that have a length, precision, or scale attribute, you can use a set of empty parentheses, specify a value, or accept the default values:

If the function was defined with a table parameter (the **LIKE TABLE name AS LOCATOR** clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

- Empty parentheses indicate that DB2 is to ignore the attribute when determining whether the data types match.

For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34).

FLOAT cannot be specified with empty parentheses because its parameter value indicates different data types (REAL or DOUBLE).

- If you use a specific value for a length, precision, or scale attribute, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

The specific value for FLOAT(*n*) does not have to exactly match the defined value of the source function because $1 \leq n \leq 21$ indicates REAL and $22 \leq n \leq 53$ indicates DOUBLE. Matching is based on whether the data type is REAL or DOUBLE.

- If length, precision, or scale is not explicitly specified and empty parentheses are not specified, the default length of the data type is implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the **FOR subtype DATA** clause or the **CCSID** clause is optional. Omission of either clause indicates that DB2 is to ignore the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

See “CREATE FUNCTION” on page 915 for more information on the specification of the parameter list.

A function with the function signature must exist in the explicitly or implicitly specified schema.

SPECIFIC FUNCTION specific-name

Identifies the external function by its specific name. A function with the specific name must exist in the schema.

The following clauses change the description of the function that has been identified to be changed.

EXTERNAL NAME *external-program-name* **or** *identifier*

Identifies the user-written code (program) that runs when the function is invoked.

If **LANGUAGE** is **JAVA**, *external-program-name* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *external-program-name* values are specified, the total length of all of them must not be greater than 1305 bytes and they must be separated by a space or a line break. Do not specify a JAR file for a Java function for which **NO SQL** is in effect.

An *external-java-routine-name* contains the following parts:

jar-name

Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER option of the BIND subcommand for a package or plan when the package or plan was created or last changed. The schema name can also be the authorization ID in the QUALIFIER option of the CREATE PROCEDURE or ALTER PROCEDURE statement for a native SQL procedure when the procedure was created or last changed. If the QUALIFIER is not specified, the schema name is the owner of the package, plan, or native SQL procedure.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the ALTER FUNCTION statement is processed.

If *jar-name* is not specified, the function is loaded from the class file directly. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed function.

method-name

Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method IDs are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what method IDs can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id

Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.UserFuncs." The Java virtual machine looks in the directory "/myPacks/UserFuncs/" for the classes.

class-id

Identifies the class identifier of the Java object.

method-id

Identifies the method identifier with the Java class to be invoked.

method-signature

Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the user-defined function involves any input or output parameters that can be NULL. When the function that is being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements that are specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the ALTER FUNCTION statement.

For other values of **LANGUAGE**, the value must conform to the naming conventions for load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

LANGUAGE

Specifies the application programming language in which the function is written. All programs must be designed to run in IBM's Language Environment[®] environment.

ASSEMBLE

The function is written in Assembler.

C The function is written in C or C++.

COBOL

The function is written in COBOL, including the object-oriented language extensions.

JAVA

The user-defined function is written in Java and is executed in the Java virtual machine. If the ALTER FUNCTION statement results in changing **LANGUAGE** to **JAVA**, **PARAMETER STYLE JAVA** and an **EXTERNAL NAME** clause must be specified to provide the appropriate values. When **LANGUAGE JAVA** is specified, the **EXTERNAL NAME** clause must also be specified with a valid *external-java-routine-name* and **PARAMETER STYLE** must be specified with **JAVA**.

Do not specify **LANGUAGE JAVA** when **SCRATCHPAD**, **FINAL CALL**, **DBINFO**, **PROGRAM TYPE MAIN**, or **RUN OPTIONS** is specified. Do not specify **LANGUAGE JAVA** for a table function.

PLI

The function is written in PL/I.

PARAMETER STYLE

Specifies the linkage convention that the function program uses to receive input parameters from and pass return values to the invoking SQL statement.

SQL

Specifies the parameter passing convention that supports passing null values both as input and for output. The parameters that are passed between the invoking SQL statement and the function include:

- Input parameters. The first n parameters are the input parameters that are specified for the function.
- Result parameters. For an external scalar function, a parameter for the result of the function. For an external table function, the next m parameters that are specified on the RETURNS TABLE clause of the CREATE statement that defined the function.
- Input parameter indicator variables. n parameters for the indicator variables for the input parameters.
- Result parameter indicator variables. For an external scalar function, a parameter for the indicator variable for the result of the function that is specified on the RETURNS clause of the CREATE statement that defined the function. For an external table function, m parameters for the indicator variables of the result columns of the function that are specified on the RETURNS TABLE clause of the CREATE statement that defined the function.
- The SQLSTATE to be returned to DB2.
- The qualified name of the function.
- The specific name of the function.
- The SQL diagnostic string to be returned to DB2.
- The scratchpad, if **SCRATCHPAD** is specified.
- The call type. For an external scalar function, the call type is passed only if **FINAL CALL** is specified. The call type is always passed for an external table function.
- The DBINFO structure, if **DBINFO** is specified.

JAVA

Indicates that the user-defined function uses a convention for passing parameters that conforms to the Java and SQLJ specifications. If the ALTER FUNCTION statement results in changing **LANGUAGE** to **JAVA**, **PARAMETER STYLE JAVA** and an **EXTERNAL NAME** clause must be specified to provide the appropriate values. **PARAMETER STYLE JAVA** can be specified only if **LANGUAGE** is **JAVA**. **JAVA** must be specified for **PARAMETER STYLE** when **LANGUAGE** is **JAVA**.

Do not specify **PARAMETER STYLE JAVA** for a table function.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC

The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT or SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers, or any function that contains SQL statements.

Some SQL functions that invoke functions that are not deterministic can receive incorrect results if the function is executed by parallel tasks. Specify the **DISALLOW PARALLEL** clause for these functions.

If a view or a materialized query table definition refers to the function, the function cannot be changed to **NOT DETERMINISTIC**. To change the function, drop any views or materialized query tables that refer to the function first.

DETERMINISTIC

The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for **SELECT** or **SQL** data change statements that refer to this function. If applicable, specify **DETERMINISTIC** to prevent non-optimal access paths from being chosen for **SQL** statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON NULL INPUT

The function is not called if any of the input arguments is null. For an external scalar function, the result is the null value. For an external table function, the result is an empty table, which is a table with no rows.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null argument values. For an external scalar function, the function can return a null or nonnull value. For an external table function, the function can return an empty table, depending on its logic.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL

Specifies which **SQL** statements, if any, can be executed in the function or any routine that is called from this function.

MODIFIES SQL DATA

Specifies that the function can execute any **SQL** statement except the statements that are not supported in functions. Do not specify **MODIFIES SQL DATA** when **ALLOW PARALLEL** is in effect.

READS SQL DATA

Specifies that the function can execute statements with a data access indication of **READS SQL DATA**, **CONTAINS SQL**, or **NO SQL**. The function cannot execute **SQL** statements that modify data.

CONTAINS SQL

Specifies that the function can execute only **SQL** statements with a data classification of **CONTAINS SQL** or **NO SQL**. **SQL** statements that neither read nor modify **SQL** data can be executed by the function. Statements that are not supported in any function return a different error.

NO SQL

Specifies that the function can execute only **SQL** statements with a data access classification of **NO SQL**. Do not specify **NO SQL** for a Java function that uses a JAR file.

NO EXTERNAL ACTION or EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

Because DB2 uses the RRS attachment for external functions, DB2 can participate in two-phase commit with any other resource manager that uses RRS. For resource managers that do not use RRS, there is no coordination of commit or rollback operations on non-DB2 resources.

NO EXTERNAL ACTION

The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT or SQL data change statements that refer to this function. If applicable, specify **NO EXTERNAL ACTION** to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

EXTERNAL ACTION

The function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the **DISALLOW PARALLEL** clause for functions that do not work correctly with parallelism.

If you specify **EXTERNAL ACTION**, DB2:

- Materializes the views and table expressions in SELECT or SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify **EXTERNAL ACTION** if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

If a view or a materialized query table definition refers to the function, the function cannot be changed to **EXTERNAL ACTION**. To change the function, drop any views or materialized query tables that refer to the function first.

DB2 does not verify that the function program is consistent with the specification of **EXTERNAL ACTION** or **NO EXTERNAL ACTION**.

NO PACKAGE PATH or PACKAGE PATH *package-path*

Identifies the package path to use when the function is run. This is the list of the possible package collections into which the DBRM that is associated with the function is bound.

NO PACKAGE PATH

Specifies that the list of package collections for the function is the same as

the list of package collections for the program that invokes the function. If the program that invokes the function does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*

Specifies a list of package collections, in the same format as used in the SET CURRENT PACKAGE PATH statement.

If the **COLLID** clause is specified with **PACKAGE PATH**, the **COLLID** clause is ignored when the function is invoked.

The *package-path* value that is associated with the function definition is checked when the function is invoked. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO SCRATCHPAD or SCRATCHPAD

Specifies whether DB2 is to provide a scratchpad for the function. Using reentrant external functions and a scratchpad (which provides an area for the function to save information from one invocation to the next) is strongly recommended.

NO SCRATCHPAD

A scratchpad is not allocated and passed to the function.

SCRATCHPAD *length*

When the function is invoked for the first time, DB2 allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- DB2 initializes the scratchpad to all binary zeros (X'00').
- The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad.

For example, assuming that user-defined function UDFX is a scalar function that is defined with the **SCRATCHPAD** option, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

```
SELECT A, UDFX(A) FROM TABLEB
WHERE UDFX(A) > 103 OR UDFX(A) < 19;
```

For another example, assume that UDFX is a user-defined table function that is defined with the **SCRATCHPAD** option. Two scratchpads are allocated for the two references to function UDFX in the following SQL statement:

```
SELECT *
FROM TABLE (UDFX(A)), TABLE (UDFX(B));
```

If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the SQL statement. Specify the **DISALLOW PARALLEL** clause for functions that do not work correctly with parallelism.

- The scratchpad is persistent. DB2 preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. DB2

initializes the scratchpads when it begins to execute an SQL statement. DB2 does not reset scratchpads when a correlated subquery begins to execute.

- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify **FINAL CALL** to ensure that DB2 calls the function one more time so that the function can free those system resources.

Each time that the function is invoked, DB2 passes an additional argument to the function that contains the address of the scratchpad.

If you specify **SCRATCHPAD**, DB2:

- Does not move the function from one TCB or address space to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

Do not specify **SCRATCHPAD** when **LANGUAGE JAVA** is specified.

NO FINAL CALL or FINAL CALL

Specifies whether a final call is made to the function. A *final call* enables the function to free any system resources that it has acquired. A final call is useful when the function has been defined with the **SCRATCHPAD** keyword and the function acquires system resource and anchors them in the scratchpad.

The effect of **NO FINAL CALL** or **FINAL CALL** depends on whether the external function is a scalar function or a table function.

For an external scalar function:

NO FINAL CALL

A final call is not made to the external scalar function. The function does not receive an additional argument that specifies the type of call.

FINAL CALL

A final call is made to the external scalar function. See the following description of call types for the characteristics of a final call. When **FINAL CALL** is specified, the function receives an additional argument that specifies the type of call to enable the function to differentiate between a final call and another type of call. Do not specify **FINAL CALL** when **LANGUAGE JAVA** is specified.

For more information on **NO FINAL CALL** and **FINAL CALL** for external scalar functions, including the types of calls, see the description of the option for “CREATE FUNCTION (external scalar)” on page 916.

For an external table function:

NO FINAL CALL

A first and final call are not made to the external table function.

FINAL CALL

A first call and final call are made to the external table function in addition to one or more other types of calls.

For both **NO FINAL CALL** and **FINAL CALL**, the function receives an additional argument that specifies the type of call. For more information on

NO FINAL CALL and FINAL CALL for external table functions, including the types of calls, see the description of the option for “CREATE FUNCTION (external table)” on page 940.

ALLOW or DISALLOW PARALLEL

Specifies whether, for a single reference to the function, the function can be executed in parallel. If the function is defined with **MODIFIES SQL DATA**, specify **DISALLOW PARALLEL**, not **ALLOW PARALLEL**.

ALLOW PARALLEL

Specifies that DB2 can consider parallelism for the function. Parallelism is not forced on the SQL statement that invokes the function or on any SQL statement in the function. Existing restrictions on parallelism apply.

See **SCRATCHPAD**, **EXTERNAL ACTION**, and **FINAL CALL** for considerations when specifying **ALLOW PARALLEL**.

DISALLOW PARALLEL

Specifies that DB2 does not consider parallelism for the function.

NO DBINFO or DBINFO

Specifies whether additional status information is passed to the function when it is invoked.

NO DBINFO

Additional information is not passed.

DBINFO

An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application runtime authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

Do not specify **DBINFO** when **LANGUAGE JAVA** is specified.

CARDINALITY *integer*

Specifies an estimate of the expected number of rows that the function returns. The number is used for optimization purposes. The value of *integer* must range from 0 to 2147483647.

If a function has an infinite cardinality (which means that the function never returns the “end-of-table” condition and always returns a row), a query that requires the end-of-table condition to work correctly needs to be interrupted. Thus, avoid using such functions in queries that involve **GROUP BY** and **ORDER BY**.

Do not specify **CARDINALITY** for external scalar functions.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is to be used when the function is executed. This is the package collection into which the DBRM that is associated with the function is bound.

NO COLLID

Specifies the package collection for the function is the same as the package collection of the program that invokes the function. If a trigger invokes the function, the collection of the trigger package is used. If the invoking program does not use a package, DB2 resolves the package by using the **CURRENT PACKAGE PATH** special register, the **CURRENT PACKAGESET**

special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

COLLID *collection-id*

Specifies the name of the package collection that is to be used when the function is executed.

WLM ENVIRONMENT

An SQL identifier that identifies the *name* of the WLM (workload manager) application environment in which the function is to run.

name

The WLM environment in which the function must run. If the user-defined function is nested and if the calling stored procedure or invoking user-defined function is not running in an address space associated with the specified WLM environment, DB2 routes the function request to a different address space.

(name,*)

When an SQL application program calls the function, *name* specifies the WLM environment in which the function runs.

If another user-defined function or a stored procedure calls the function, the function runs in the same environment that the calling routine uses. In this case, authorization to run the function in the WLM environment is not checked because the authorization of the calling routine suffices.

The *name* of the WLM environment is an SQL identifier.

To change the environment in which the function is to run, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see *Running stored procedures*.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of the function can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a function, setting a limit can be helpful if the function gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the service units.

LIMIT *integer*

The limit on the service units is a positive integer in the range of 1 to 2 147 483 647. If the function uses more service units than the specified value, DB2 cancels the function.

STAY RESIDENT

Specifies whether the load module for the function is to remain resident in memory when the function ends.

NO The load module is deleted from memory after the function ends. Use **NO** for non-reentrant functions.

YES

The load module remains resident in memory after the function ends. Use **YES** for reentrant functions.

PROGRAM TYPE

Specifies whether the function program runs as a main routine or a subroutine.

SUB

The function runs as a subroutine.

MAIN

The function runs as a main routine.

Do not specify **PROGRAM TYPE MAIN** when **LANGUAGE JAVA** is in effect.

SECURITY

Specifies how the function interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2

The function does not require an external security environment. If the function accesses resources that an external security product protects, the access is performed using the authorization ID associated with the WLM-established stored procedure address space.

USER

An external security environment should be used with the function. If the function accesses resources that the external security product protects, the access is performed using the primary authorization ID of the process that invoked the function.

DEFINER

An external security environment should be used with the function. If the function accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the function.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE

Specifies whether the routine is to be put in a stopped state after some number of failures. The following options must not be specified for SQL functions or sourced functions.

STOP AFTER SYSTEM DEFAULT FAILURES

Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX.

STOP AFTER *nn* FAILURES

Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE

Specifies that this routine should not be placed in a stopped state after any failure.

RUN OPTIONS *run-time-options*

Specifies the Language Environment runtime options to be used for the function. You must specify *run-time-options* as a character string that is no longer than 254 bytes. To replace any existing runtime options with no options, specify an empty string with **RUN OPTIONS**. When you specify an empty string, DB2 does not pass any runtime options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

Do not specify **RUN OPTIONS** when **LANGUAGE JAVA** is specified.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Specifies that special registers should be inherited according to the rules listed in the table for characteristics of special registers in a user-defined function in “Using special registers in a user-defined function or a stored procedure” on page 151.

DEFAULT SPECIAL REGISTERS

Specifies that special registers should be initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a user-defined function in “Using special registers in a user-defined function or a stored procedure” on page 151.

STATIC DISPATCH

At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters.

Notes

Changes are immediate: Any changes that the ALTER FUNCTION statement causes to the definition of an external function take effect immediately. The changed definition is used the next time that the function is invoked.

Invalidation of plans and packages: When an external function is altered, all the plans and packages that refer to that function are marked invalid.

LANGUAGE C and the PARAMETER VARCHAR clause: The ALTER statement does not allow you to alter the value of the **PARAMETER VARCHAR** or **PARAMETER CCSID** clauses that are associated with the function definition. However, you can alter the **LANGUAGE** clause for the function. If the **PARAMETER VARCHAR** clause is specified for the creation of a LANGUAGE C function, the catalog information for that option is not affected by a subsequent ALTER function statement. The function might be changed to a language other than C, in which case the **PARAMETER VARCHAR** setting is ignored. If the function is later changed back to LANGUAGE C, the setting of the **PARAMETER VARCHAR** option that was specified during the CREATE FUNCTION statement will be used.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**
- NOT NULL CALL as a synonym for **RETURNS NULL ON NULL INPUT**
- NULL CALL as a synonym for **CALLED ON NULL INPUT**
- PARAMETER STYLE DB2SQL as a synonym for **PARAMETER STYLE SQL**

Examples

Example 1: Assume that two functions CENTER are in the PELLOW schema. The first function has two input parameters with INTEGER and FLOAT data types,

respectively. The specific name for the first function is FOCUS1. The second function has three parameters with CHAR(25), DEC(5,2), and INTEGER data types.

Using the specific name to identify the function, change the WLM environment in which the first function runs from WLMENVNAME1 to WLMENVNAME2:

```
ALTER SPECIFIC FUNCTION ENGLES.FOCUS1 WLM ENVIRONMENT WLMENVNAME2;
```

Example 2: Change the second function that is described in *Example 1* so that it is not invoked when any of the arguments are null. Use the function signature to identify the function:

```
ALTER FUNCTION ENGLES.CENTER (CHAR(25), DEC(5,2), INTEGER)
  RETURNS NULL ON NULL INPUT;
```

You can also code the ALTER FUNCTION statement without the exact values for the CHAR and DEC data types:

```
ALTER FUNCTION ENGLES.CENTER (CHAR(), DEC(), INTEGER)
  RETURNS NULL ON NULL INPUT;
```

If you use empty parentheses, DB2 is to ignore the length, precision, and scale attributes when looking for matching data types to find the function.

ALTER FUNCTION (SQL scalar)

The ALTER FUNCTION (SQL scalar) statement changes the description of a user-defined SQL scalar function at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- Ownership of the function
- The ALTERIN privilege on the schema
- SYSADM authority
- SYSCTRL authority

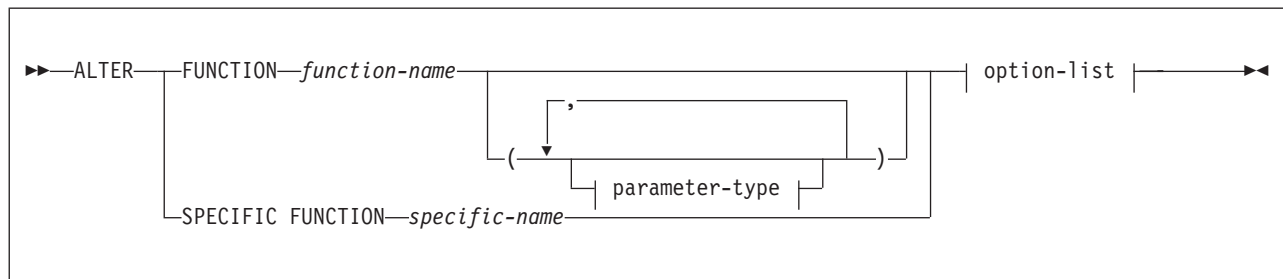
The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

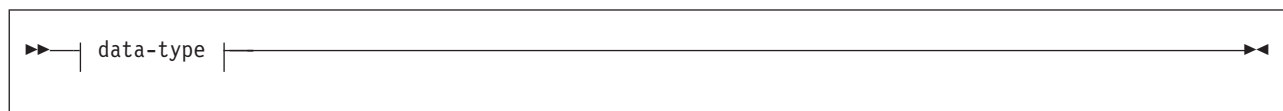
If the statement is dynamically prepared, the privilege set is the privileges that are held by the authorization IDs of the process. The specified function name can include a schema name (a qualifier). However, if the schema name is not the same as one of these authorization IDs, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- An authorization ID of the process has the ALTERIN privilege on the schema.

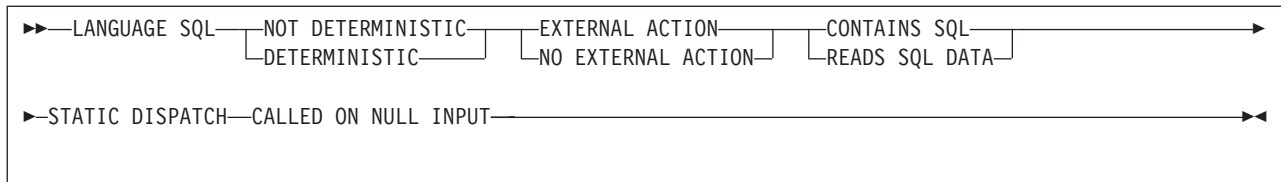
Syntax



parameter-type:



data-type:



Description

One of the following three clauses identifies the function to be changed.

FUNCTION *function-name*

Identifies the SQL function by its function name.

The identified function must be an SQL scalar function. There must be exactly one function with *function-name* in the schema. The function can have any number of input parameters. If the schema does not contain a function with *function-name* or contains more than one function with this name, an error occurs.

FUNCTION *function-name* (*parameter-type*,...)

Identifies the SQL function by its function signature, which uniquely identifies the function.

function-name

Gives the function name of the SQL function.

If *function-name*() is specified, the function that is identified must have zero parameters.

(*parameter-type*,...)

Identifies the number of input parameters of the function and the data type of each parameter. The data type of each parameter must match the data type that was specified in the CREATE FUNCTION statement for the parameter in the corresponding position. The number of data types and the logical concatenation of the data types are used to uniquely identify the function. Therefore, you cannot change the number of parameters or the data types of the parameters.

For data types that have a length, precision, or scale attribute, you can use a set of empty parentheses, specify a value, or accept the default values:

If the function was defined with a table parameter (the **LIKE TABLE** *name* **AS LOCATOR** clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

- Empty parentheses indicate that DB2 is to ignore the attribute when determining whether the data types match.

For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34).

FLOAT cannot be specified with empty parentheses because its parameter value indicates different data types (REAL or DOUBLE).

- If you use a specific value for a length, precision, or scale attribute, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

The specific value for `FLOAT(n)` does not have to exactly match the defined value of the source function because `1<=n<= 21` indicates `REAL` and `22<=n<=53` indicates `DOUBLE`. Matching is based on whether the data type is `REAL` or `DOUBLE`.

- If length, precision, or scale is not explicitly specified and empty parentheses are not specified, the default length of the data type is implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the `CREATE FUNCTION` statement.

For data types with a subtype or encoding scheme attribute, specifying the **FOR** *subtype* **DATA** clause or the **CCSID** clause is optional. Omission of either clause indicates that DB2 is to ignore the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the `CREATE FUNCTION` statement.

See “`CREATE FUNCTION`” on page 915 for more information on the specification of the parameter list.

A function with the function signature must exist in the explicitly or implicitly specified schema.

SPECIFIC FUNCTION *function-name*

Identifies a particular user-defined function by its specific name. The name is implicitly or explicitly qualified with a schema name. A function with the specific name must exist in the schema. If the specific name is not qualified, it is implicitly qualified with a schema name as described in the preceding description for **FUNCTION** *function-name*.

LANGUAGE SQL

Specifies the application programming language in which the stored function is written. The value of the function is written as DB2 SQL in the *expression* of the `RETURN` clause in the `CREATE FUNCTION` statement.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC

The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing `SELECT` or `SQL` data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers.

NOT DETERMINISTIC must be specified explicitly or implicitly if the function program accesses a special register or invokes another function that is not deterministic.

DETERMINISTIC

The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for `SELECT` or `SQL` data change statements that refer to this function. If applicable, specify **DETERMINISTIC** to prevent non-optimal access paths from being chosen for `SQL` statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

EXTERNAL ACTION

The function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function.

If you specify **EXTERNAL ACTION**, DB2:

- Materializes the views and table expressions in **SELECT** or SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify **EXTERNAL ACTION** if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between **FETCH** operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared **WITH HOLD**.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under **RRS** control.

EXTERNAL ACTION must be specified implicitly or explicitly specified if the SQL routine body invokes a function that is defined with **EXTERNAL ACTION**.

NO EXTERNAL ACTION

The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for **SELECT** or SQL data change statements that refer to this function. If applicable, specify **NO EXTERNAL ACTION** to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of **EXTERNAL ACTION** or **NO EXTERNAL ACTION**.

READS SQL DATA or CONTAINS SQL

Specifies which SQL statements, if any, can be executed in the function or any routine that is called from this function. The default is **READS SQL DATA**.

READS SQL DATA

Specifies that function can execute statements with a data access indication of **READS SQL DATA**, **CONTAINS SQL**, or **NO SQL**. The function cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the function can execute only SQL statements with a data classification of **CONTAINS SQL**. 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.

STATIC DISPATCH

At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null arguments. The function can return null.

Notes

Changes are immediate: Any changes that the ALTER FUNCTION statement causes to the definition of an SQL function take effect immediately. The changed definition is used the next time that the function is invoked.

Invalidation of plans and packages: When an SQL function is altered, all the plans and packages that refer to that function are marked invalid.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**
- NULL CALL as a synonym for **CALLED ON NULL INPUT**

Examples

Example 1: Modify the definition for an SQL function to indicate that the function is deterministic.

```
ALTER FUNCTION MY_UDF1
DETERMINISTIC;
```

ALTER INDEX

The ALTER INDEX statement changes the description of an index at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include one of the following:

- Ownership of the index
- Ownership of the table on which the index is defined
- DBADM authority for the database that contains the table
- SYSADM or SYSCTRL authority

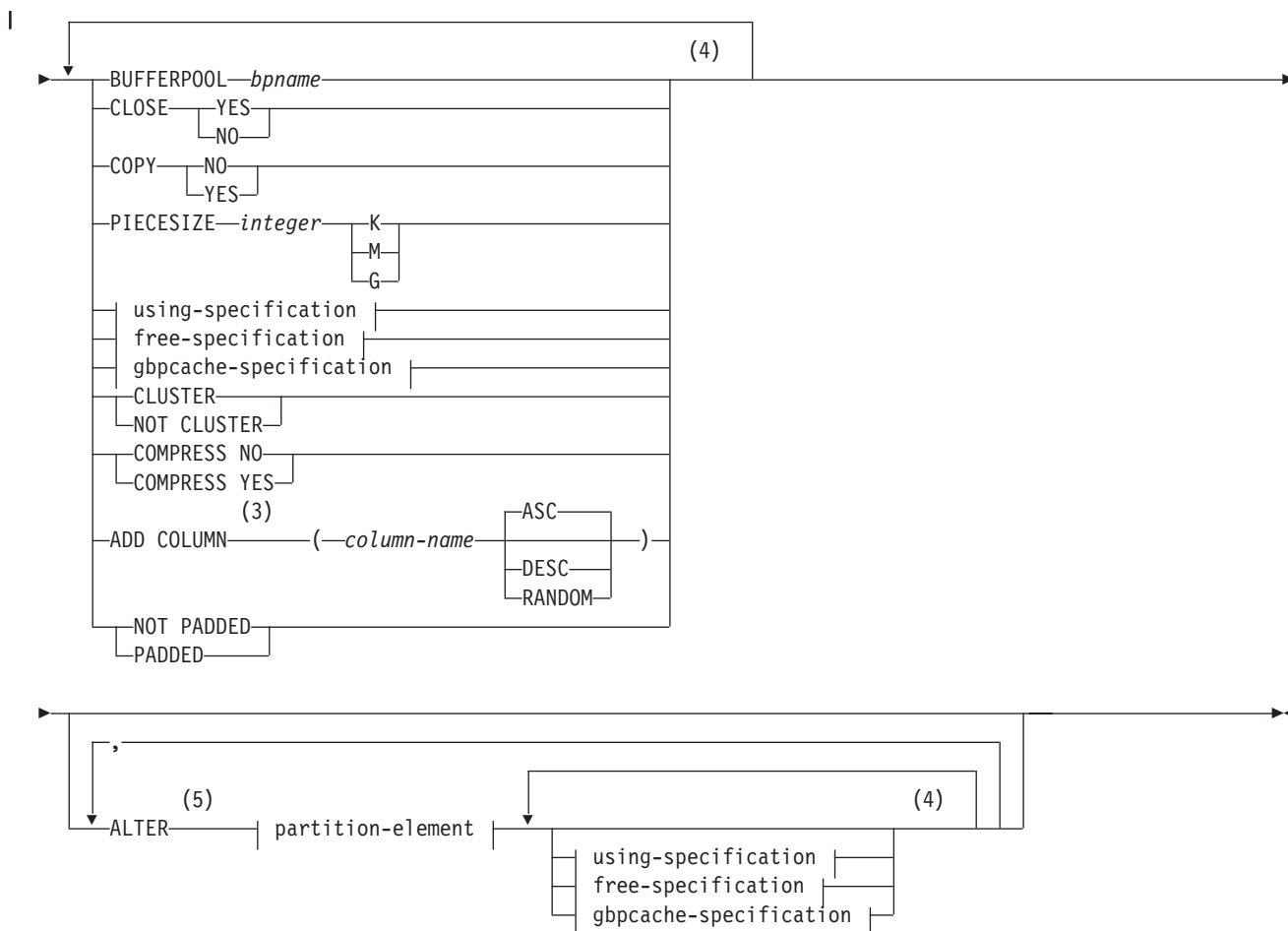
If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If **BUFFERPOOL** or **USING STOGROUP** is specified, additional privileges could be needed, as explained in the description of those clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Syntax

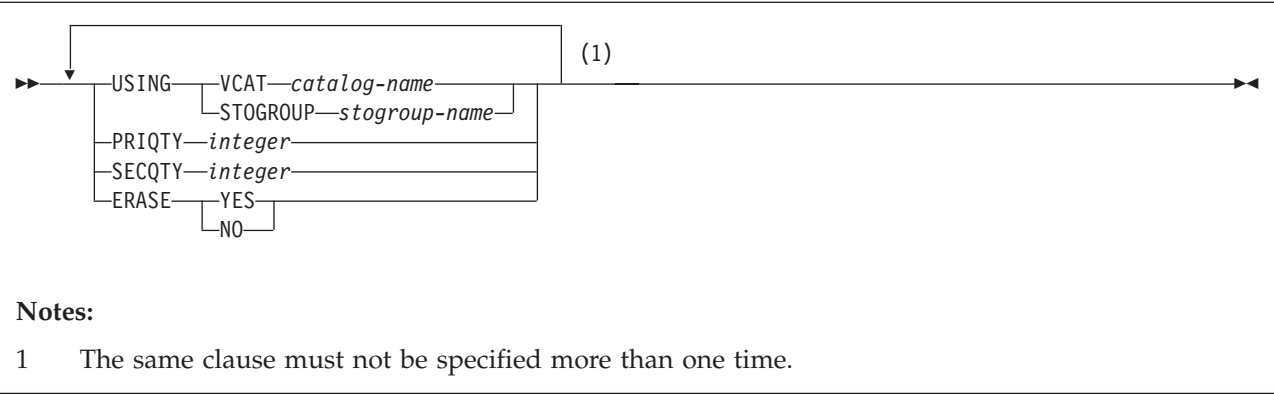
(1)
►► ALTER INDEX—*index-name*—(2)
REGENERATE—►



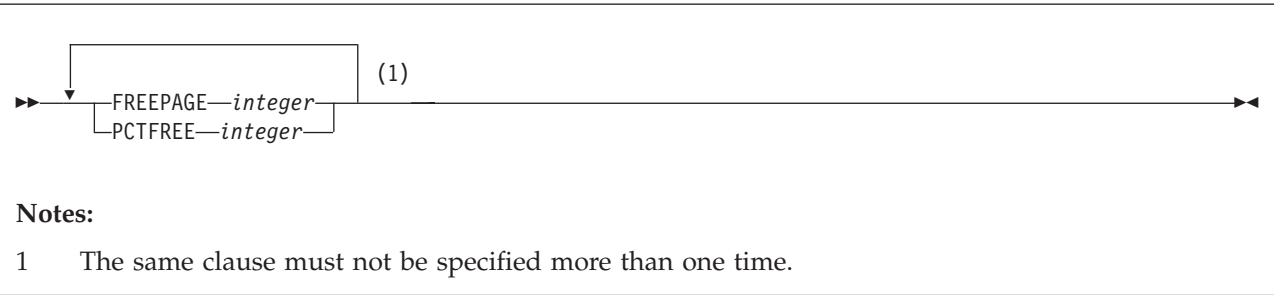
Notes:

- At least one clause must be specified after *index-name*. It can be from the optional list or it can be **ALTER PARTITION**.
- If **REGENERATE** is specified, it must be the only clause specified on the **ALTER INDEX** statement.
- If **ADD COLUMN** and **PADDED** or **NOT PADDED** are specified, **ADD COLUMN** must be specified before **PADDED** or **NOT PADDED**.
- The same clause must not be specified more than one time.
- The **ALTER** clause can only be specified for partitioned indexes. The **ALTER** clause must be specified last.

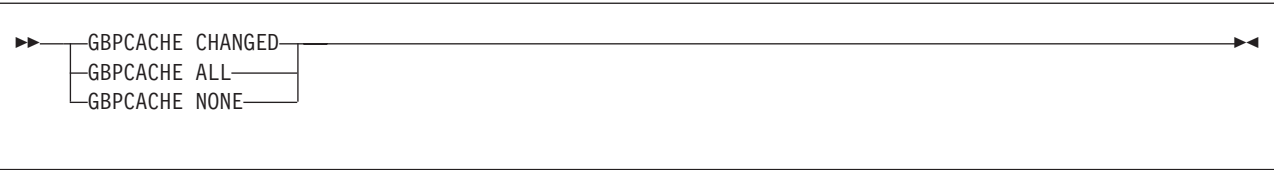
using-specification:



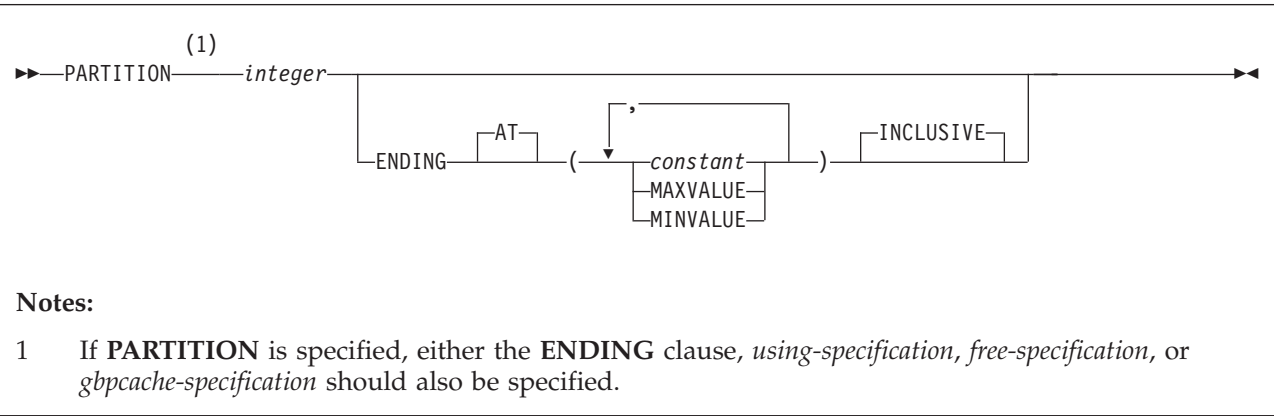
free-specification:



gbpcache-specification:



partition-element:



Description

index-name

Identifies the index to be changed or regenerated. The name must identify a

user-created index that exists at the current server. The name must not identify an index that is defined on a declared temporary table.

REGENERATE

Specifies that the index will be regenerated. The structure that represents the index definition is regenerated. The index definition will be composed from the catalog. Existing authorities and dependencies, if any, are retained. The catalog is updated with the regenerated index definition. The index is put into rebuild-pending state, all packages that depend on the index are invalidated, and catalog entries for the index statistics are deleted.

If the index cannot be successfully regenerated, an error is returned. In this case, the index must be dropped and recreated.

BUFFERPOOL *bpname*

Identifies the buffer pool that is to be used for the index. *bpname* must identify an activated 4K, 8 KB, 16 KB, or 32 KB buffer pool, and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the buffer pool.

A buffer pool with a smaller size should be chosen for indexes with random insert patterns. A buffer pool with a larger size should be chosen for indexes with sequential insert patterns.

If the index is changed to use index compression (the **COMPRESS YES** clause), the buffer pool must be greater than 4 KB in size.

The change to the description of the index takes effect the next time the data sets of the index space are opened. The data sets can be closed and reopened by a STOP DATABASE command to stop the index followed by a START DATABASE command to start the index.

If the buffer pool is changed to a buffer pool with a different page size, the index is placed into REBUILD-pending status.

In a data sharing environment, if you specify **BUFFERPOOL**, the index space must be in the stopped state when the ALTER INDEX statement is executed.

CLOSE

Specifies whether the data set is eligible to be closed when the index is not being used and the limit on the number of open data sets is reached. The change to the close rule takes effect the next time the data sets of the index space are opened.

YES

Eligible for closing.

NO Not eligible for closing.

If DSMAX is reached and there are no CLOSE YES page sets to close, CLOSE NO page sets will be closed.

COPY

Indicates whether the COPY utility is allowed for the index.

NO Does not allow full image or concurrent copies or the use of the RECOVER utility on the index.

YES

Allows full image or concurrent copies and the use the RECOVER utility on the index. For data sharing, changing **COPY** to **YES** causes additional SCA (Shared Communications Area) storage to be used until the next full or incremental image copy is taken or until **COPY** is set back to **NO**.

PIECESIZE *integer*

Specifies the maximum addressability of each data set for a secondary index. The **PIECESIZE** clause can only be specified for secondary indexes.

Be aware that when you alter the **PIECESIZE** value, the index is placed into page set REBUILD-pending (PSRBD) status. The entire index space becomes inaccessible. You must run the REBUILD INDEX or the REORG TABLESPACE utility to remove that status.

The subsequent keyword **K**, **M**, or **G**, indicates the units of the value that is specified in *integer*.

- K** Indicates that the *integer* value is to be multiplied by 1024 to specify the maximum data set size in bytes. The integer must be a power of two between 256 and 67 108 864.
- M** Indicates that the *integer* value is to be multiplied by 1 048 576 to specify the maximum data set size in bytes. The integer must be a power of two between 1 and 65 536.
- G** Indicates that the *integer* value is to be multiplied by 1 073 741 824 to specify the maximum data set size in bytes. The integer must be a power of two between 1 and 64.

Table 82 shows the valid values for data set size, which depend on the size of the table space.

Table 82. Valid values of **PIECESIZE** clause

K units	M units	G units	Size attribute of table space
256K			
512K			
1024K	1M		
2048K	2M		
4096K	4M		
8192K	8M		
16384K	16M		
32768K	32M		
65536K	64M		
131072K	128M		
262144K	256M		
524288K	512M		
1048576K	1024M	1G	
2097152K	2048M	2G	
4194304K	4096M	4G	LARGE, DSSIZE 4G (or greater)
8388608K	8192M	8G	DSSIZE 8G (or greater)
16777216K	16384M	16G	DSSIZE 16G (or greater)
33554432K	32768M	32G	DSSIZE 32G (or greater)
67108864K	65536M	64G	DSSIZE 64G

The data set size limit for partitioned table spaces with more than 256 partitions is 4096.

begin using-specification block

The components of the *using-specification* are discussed below, first for non-partitioned indexes and then for partitioned indexes.

USING (specification for nonpartitioned indexes)

For nonpartitioned indexes, the **USING** clause specifies whether the data sets for the index are to be managed by the user or managed by DB2. The **USING** clause applies to every data set that can be used for the index.

If you specify **USING**, the index must be in the stopped state when the **ALTER INDEX** statement is executed. See *Altering storage attributes* to determine how and when changes take effect.

VCAT *catalog-name*

Specifies a user-managed data set with a name that starts with the specified catalog name. You must specify the catalog name in the form of an SQL identifier. Thus, you must specify an alias if the name of the integrated catalog facility catalog is longer than eight characters. When the new description of the index is applied, the integrated catalog facility catalog must contain an entry for the data set that conforms to the DB2 naming conventions described in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

STOGROUP *stogroup-name*

Specifies using a DB2-managed data set that resides on a volume of the specified storage group. *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group. When the new description of the index is applied, the description of the storage group must include at least one volume serial number. Each volume serial number must identify a volume that is accessible to z/OS for dynamic allocation of the data set, and all identified volumes must be of the same device type. Furthermore, the integrated catalog facility catalog used for the storage group must not contain an entry for the data set.

If you specify **USING STOGROUP** and the current data set is DB2-managed, omission of the **PRIQTY**, **SECQTY**, or **ERASE** clause is an implicit specification of the current value of the omitted clause.

If you specify **USING STOGROUP** to convert from user-managed data sets to DB2-managed data sets:

- Omission of the **PRIQTY** clause is an implicit specification of the default value. For information on how DB2 determines the default value, see *Rules for primary and secondary space allocation*.
- Omission of the **SECQTY** clause is an implicit specification of the default value. For information on how DB2 determines the default value, see *Rules for primary and secondary space allocation*.
- Omission of the **ERASE** clause is an implicit specification of **ERASE NO**.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set. This clause can be specified only if the data set is currently managed by DB2 and **USING VCAT** is not specified.

If **PRIQTY** is specified (with a value other than -1), the primary space allocation is at least *n* kilobytes, where *n* is:

12 If *integer* is less than 12

integer

If *integer* is between 12 and 4194304

2097152

If both of the following conditions are true:

- *integer* is greater than 2097152.
- The index is a non-partitioned index on a table space that is not defined with the **LARGE** or **DSSIZE** attribute.

4194304

If *integer* is greater than 4194304

If **PRIQTY** -1 is specified, DB2 uses a default value for the primary space allocation. For information on how DB2 determines the default value for primary space allocation, see Rules for primary and secondary space allocation.

If **USING STOGROUP** is specified and **PRIQTY** is omitted, the value of **PRIQTY** is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of **USING STOGROUP**.)

If you specify **PRIQTY** and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of 4 KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see the description of the **DEFINE CLUSTER** command in *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

When determining a suitable value for **PRIQTY**, be aware that two of the pages of the primary space could be used by DB2 for purposes other than storing index entries.

SECQTY *integer*

Specifies the minimum secondary space allocation for a DB2-managed data set. This clause can be specified only if the data set is currently managed by DB2 and **USING VCAT** is not specified.

If **SECQTY** -1 is specified, DB2 uses a default value for the secondary space allocation.

If **USING STOGROUP** is specified and **SECQTY** is omitted, the value of **SECQTY** is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of **USING STOGROUP**.)

For information on the actual value that is used for secondary space allocation, whether you specify a value or DB2 uses a default value, see Rules for primary and secondary space allocation.

If you specify **SECQTY** and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of 4 KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see the description of the **DEFINE CLUSTER** command in *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

ERASE

Indicates whether the DB2-managed data sets are to be erased when they are deleted during the execution of a utility or an SQL statement that drops the index. Refer to for more information.

NO Does not erase the data sets. Operations involving data set deletion will perform better than **ERASE YES**. However, the data is still accessible, though not through DB2.

YES

Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

This clause can be specified only if the data set is currently managed by DB2 and **USING VCAT** is not specified. If you specify **ERASE**, the index must be in the stopped state when the ALTER INDEX statement is executed. See Altering storage attributes to determine how and when changes take effect.

USING (specification for partitioned indexes:)

For a partitioned index, there is an optional **PARTITION** clause for each partition. A *using-specification* can be specified at the global level or at the partition level. A *using-specification* within a **PARTITION** clause applies only to that partition. A *using-specification* specified before any **PARTITION** clauses applies to every partition except those with a **PARTITION** clause with a *using-specification*.

For DB2-managed data sets, the values of **PRIQTY**, **SECQTY**, and **ERASE** for each partition are given by the first of these choices that applies:

- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in the *using-specification* within the **PARTITION** clause for the partition. Do not use more than one *using-specification* in any **PARTITION** clause.
- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in the *using-specification* before any **PARTITION** clause
- The current values of **PRIQTY**, **SECQTY**, and **ERASE**

For data sets that are being changed from user-managed to DB2-managed, the values of **PRIQTY**, **SECQTY**, and **ERASE** for each partition are given by the first of these choices that applies:

- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in the *using-specification* within the **PARTITION** clause for the partition. Do not use more than one *using-specification* in any **PARTITION** clause.
- The values of **PRIQTY**, **SECQTY**, and **ERASE** given in a *using-specification* before any **PARTITION** clauses
- The default values of **PRIQTY**, **SECQTY**, and **ERASE**, which are:
 - **PRIQTY** 12
 - **SECQTY** 12, if **PRIQTY** is not specified in either *using-specification*, or 10% of **PRIQTY** or 3 times the index page size (whichever is larger) when **PRIQTY** is specified
 - **ERASE** NO

Any partition for which **USING** or **ERASE** is specified (either explicitly at the partition level or implicitly at the global level) must be in the stopped state when the ALTER INDEX statement is executed. See Altering storage attributes to determine how and when changes take effect.

VCAT catalog-name

Specifies a user-managed data set with a name that starts with the

specified catalog name. You must specify the catalog name in the form of an SQL identifier. Thus, you must specify an alias if the name of the integrated catalog facility catalog is longer than eight characters.

If n is the number of the partition, the identified integrated catalog facility catalog must already contain an entry for the v th data set of the index, conforming to the DB2 naming convention for data sets described in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DB2 assumes one and only one data set for each partition.

STOGROUP *stogroup-name*

If **USING STOGROUP** is used, *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group.

DB2 assumes one and only one data set for each partition.

For information on the **PRIQTY**, **SECQTY**, and **ERASE** clauses, see the description of those clauses in the *using-specification* for secondary indexes.

end using-specification block

begin free-specification block

FREEPAGE *integer*

| Specifies how often to leave a page of free space when index entries are
| created as the result of executing a DB2 utility. One free page is left for every
| *integer* pages. The value of *integer* can range from 0 to 255. The change to the
| description of the index or partition has no effect until it is loaded or
| reorganized using a DB2 utility. Do not specify **FREEPAGE** for an implicitly
| created XML index.

PCTFREE *integer*

| Determines the percentage of free space to leave in each nonleaf page and leaf
| page when entries are added to the index or partition as the result of executing
| a DB2 utility. The first entry in a page is loaded without restriction. When
| additional entries are placed in a nonleaf or leaf page, the percentage of free
| space is at least as great as *integer*.

| The value of *integer* can range from 0 to 99, however, if a value greater than 10
| is specified, only 10 percent of free space will be left in nonleaf pages. The
| change to the description of the index or partition has no effect until it is
| loaded or reorganized using a DB2 utility. Do not specify **PCTFREE** for an
| implicitly created XML index.

If the index is partitioned, the values of FREEPAGE and PCTFREE for a particular partition are given by the first of these choices that applies:

- The values of **FREEPAGE** and **PCTFREE** given in the **PARTITION** clause for that partition. Do not use more than one *free-specification* in any **PARTITION** clause.
- The values given in a *free-specification* before any **PARTITION** clauses.
- The current values of **FREEPAGE** and **PCTFREE** for that partition.

end free-specification block

begin gbpcache-specification block

GBPCACHE

Specifies what index pages are written to the group buffer pool in a data sharing environment. In a non-data-sharing environment, you can specify this option, but it is ignored.

CHANGED

When there is inter-DB2 read-write interest on the index or partition, updated pages are written to the group buffer pool. When there is no inter-DB2 read-write interest, the group buffer pool is not used. Inter-DB2 read-write interest exists when more than one member in the data sharing group has the index or partition open, and at least one member has it open for update.

If the index is in a group buffer pool that is defined as **GBPCACHE(NO)**, **CHANGED** is ignored and no pages are cached to the group buffer pool.

ALL

Indicates that pages are to be cached to the group buffer pool as they are read in from DASD, with one exception. When the page set is not GBP-dependent and one DB2 data sharing member has exclusive read-write interest in that page set (no other group members have any interest in the page set), no pages are cached in the group buffer pool.

If the index is in a group buffer pool that is defined as **GBPCACHE(NO)**, **ALL** is ignored and no pages are cached to the group buffer pool.

NONE

Indicates that no pages are to be cached to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If you specify **NONE**, the index or partition must not be in group buffer pool recover-pending (GRECP) status.

If the index is partitioned, the value of **GBPCACHE** for a particular partition is given by the first of these choices that applies:

1. The value of **GBPCACHE** given in the **PARTITION** clause for that partition. Do not use more than one *gbpcache-specification* in any **PARTITION** clause.
2. The value given in a *gbpcache-specification* before any **PARTITION** clauses.
3. The current value of **GBPCACHE** for that partition.

If you specify **GBPCACHE** in a data sharing environment, the index or partition must be in the stopped state when the ALTER INDEX statement is executed. You cannot alter the GBPCACHE value for certain indexes on DB2 catalog tables; for more information, see "SQL statements allowed on the catalog" on page 1689.

end gbpcache-specification block

CLUSTER or NOT CLUSTER

Specifies whether the index is the clustering index for the table.

CLUSTER

The index is used as the clustering index for the table. This change takes effect immediately. Any subsequent insert operations will use the new

clustering index. Existing data remains clustered by the previous clustering index until the table space is reorganized.

The implicit or explicit clustering index is ignored when data is inserted into a table space that is defined with **MEMBER CLUSTER**. Instead of using cluster order, DB2 chooses where to locate the data based on available space. The **MEMBER CLUSTER** attribute affects only data that is inserted with an insert operation; data is always loaded and reorganized in cluster order.

Do not specify **CLUSTER** in the following cases:

- The index is for an auxiliary table.
- **CLUSTER** was used already for a different index on the table.
- The index is an XML index.
- The index includes expressions.

NOT CLUSTER

The index is not used as the clustering index of the table. If the index is already defined as the clustering index, it continues to be used as the clustering index by DB2 and the REORG utility until clustering is explicitly changed by specifying **CLUSTER** for a different index.

Specifying **NOT CLUSTER** for an index that is not a clustering index is ignored.

If the index is the partitioning index for a table that uses index-controlled partitioning, the table is converted to use table-controlled partitioning. The high limit key for the last partition is set to the highest possible value for ascending key columns or the lowest possible value for descending key columns.

COMPRESS NO or COMPRESS YES

Specifies whether the index data will be compressed. If the index is partitioned, this option will apply to all partitions.

COMPRESS NO

Specifies that index compression will be turned off. If the index was created using the **COMPRESS YES** clause, changing to **COMPRESS NO** will place the index in advisory reorg-pending state.

COMPRESS YES

Specifies that the index will use index compression. The index must be in a buffer pool that has a page size greater than 4 KB. Index compression will not take place immediately and the index will be placed in advisory reorg-pending state. Index compression will not be in effect until the REORG INDEX, REBUILD INDEX, or REORG TABLESPACE utility is run. **COMPRESS YES** can be specified for user-managed data sets only if the control interval size is 4K.

NOT PADDED or PADDED

Specifies how varying-length string columns are to be stored in the index. If the index contains no varying-length columns, this option is ignored, and a warning message is returned.

NOT PADDED

Specifies that varying-length string columns are not to be padded to their maximum length in the index. The length information for a varying-length column is stored with the key.

NOT PADDED is ignored and has no effect if the index is on an auxiliary table. Indexes on auxiliary tables are always padded.

When **PADDED** is changed to **NOT PADDED**, the maximum key length is recalculated with the varying-length formula $(2000 - n - 2m)$, where n is the number of columns that can contain null values and m is the number of varying-length columns in the key). If it is possible that the index key length might exceed the maximum length (because when it was padded, the formula $2000 - n$ was used), an error occurs.

PADDED

Specifies that varying-length string columns within the index are always padded with the default pad character to their maximum length.

When an index with at least one varying-length column is changed from **PADDED** to **NOT PADDED**, or vice versa, the index is placed in restricted rebuild-pending status (RBDP). The index cannot be accessed until it is rebuilt from the table (using the REBUILD INDEX, REORG TABLESPACE, or LOAD REPLACE utility). For nonpartitioned secondary indexes (NPSIs), the index is placed in page set rebuild-pending status (PSRBD), and the entire index must be rebuilt. In addition, plans and packages that are dependent on the table are quiesced, and dynamically cached statements that are dependent on the index are invalidated.

Do not specify **PADDED** if the index is an XML index.

ADD COLUMN *column-name*

Adds *column-name* to the index. *column-name* must be unqualified, must identify a column of the table, must not be one of the existing columns of the index, and must not be a LOB column, a DECFLOAT column, or a distinct-type column that is based on a LOB or DECFLOAT data type. The column cannot be a VARBINARY column or a distinct-type column that is based on a VARBINARY data type if the column is defined with the DESC attribute. The total number of columns for the index cannot exceed 64.

For **PADDED** indexes, the sum of the length attributes of the columns must not be greater than $2000 - n$, where n is the number of columns that can contain null values. For **NOT PADDED** indexes, the sum of the length attributes of the columns must not be greater than $2000 - n - 2m$, where n is the number of nullable columns and m is the number of varying-length columns.

The index cannot be any of the following types of indexes:

- A system-defined catalog index
- An index that enforces a primary key, unique key, or referential constraint, or matches a foreign key
- A partitioning index when index-controlled partitioning is being used
- A unique index required for a ROWID column defined as GENERATED BY DEFAULT
- An auxiliary index
- An XML index
- An index that includes expressions

If a column is added to both a table and an associated index within the same commit scope, the index is put into rebuild-pending (RBDP) status if rows are inserted into the table within the same commit scope or the column is a ROWID column; otherwise, the index is put into an advisory reorg-pending (AREO*) state.

ASC

Index entries are put in ascending order by the column.

DESC

Index entries are put in descending order by the column.

RANDOM

Index entries are put in a random order by the column. **RANDOM** cannot be specified for an index key column that is varying length in an index that is created with the **NOT PADDED** option.

ALTER PARTITION *integer*

Identifies the partition of the index to be altered. For an index that has n partitions, you must specify an integer in the range 1 to n . You must not use this clause if the index is nonpartitioned. You must use this clause if the index is partitioned and you specify the **ENDING AT** clause.

ENDING AT(*constant*), **MAXVALUE**, or **MINVALUE**

Specifies the highest value of the index key for the identified partition of the partitioning index. In this context, highest means highest in the sorting sequence of the index columns. In a column defined as ascending (**ASC**), highest and lowest have the usual meanings. In a column defined as descending (**DESC**), the lowest actual value is highest in the sorting sequence.

You must use at least one value (*constant*, **MAXVALUE**, or **MINVALUE**) after **ENDING AT** in each **PARTITION** clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the corresponding partition of the index. The length of each highest key value (also called the limit key) is the same as the length of the partitioning index

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (**GX**) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the **MINVALUE** clause cannot be specified following **MAXVALUE**. After **MAXVALUE** is specified, all subsequent columns must be **MAXVALUE**.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the **MAXVALUE** clause cannot be specified following **MAXVALUE**. After **MINVALUE** is specified, all subsequent columns must be **MINVALUE**.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on.
- If a key includes a ROWID column (or a column with a distinct type that is based on a ROWID data type), the values of the ROWID column are assumed to be in the range of X'000...00' to X'FFF...FF'. Only the first 17 bytes of the value that is specified for the corresponding ROWID column are considered.
- Using fewer values than there are columns in the key has the same effect as using the highest possible values for all omitted columns for an ascending index.
- If the key exceeds 255 bytes, only the first 255 bytes are considered.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition.
- The highest value of the key in the last partition depends on how the table space was defined. For table spaces created without the LARGE or DSSIZE option, the constants you specify after **ENDING AT** are not enforced. The highest value of the key that can be placed in the table is the highest possible value of the key.

For table spaces created with the LARGE or DSSIZE options, the constants you specify after **ENDING AT** are enforced. The value specified for the last partition is the highest value of the key that can be placed in the table. Any keys that are made invalid after the ALTER TABLE statement is executed are placed in a discard data set when you run the REORG utility. If the last partition is in reorg-pending status, regardless of whether you changed its limiting key values, you must specify a discard data set when you run the REORG utility.

ENDING AT must not be specified for any indexes defined on a table that uses table-controlled partitioning. Use ALTER TABLE ALTER PARTITION to modify the partitioning boundaries for a table that uses table-controlled partitioning.

INCLUSIVE

Specifies that the specified range values are included in the data partition.

Notes

Altering storage attributes: The **USING**, **PRIQTY**, **SECQTY**, and **ERASE** clauses define the storage attributes of the index or partition. If you specify the **USING** or **ERASE** clause when altering storage attributes, the index or partition must be in the stopped state when the ALTER INDEX statement is executed. A STOP DATABASE...SPACENAM... command can be used to stop the index or partition.

If the catalog name changes, the changes take effect after you move the data and start the index or partition using the START DATABASE...SPACENAM... command. The catalog name can be implicitly or explicitly changed by the ALTER INDEX statement. The catalog name also changes when you move the data to a different device. See the procedures for moving data in *DB2 Administration Guide*.

Changes to the secondary space allocation (SECQTY) take effect the next time DB2 extends the data set; however, the new value is not reflected in the integrated catalog until you use the REORG, RECOVER, or LOAD REPLACE utility on the index or partition. Changes to the other storage attributes take effect the next time you use the REORG, RECOVER, or LOAD REPLACE utility on the index or

partition. If you change the primary space allocation parameters or erase rule, you can have the changes take effect earlier if you move the data before you start the index or partition.

Altering indexes on DB2 catalog tables: For details on altering options on catalog tables, see “SQL statements allowed on the catalog” on page 1689.

Altering limit keys: If you specify **ALTER PARTITION integer ENDING AT** to change the limit key values of a partitioning index, the plans and packages that are dependent on that index are marked invalid and go through automatic rebind the next time they are run.

Invalidation of plans and packages: When an index is altered, all the plans and packages that refer to that index are marked invalid if one of the following conditions is true:

- A column is added to the index.
- The index is altered to be **PADDED** or **NOT PADDED**.
- The index is a partitioning index on a table that uses index-controlled partitioning, and one or more limit key values is altered.
- The index is altered to **REGENERATE**.

Restrictions on SQL data change statements in the same commit scope as ALTER INDEX: SQL data change statements that affect an index cannot be performed in the same commit scope as ALTER INDEX statements that affect that index.

Altering indexes for tables that are involved in a clone relationship: You cannot change any index for a table that is involved in a clone relationship (base table or clone table). If a change to an index is required, the clone table must be dropped, then the index can be changed. After the index is changed, the clone table can be created again.

Running utilities: You cannot execute the ALTER INDEX statement while a DB2 utility has control of the index or its associated table space.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords when altering the partitions of a partitioned index:

- PART can be specified as a synonym for **PARTITION**. In addition, the **ALTER** keyword that precedes **PARTITION** is optional. In addition, if you alter more than one partition, specifying a comma between each **ALTER PARTITION integer** clause is optional.
- VALUES can be specified as a synonym for **ENDING AT**.

Although these keywords are supported as alternatives, they are not the preferred syntax.

Examples

Example 1: Alter the index DSN8910.XEMP1. Indicate that DB2 is not to close the data sets that support the index when there are no current users of the index.

```
ALTER INDEX DSN8910.XEMP1
CLOSE NO;
```

Example 2: Alter the index DSN8910.XPROJ1. Use BP1 as the buffer pool that is to be associated with the index, indicate that full image or concurrent copies on the index are allowed, and change the maximum size of each data set to 8 megabytes.

```
ALTER INDEX DSN8910.XPROJ1
  BUFFERPOOL BP1
  COPY YES
  PIECESIZE 8M;
```

Example 3: Assume that index X1 contains a least one varying-length column and is a padded index. Alter the index to an index that is not padded.

```
ALTER INDEX X1
  NOT PADDED;
```

The index is placed in restricted rebuild-pending status (RBDP) and cannot be accessed until it is rebuilt from the table

Example 4: Alter partitioned index DSN8910.DEPT1. For partition 3, leave one page of free space for every 13 pages and 13 percent of free space per page. For partition 5, leave one page for every 25 pages and 25 percent of free space. For all the other partitions, leave one page of free space for every 6 pages and 11 percent of free space. Ensure that index pages are cached to the group buffer pool for all partitions except partition 4. For partition 4, write pages only when there is inter-DB2 read-write interest on the partition.

```
ALTER INDEX DSN8910.XDEPT1
  BUFFERPOOL BP1
  CLOSE YES
  COPY YES
  USING VCAT CATLGG
  FREEPAGE 6
  PCTFREE 11
  GBPCACHE ALL
  ALTER PARTITION 3
    USING VCAT CATLGG
    FREEPAGE 13
    PCTFREE 13,
  ALTER PARTITION 4
    USING VCAT CATLGG
    GBPCACHE CHANGED,
  ALTER PARTITION 5
    USING VCAT CATLGG
    FREEPAGE 25
    PCTFREE 25;
```

ALTER PROCEDURE (external)

The ALTER PROCEDURE statement changes the description of an external stored procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- Ownership of the stored procedure
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the authorization IDs of the process. The specified procedure name can include a schema name (a qualifier). However, if the schema name is not the same as one of these authorization IDs, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- An authorization ID of the process has the ALTERIN privilege on the schema.

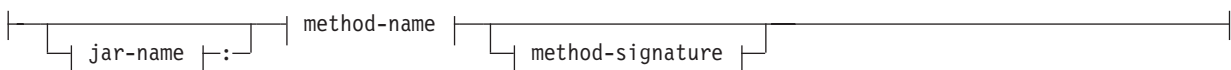
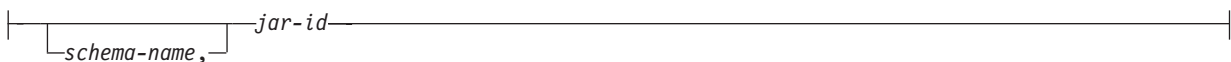
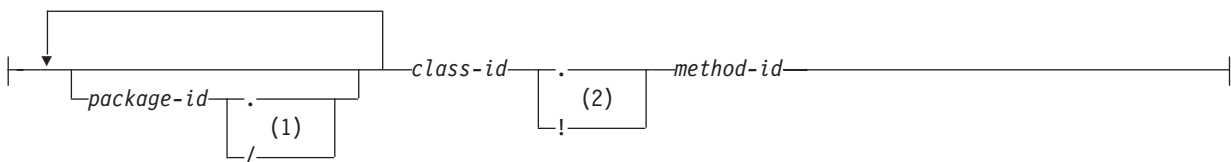
If the environment in which the stored procedure is to run is being changed, the authorization ID must have authority to use the WLM environment. This authorization is obtained from an external security product, such as RACF.

When **LANGUAGE** is **JAVA** and a *jar-name* is specified in the **EXTERNAL NAME** clause, the privilege set must include **USAGE** on the JAR file, the Java archive file.

Syntax

▶▶ALTER PROCEDURE—*procedure-name*—| option-list |————▶▶

option-list: (Specify options in any order. Specify at least one option. Do not specify the same option more than once.)

external-java-routine-name:**jar-name:****method-name:****method-signature:****Notes:**

- 1 The slash (/) is supported for compatibility with previous releases of DB2 for z/OS.
- 2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

Description

procedure-name

Identifies the stored procedure to be altered.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of query result sets that the stored procedure can return. The value must be between 0 and 32767.

EXTERNAL NAME *external-program-name* **or** *identifier*

Specifies the name of the MVS load module for the program that runs when the procedure name is specified in an SQL CALL statement.

If **LANGUAGE** is **JAVA**, *external-program-name* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *external-program-name* values are specified, the total length of all of the values must not be greater than 1305 bytes and each value must be separated by a space or a line break. Do not specify a JAR file for a Java procedure for which **NO SQL** is in effect.

An *external-java-routine-name* contains the following parts:

jar-name

Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER bind option when the package or plan was created or last rebound. If the QUALIFIER was not specified, the schema name is the owner of the package or plan.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the ALTER PROCEDURE statement is processed.

If *jar-name* is not specified, the procedure is loaded from the class file directly instead of being loaded from a JAR file. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed stored procedure.

method-name

Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method ID's are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what these can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id

Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.StoredProcs." The Java virtual machine looks in the directory "/myPacks/StoredProcs/" for the classes.

class-id

Identifies the class identifier of the Java object.

method-id

Identifies the method identifier with the Java class to be invoked.

method-signature

Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the procedure involves any input or output parameters that can be NULL. When the stored procedure being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the ALTER PROCEDURE statement.

For other values of **LANGUAGE**, the value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8

bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

LANGUAGE

Specifies the application programming language in which the stored procedure is written. Assembler, C, COBOL, and PL/I programs must be designed to run in IBM's Language Environment.

ASSEMBLE

The stored procedure is written in Assembler.

C The stored procedure is written in C or C++.

COBOL

The stored procedure is written in COBOL, including the OO-COBOL language extensions.

JAVA

The stored procedure is written in Java and is executed in the Java Virtual Machine. When **LANGUAGE JAVA** is specified, the **EXTERNAL NAME** clause must also be specified with a valid *external-java-routine-name* and **PARAMETER STYLE** must be specified with **JAVA**. The procedure must be a public static method of the specified Java class.

Do not specify **LANGUAGE JAVA** when **DBINFO**, **PROGRAM TYPE MAIN**, or **RUN OPTIONS** is in effect.

PLI

The stored procedure is written in PL/I.

REXX

The stored procedure is written in REXX. Do not specify **LANGUAGE REXX** when **PARAMETER STYLE SQL** is specified.

PARAMETER STYLE

Identifies the linkage convention used to pass parameters to and return values from the stored procedure. All of the linkage conventions provide arguments to the stored procedure that contain the parameters specified on the **CALL** statement. Some of the linkage conventions pass additional arguments to the stored procedure that provide more information to the stored procedure. For more information on linkage conventions, see *DB2 Application Programming and SQL Guide*.

SQL

Specifies that, in addition to the parameters on the **CALL** statement, several additional parameters are passed to the stored procedure. The following parameters are passed:

- The first *n* parameters that are specified on the **CREATE PROCEDURE** statement.
- *n* parameters for indicator variables for the parameters.
- The **SQLSTATE** to be returned.
- The qualified name of the stored procedure.
- The specific name of the stored procedure.
- The SQL diagnostic string to be returned to DB2.
- If **DBINFO** is specified, the **DBINFO** structure.

Do not specify **PARAMETER STYLE SQL** when **LANGUAGE REXX** is specified.

GENERAL

Specifies that the stored procedure uses a parameter passing mechanism where the stored procedure receives only the parameters specified on the CALL statement. Arguments to procedures defined with this parameter style cannot be null.

GENERAL WITH NULLS

Specifies that, in addition to the parameters on the CALL statement as specified in **GENERAL**, another argument is also passed to the stored procedure. The additional argument contains an indicator array with an element for each of the parameters on the CALL statement. In C, this is an array of short integers. The indicator array enables the stored procedure to accept or return null parameter values.

JAVA

Specifies that the stored procedure uses a parameter passing convention that conforms to the Java and SQLJ Routines specifications. **PARAMETER STYLE JAVA** can be specified only if **LANGUAGE** is **JAVA**. If the ALTER PROCEDURE statement results in changing **LANGUAGE** to **JAVA**, **PARAMETER STYLE JAVA**, and an **EXTERNAL NAME** clause might need to be specified to provide appropriate values. **JAVA** must be specified for **PARAMETER STYLE** when **LANGUAGE** is **JAVA**.

INOUT and **OUT** parameters are passed as single-entry arrays. The **INOUT** and **OUT** parameters are declared in the Java method as single-element arrays of the Java type.

PARAMETER STYLE SQL cannot be used with **LANGUAGE REXX**.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the stored procedure returns the same results each time the stored procedure is called with the same **IN** and **INOUT** arguments.

DETERMINISTIC

The stored procedure always returns the same results each time the stored procedure is called with the same **IN** and **INOUT** arguments, if the referenced data in the database has not changed.

NOT DETERMINISTIC

The stored procedure might not return the same result each time the procedure is called with the same **IN** and **INOUT** arguments, even when the referenced data in the database has not changed.

DB2 does not verify that the stored procedure code is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

NO PACKAGE PATH or PACKAGE PATH *package-path*

Identifies the package path to use when the procedure is run. This is the list of the possible package collections into which the DBRM this is associated with the procedure is bound.

NO PACKAGE PATH

Specifies that the list of package collections for the procedure is the same as the list of package collections for the calling program. If the calling program does not use a package, DB2 resolves the package by using the **CURRENT PACKAGE PATH** special register, the **CURRENT PACKAGESET** special register, or the **PKLIST** bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

| **PACKAGE PATH** *package-path*

| Specifies a list of package collections, in the same format as used in the
| CURRENT PACKAGE PATH special register.

| If the **COLLID** clause is specified with **PACKAGE PATH**, the **COLLID**
| clause is ignored when the routine is invoked.

| The *package-path* value that is associated with the procedure definition is
| checked when the procedure is invoked. If *package-path* contains
| SESSION_USER, USER, PATH, or PACKAGE PATH, an error is returned
| when the *package-path* value is checked.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL

Specifies which SQL statements, if any, can be executed in the procedure or any routine that is called from this procedure. For the data access classification of each statement, see Table 138 on page 1605.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA

| Specifies that procedure can execute statements with a data access
| indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The
| procedure cannot execute SQL statements that modify data.

CONTAINS SQL

| Specifies that the procedure can execute only SQL statements with an
| access indication of CONTAINS SQL. The procedure cannot execute
| statements that read or modify data.

NO SQL

Specifies that the procedure can execute only SQL statements with a data access classification of NO SQL. Do not specify **NO SQL** for a Java procedure that uses a JAR file.

NO DBINFO or DBINFO

Specifies whether additional status information is passed to the stored procedure when it is invoked.

NO DBINFO

Additional information is not passed.

DBINFO

An additional argument is passed when the stored procedure is invoked. The argument is a structure that contains information such as the application run-time authorization ID, the schema name, the name of a table or column that the procedure might be inserting into or updating, and identification of the database server that invoked the procedure. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

DBINFO can be specified only if **PARAMETER STYLE SQL** is specified.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is to be used when the stored procedure is executed. This is the package collection into which the DBRM that is associated with the stored procedure is bound.

NO COLLID

Specifies that the package collection for the stored procedure is the same as the package collection of the calling program. If the invoking program does

not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

COLLID *collection-id*

Identifies the package collection that is to be used when the stored procedure is executed. It is the name of the package collection into which the DBRM associated with the stored procedure is bound.

For REXX stored procedures, *collection-id* can be DSNREXRR, DSNREXRS, DSNREXCR, or DSNREXCS.

WLM ENVIRONMENT

Identifies the WLM (workload manager) environment in which the stored procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

name

The WLM environment in which the stored procedure must run. If another stored procedure or a user-defined function calls the stored procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the stored procedure request to a different address space.

(name,)*

When the stored procedure is called directly by an SQL application program, the WLM environment in which the stored procedure runs.

If another stored procedure or a user-defined function calls the stored procedure, the stored procedure runs in the same WLM environment that the calling routine uses.

To change the environment in which the procedure is to run, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see *Running stored procedures*.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of a stored procedure can run. The value is unrelated to the ASUTIME column in the resource limit specification table.

When you are debugging a stored procedure, setting a limit can be helpful in case the stored procedure gets caught in a loop. For information on CPU service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the service units.

LIMIT *integer*

The limit on the service units is a positive *integer* in the range of 1 to 2 147 483 647. If the stored procedure uses more service units than the specified value, DB2 cancels the stored procedure.

STAY RESIDENT

Specifies whether the stored procedure load module is to remain resident in memory when the stored procedure ends.

NO The load module is deleted from memory after the stored procedure ends. Use **NO** for non-reentrant stored procedures.

YES

The load module remains resident in memory after the stored procedure ends.

PROGRAM TYPE

Specifies whether the stored procedure runs as a main routine or a subroutine. If PROGRAM TYPE is altered, the stored procedure needs to be re-compiled for the change to take effect.

SUB

The stored procedure runs as a subroutine.

Do not specify **PROGRAM TYPE SUB** for stored procedures with a **LANGUAGE** value of **REXX**.

MAIN

The stored procedure runs as a main routine.

Do not specify **PROGRAM TYPE MAIN** when **LANGUAGE JAVA** is specified.

SECURITY

Specifies how the stored procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2

The stored procedure does not require a special external security environment. If the stored procedure accesses resources that an external security product protects, the access is performed using the authorization ID associated with the stored procedure address space.

USER

An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the stored procedure.

DEFINER

An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the stored procedure.

RUN OPTIONS *run-time-options*

Specifies the Language Environment run-time options to be used for the stored procedure. For a REXX stored procedure, specifies the Language Environment run-time options to be passed to the REXX language interface to DB2. You must specify *run-time-options* as a character string that is no longer than 254 bytes. To replace any existing run-time options with no options, specify an empty string with **RUN OPTIONS**. When you specify an empty string, DB2 does not pass any run-time options to Language Environment, and Language Environment uses its installation defaults. For a description of the Language Environment run-time options, see *z/OS Language Environment Programming Reference*.

Do not specify **RUN OPTIONS** when **LANGUAGE JAVA** is specified.

COMMIT ON RETURN

Indicates whether DB2 is to commit the transaction immediately on return from the stored procedure.

NO DB2 does not issue a commit when the stored procedure returns.

YES

DB2 issues a commit when the stored procedure returns if the following statements are true:

- The SQLCODE that is returned by the CALL statement is not negative.
- The stored procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the stored procedure.

If the stored procedure returns result sets, the cursors that are associated with the result sets must have been defined WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Indicates that values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a stored procedure in Table 37 on page 151.

DEFAULT SPECIAL REGISTERS

Indicates that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a stored procedure in Table 37 on page 151.

CALLED ON NULL INPUT

Specifies that the procedure is to be called even if any or all of the argument values are null, which means that the procedure must be coded to test for null argument values. The procedure can return null or nonnull values.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE

Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES

Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation field DSNTIPX.

STOP AFTER *nn* FAILURES

Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE

Specifies that this routine should not be placed in a stopped state after any failure.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE

Specifies whether the procedure can be run in debugging mode.

Do not specify this option unless the procedure is defined with LANGUAGE JAVA.

ALLOW DEBUG MODE

Specifies that the procedure can be run in debugging mode.

DISALLOW DEBUG MODE

Specifies that the procedure cannot be run in debugging mode.

You can use a subsequent ALTER PROCEDURE statement to change this option to ALLOW DEBUG MODE.

DISABLE DEBUG MODE

Specifies that the procedure can never be run in debugging mode.

The procedure cannot be changed to specify **ALLOW DEBUG MODE** or **DISALLOW DEBUG MODE** when the procedure has been created or altered to use **DISABLE DEBUG MODE**. To change this option, you must drop and recreate the procedure using the desired option.

Notes

Changes are immediate: Any changes that the ALTER PROCEDURE statement causes to the definition of a procedure take effect immediately, with the exception of changes to the PROGRAM TYPE clause. If PROGRAM TYPE is changed, the stored procedure needs to be re-compiled for the change to take effect. The changed definition is used the next time that the procedure is called.

Invalidation of plans and packages: When an external procedure is altered, all the plans and packages that refer to that procedure are marked invalid.

LANGUAGE C and the PARAMETER VARCHAR clause: The ALTER PROCEDURE statement does not allow you to alter the value of the **PARAMETER VARCHAR** or **PARAMETER CCSID** clauses that are associated with the procedure definition. However, you can alter the **LANGUAGE** clause for the procedure. If the **PARAMETER VARCHAR** clause is specified for the creation of a LANGUAGE C procedure, the catalog information for that option is not affected by subsequent ALTER PROCEDURE statements. The procedure might be changed to a language other than C, in which case the **PARAMETER VARCHAR** setting is ignored. If the procedure is later changed back to LANGUAGE C, the setting of the **PARAMETER VARCHAR** option that was specified for the CREATE PROCEDURE statement (which is still in the catalog) will be used.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- DYNAMIC RESULT SET, RESULT SET, and RESULT SETS as synonyms for **DYNAMIC RESULT SETS**
- STANDARD CALL as a synonym for **DB2SQL**
- SIMPLE CALL as a synonym for **GENERAL**
- SIMPLE CALL WITH NULLS as a synonym for **GENERAL WITH NULLS**
- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**
- NULL CALL as a synonym for **CALLED ON NULL INPUT**
- PARAMETER STYLE DB2SQL as a synonym for **PARAMETER STYLE SQL**

Example

Assume that stored procedure SYSPROC.MYPROC is currently defined to run in WLM environment PARTSA and that you have appropriate authority on that WLM environment and WLM environment PARTSEC. Change the definition of the stored procedure so that it runs in PARTSEC.

```
ALTER PROCEDURE SYSPROC.MYPROC WLM ENVIRONMENT PARTSEC;
```

ALTER PROCEDURE (SQL - external)

The ALTER PROCEDURE statement changes the description, at the current server, of an external SQL procedure.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- Ownership of the stored procedure
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. The specified procedure name can include a schema name (a qualifier). However, if the schema name is not the same as the SQL authorization ID, one of the following conditions must be met:

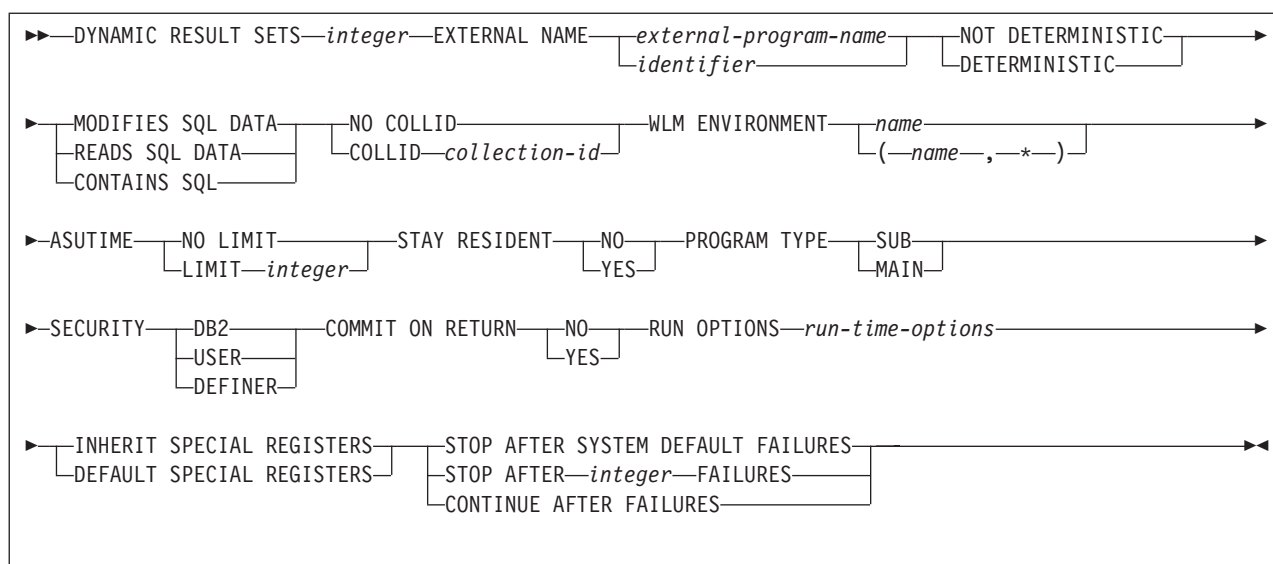
- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the ALTERIN privilege on the schema.

The SQL authorization ID that is used to alter the procedure definition must have appropriate authority for the WLM environment in which the procedure is currently defined to run. This authorization is obtained from an external security product, such as RACF.

Syntax

```
►►—ALTER PROCEDURE—procedure-name—| option-list |—————►◄
```

option-list: (Specify options in any order. Specify at least one option. Do not specify the same option more than once.)



Description

procedure-name

Identifies the stored procedure to be altered.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of query result sets that the procedure can return. The value must be between 0 and 32767.

EXTERNAL NAME *external-program-name* **or** *identifier*

Specifies the name of the MVS load module for the program that runs when the procedure name is specified in an SQL CALL statement. The value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

NOT DETERMINISTIC

The procedure might not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

DETERMINISTIC

The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, if the referenced data in the database has not changed.

DB2 does not verify that the procedure code is consistent with the specification of **DETERMINISTIC** or **NOT DETERMINISTIC**.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL

Specifies the classification of SQL statements that the procedure can execute. For the data access classification of each statement, see Table 138 on page 1605. Statements that are not supported in any procedure will return an error.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA

Specifies that procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the procedure can execute only SQL statements with an access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is to be used when the procedure is executed. This is the package collection into which the DBRM that is associated with the procedure is bound.

NO COLLID

Indicates that the package collection for the procedure is the same as the package collection of the calling program. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

COLLID *collection-id*

Specifies the package collection for the procedure.

WLM ENVIRONMENT *name* or (*name*,*)

Identifies the WLM (workload manager) environment in which the procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

name

Specifies the WLM environment in which the procedure must run. If another routine calls the procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the procedure request to a different address space.

(*name*,*)

When an SQL application program directly calls a procedure, *name* specifies the WLM environment in which the stored procedure runs.

If another routine calls the procedure, the procedure runs in the same WLM environment that the calling routine uses.

To change the environment in which the procedure is to run, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of a procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a procedure, setting a limit can be helpful in case the procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the number of CPU service units that the procedure can run.

LIMIT *integer*

The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure.

STAY RESIDENT

Specifies whether the load module for the procedure is to remain resident in memory when the procedure ends.

NO The load module is deleted from memory after the procedure ends.

YES

The load module remains resident in memory after the procedure ends.

PROGRAM TYPE

Specifies whether the procedure runs as a main routine or a subroutine. If PROGRAM TYPE is altered, the stored procedure needs to be re-compiled for the change to take effect.

SUB

The procedure runs as a subroutine.

MAIN

The procedure runs as a main routine.

SECURITY

Specifies how the procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2

The procedure does not require a special external security environment. If the procedure accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the address space in which the procedure runs.

USER

An external security environment should be established for the procedure. If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the procedure.

DEFINER

An external security environment should be established for the procedure. If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the procedure.

RUN OPTIONS *run-time-options*

Specifies the Language Environment runtime options that are to be used for the procedure. You must specify *run-time-options* as a character string that is no longer than 254 bytes. If you do not specify **RUN OPTIONS** or pass an empty string, DB2 does not pass any runtime options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

COMMIT ON RETURN

Indicates whether DB2 commits the transaction immediately on return from the procedure.

NO DB2 does not issue a commit when the procedure returns.

YES

DB2 issues a commit when the procedure returns if the following statements are true:

- A positive SQLCODE is returned by the CALL statement.
- The procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Specifies that special registers should be inherited according to the rules listed in the table for characteristics of special registers in a procedure in Table 37 on page 151.

DEFAULT SPECIAL REGISTERS

Specifies that special registers should be initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a procedure in Table 37 on page 151.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE

Specifies if the routine is stopped after failures.

STOP AFTER SYSTEM DEFAULT FAILURES

Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX.

STOP AFTER *nn* FAILURES

Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURES

Specifies that this routine should not be placed in a stopped state after any failure.

Notes

Changes are immediate: Any changes that the ALTER PROCEDURE statement causes to the definition of a procedure take effect immediately, with the exception of changes to the PROGRAM TYPE clause. If PROGRAM TYPE is changed, the stored procedure needs to be re-compiled for the change to take effect. The changed definition is used the next time that the procedure is called.

Changing to a native SQL procedure: You cannot change an external SQL procedure to a native SQL procedure. You can drop the procedure that you want to

| change using the DROP statement and create a native SQL procedure with a
| similar definition using the CREATE PROCEDURE statement. Alternatively, you
| can create a native SQL procedure using a different schema.

Invalidation of plans and packages: When an SQL procedure is altered, all the plans and packages that refer to that procedure are marked invalid.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 UDB family, DB2 supports the following keywords:

- RESULT SET, RESULT SETS, and DYNAMIC RESULT SET as synonyms for **DYNAMIC RESULT SETS**.
- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**

Example

Modify the definition for an SQL procedure so that SQL changes are committed on return from the SQL procedure and the SQL procedure runs in the WLM environment named WLMSQLP.

```
ALTER PROCEDURE UPDATE_SALARY_1  
  COMMIT ON RETURN YES  
  WLM ENVIRONMENT WLMSQLP;
```

ALTER PROCEDURE (SQL - native)

The ALTER PROCEDURE statement changes the definition of an SQL procedure at the current server. The procedure options, parameter names, and routine body can be changed and additional versions of the procedure can be defined and maintained using the ALTER PROCEDURE statement.

For information about the SQL control statements that are supported in native SQL procedures, refer to Chapter 6, “SQL control statements for native SQL procedures,” on page 1541.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

If the ALTER statement contains either a **REPLACE VERSION** or **ADD VERSION** clause, the statement can only be dynamically prepared.

Authorization

The privilege set that is defined below must include at least one of the following:

- Ownership of the procedure
- The ALTERIN privilege on the schema
- SYSADM authority
- SYSCTRL authority

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the procedure.

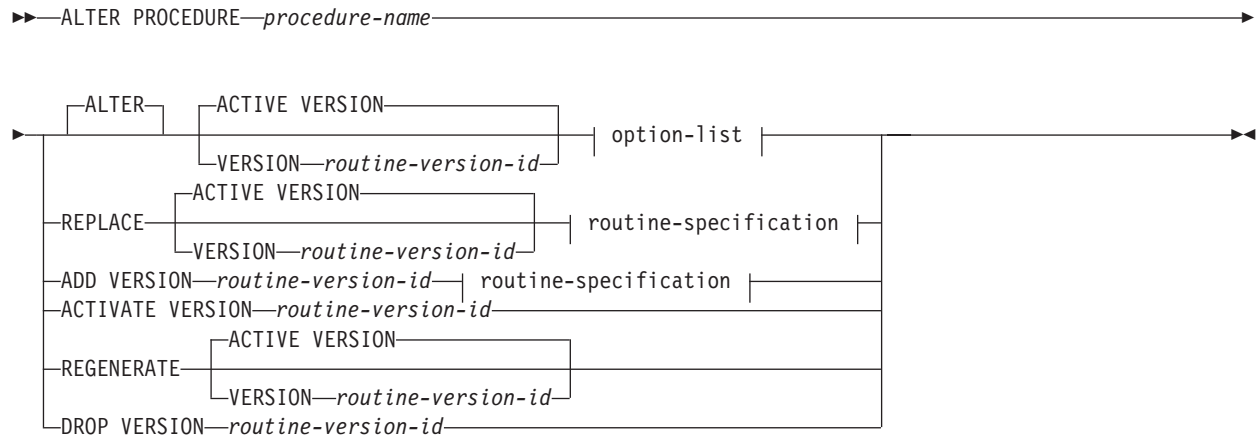
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. The specified procedure name can include a schema name (a qualifier). However, if the schema name is not the same as the SQL authorization ID, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the ALTERIN privilege on the schema.

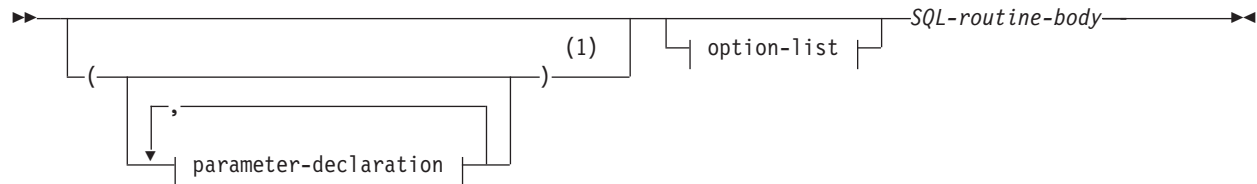
Additional privileges might be required in the following situations:

- If *SQL-routine-body* is specified, the privilege set must include the privileges that are required to execute the statements in *SQL-routine-body*.
- If the WLM ENVIRONMENT FOR DEBUG MODE clause is specified, the privilege set must include the authority to define programs that run in the specified WLM environment. This authorization is obtained from an external security product, such as RACF.
- When defining a new version of a procedure (using the ADD VERSION clause) or when replacing an existing version (using the REPLACE VERSION clause), the privilege set must include the required authorization to add a new package or a new version of an existing package depending on the value of the BIND NEW PACKAGE field on installation panel DSNTIPP, or the privilege set must include SYSADM or SYSCTRL authority.

Syntax



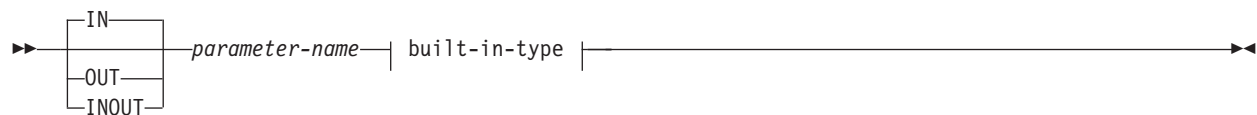
routine-specification:



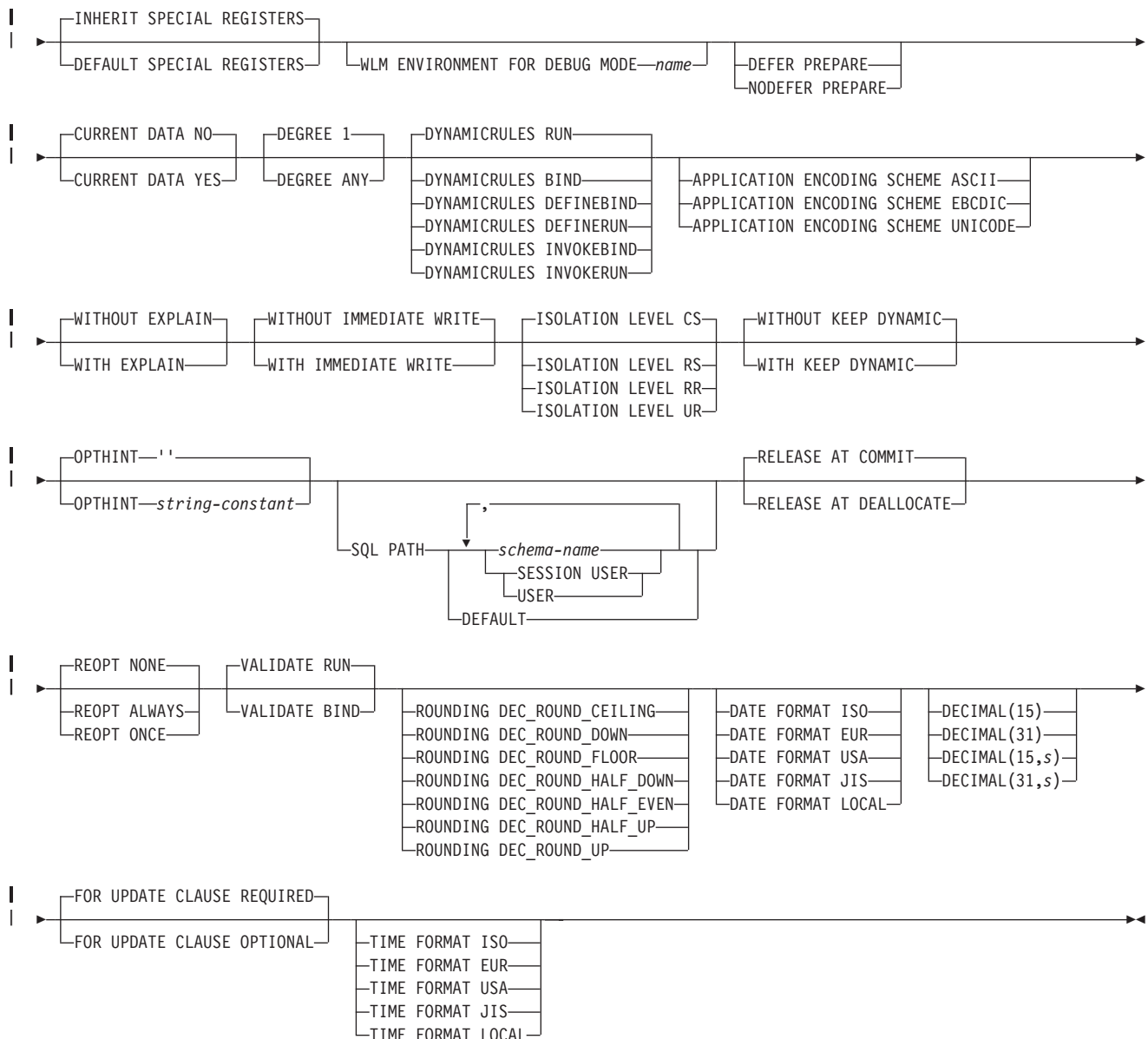
Notes:

- 1 All versions of the procedure must have the same number of parameters.

parameter-declaration:



built-in-type:



Description

procedure-name

Identifies the procedure to alter. The procedure that is identified in *procedure-name* must exist at the current server.

ACTIVE VERSION or **VERSION** *routine-version-id*

Identifies the version of the procedure that is to be changed, replaced, or regenerated depending on whether the ALTER, REPLACE, or REGENERATE keyword is specified.

ACTIVE VERSION

Specifies that the currently active version of the procedure is to be changed, replaced, or regenerated.

VERSION *routine-version-id*

Identifies the version of the procedure that is to be changed, replaced, or regenerated. *routine-version-id* is the version identifier that is assigned when

the version is defined. *routine-version-id* must identify a version of the specified procedure that exists at the current server.

ALTER

Specifies that a version of the procedure is to be changed.

When you change a procedure to add or replace a version of the procedure, any option that is not explicitly specified will use the existing value from the version of the procedure that is being changed.

REPLACE

Specifies that a version of the procedure is to be replaced.

Binding the replaced version of the procedure might result in a new access path even if the routine body is not changed.

When you replace a procedure, the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the procedure. For options that are not explicitly specified, the system default values for those options are used, even if those options were explicitly specified for the version of the procedure that is being replaced. This is not the case for versions of the procedure that specified DISABLE DEBUG MODE. If DISABLE DEBUG MODE is specified for a version of a procedure, it cannot be changed by the REPLACE clause.

ADD VERSION *routine-version-id*

Specifies that a new version of the procedure is to be created. *routine-version-id* is the version identifier for the new version of the procedure. *routine-version-id* must not identify a version of the specified procedure that already exists at the current server.

When a new version of a procedure is created, the comment that is recorded in the catalog for the new version will be the same as the comment that is in the catalog for the currently active version.

When you add a new version of a procedure the data types, CCSID specifications, and character data attributes (FOR BIT/SBCS/MIXED DATA) of the parameters must be the same as the attributes of the corresponding parameters for the currently active version of the procedure. The parameter names can differ from the other versions of the procedure. For options that are not explicitly specified, the system default values will be used.

ACTIVATE VERSION *routine-version-id*

Specifies the version of the procedure that is to be the currently active version of the procedure. *routine-version-id* is the version identifier that is assigned when the version of the procedure is defined. The version that is specified with *routine-version-id* is the version that will be invoked by the CALL statement, unless the value of the CURRENT ROUTINE VERSION special register overrides the currently active version of the procedure when the procedure is invoked. *routine-version-id* must identify a version of the procedure that already exists at the current server.

REGENERATE

Regenerates a version of the procedure. When DB2 maintenance is applied that changes how an SQL procedure is generated, the procedure might need to be regenerated to process the maintenance changes.

REGENERATE automatically rebinds, at the local server, the package for the SQL control statements for the procedure and rebinds the package for the SQL

statements that are included in the procedure body. If a remote bind is also needed, the BIND PACKAGE COPY command must be explicitly done for all of the remote servers.

REGENERATE is different from a REBIND PACKAGE command where the SQL statements are rebound (i.e. to generate better access paths for those statements), but the SQL control statements in the procedure definition remain the same.

DROP VERSION *routine-version-id*

Drops the version of the procedure that is identified with *routine-version-id*. *routine-version-id* is the version identifier that is assigned when the version is defined. *routine-version-id* must identify a version of the procedure that already exists at the current server and must not identify the currently active version of the procedure. Only the identified version of the procedure is dropped.

When only a single version of the procedure exists at the current server, use the DROP PROCEDURE statement to drop the procedure. A version of the procedure for which the version identifier is the same as the contents of the CURRENT ROUTINE VERSION special register can be dropped if that version is not the currently active version of the procedure.

(parameter-declaration,...)

Specifies the number of parameters of the procedure, the data type and usage of each parameter, and the name of each parameter for the version of the procedure that is being defined or changed. The number of parameters and the specified data type and usage of each parameter must match the data types in the corresponding position of the parameter for all other versions of this procedure. Synonyms for data types are considered to be a match.

IN, OUT, and INOUT specify the usage of the parameter. The usage of the parameters must match the implicit or explicit usage of the parameters of other versions of the same procedure.

IN Identifies the parameter as an input parameter to the procedure. The value of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT

Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT

Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned.

parameter-name

Names the parameter for use as an SQL variable. The name cannot be the same as the name of any other *parameter-name* for this version of the procedure. The name of the parameter in this version of the procedure can be different than the name of the corresponding parameter for other versions of this procedure.

built-in-type

Specifies the data type of the parameter. See “CREATE PROCEDURE (SQL - native)” on page 1046 for more information on data type specifications.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the procedure returns the same results each time it is called with the same IN and INOUT arguments.

NOT DETERMINISTIC

The procedure might not return the same result each time it is called with the same IN and INOUT arguments, even when the data that is referenced in the database has not changed.

NOT DETERMINISTIC is the default.

DETERMINISTIC

The procedure always returns the same results each time it is called with the same IN and INOUT arguments if the data that is referenced in the database has not changed.

DB2 does not verify that the procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL

Specifies the classification of SQL statements that the procedure can execute.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

MODIFIES SQL DATA is the default.

READS SQL DATA

Specifies that procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the procedure can execute only SQL statements with an access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

CALLED ON NULL INPUT

Specifies that the procedure will be called if any, or even if all parameter values are null.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of query result sets that the procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that there are no result sets. The value must be between 0 and 32767.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE

Specifies whether the version of the procedure can be run in debugging mode. The default for a new version of a procedure is determined using the value of the CURRENT DEBUG MODE special register.

ALLOW DEBUG MODE

Specifies that this version of the procedure can be run in debugging mode. When this version of the procedure is invoked and debugging is attempted, a WLM environment must be available.

DISALLOW DEBUG MODE

Specifies that the version of the procedure cannot be run in debugging mode.

You can use a subsequent ALTER PROCEDURE statement to change this option to ALLOW DEBUG MODE.

DISABLE DEBUG MODE

Specifies that the version of the procedure can never be run in debugging mode.

The version of the procedure cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE after the version of the procedure has been created, replaced, or altered to use DISABLE DEBUG MODE. To change DEBUG MODE for a version of a procedure that specifies DISABLE DEBUG MODE, you must drop and recreate the version of the procedure using the desired option.

When DISABLE DEBUG MODE is in effect, the WLM ENVIRONMENT FOR DEBUG MODE option is ignored.

PARAMETER CCSID

Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or Unicode. The value must be the same as the PARAMETER CCSID value determined when the procedure was created. The default value is the same as the PARAMETER CCSID value that was determined when the procedure was created.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value that is specified in all of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme that will be used for system-generated parameters of the routine.

QUALIFIER *schema-name*

Specifies the implicit qualifier that is used for unqualified names of tables, views, indexes, and aliases that are referenced in the procedure body. The default value is determined from the CURRENT SCHEMA special register.

PACKAGE OWNER *authorization-name*

Specifies the owner of the package that is associated with the version of the procedure. The SQL authorization ID of the process is the default value.

This authorization ID must have the privileges required to execute the SQL statements that are contained in the body of the routine. The value of the PACKAGE OWNER option is subject to translation when sent to a remote system.

If the privilege set lacks SYSADM or SYSCTRL authority, *authorization-name* must be the same as one of the authorization IDs of the process. If the privilege set includes SYSADM or SYSCTRL authority, *authorization-name* can be any authorization ID.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of a procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a procedure, setting a limit can be helpful in case the procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

Specifies that there is no limit on the number of CPU service units that the procedure can run.

NO LIMIT is the default.

LIMIT *integer*

Specifies that the limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure.

COMMIT ON RETURN NO or COMMIT ON RETURN YES

Indicates whether DB2 commits the transaction immediately on return from the procedure.

COMMIT ON RETURN NO

DB2 does not issue a commit when the procedure returns. NO is the default.

COMMIT ON RETURN YES,

DB2 issues a commit when the procedure returns if the following statements are true:

- The SQLCODE that is returned by the CALL statement is not negative.
- The procedure is not in a must-abort state.

The commit operation includes the work that is performed by the calling application process and by the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a procedure in Table 37 on page 151.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS

Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a procedure in Table 37 on page 151.

WLM ENVIRONMENT FOR DEBUG MODE *name*

Specifies the WLM (workload manager) application environment used by DB2 when debugging the procedure. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT FOR DEBUG MODE, DB2 uses the default WLM-established stored procedure address space that is specified at installation time.

The WLM ENVIRONMENT FOR DEBUG MODE value is ignored when DISABLE DEBUG MODE is in effect.

To change the environment that DB2 uses for debugging this procedure, you must have the appropriate authority for the WLM application environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

DEFER PREPARE or NODEFER PREPARE

Specifies whether to defer preparation of dynamic SQL statements that refer to remote objects, or to prepare them immediately.

The default depends on the value that is specified for the REOPT option. If REOPT NONE is specified, the default is NODEFER PREPARE. Otherwise, the default is DEFER PREPARE.

DEFER PREPARE

Specifies that the preparation of dynamic SQL statements that refer to remote objects will be deferred.

Refer to the DEFER(PREPARE) option in *DB2 Command Reference* for considerations with distributed processing.

NODEFER PREPARE

Specifies that the preparation of dynamic SQL statements that refer to remote objects will not be deferred.

CURRENT DATA

Specifies whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is in effect. CURRENT DATA also determines whether block fetch can be used for distributed, ambiguous cursors. For more information about updating the current row of a cursor, block fetch, and data currency, see *DB2 Application Programming and SQL Guide*.

YES

Specifies that data currency is required for read-only and ambiguous cursors. DB2 acquired page or row locks to ensure data currency. Block fetch is not allowed for distributed, ambiguous cursors.

NO Specifies that data currency is not required for read-only and ambiguous cursors. Block fetch is allowed for distributed, ambiguous cursors. Use of CURRENT DATA(NO) is not recommended if the procedure attempts to dynamically prepare and execute a DELETE WHERE CURRENT OF statement against an ambiguous cursor after that cursor is opened. You receive a negative SQLCODE if your procedure attempts to use a DELETE WHERE CURRENT OF statement for any of the following cursors:

- A cursor that is using block fetch
- A cursor that is using query parallelism
- A cursor that is positioned on a row that is modified by this or another application process

No is the default.

DEGREE

Specifies whether to attempt to run a query using parallel processing to maximize performance.

1 Specifies that parallel processing should not be used.

1 is the default.

ANY

Specifies that parallel processing can be used.

DYNAMICRULES

Specifies the values that apply, at run time, for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements

DYNAMICRULES also specifies whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements.

In addition to the value of the DYNAMICRULES clause, the runtime environment of a native SQL procedure controls how dynamic SQL statements behave at run time. The combination of the DYNAMICRULES value and the runtime environment determines the value for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The following values can be specified:

RUN

Specifies that dynamic SQL statements are to be processed using run behavior.

RUN is the default.

BIND

Specifies that dynamic SQL statements are to be processed using bind behavior.

DEFINEBIND

Specifies that dynamic SQL statements are to be processed using either define behavior or bind behavior.

DEFINERUN

Specifies that dynamic SQL statements are to be processed using either define behavior or run behavior.

INVOKEBIND

Specifies that dynamic SQL statements are to be processed using either invoke behavior or bind behavior.

INVOKERUN

Specifies that dynamic SQL statements are to be processed using either invoke behavior or run behavior.

See “Authorization IDs and dynamic SQL” on page 64 for information on the effects of these options.

APPLICATION ENCODING SCHEME

Specifies the default encoding scheme for SQL variables in static SQL statements in the procedure body. The value is used for defining an SQL variable in a compound statement if the CCSID clause is not specified as part of the data type, and the PARAMETER CCSID routine option is not specified.

ASCII

Specifies that the data is encoded using the ASCII CCSIDs of the server.

EBCDIC

Specifies that the data is encoded using the EBCDIC CCSIDs of the server.

UNICODE

Specifies that the data is encoded using the Unicode CCSIDs of the server.

See the ENCODING bind option in *DB2 Command Reference* for information about how the default for this option is determined.

WITH EXPLAIN or WITHOUT EXPLAIN

Specifies whether information will be provided about how SQL statements in the procedure will execute.

WITHOUT EXPLAIN

Specifies that information will not be provided about how SQL statements in the procedure will execute.

You can get EXPLAIN output for a statement that is embedded in a native SQL procedure that is specified using WITHOUT EXPLAIN by embedding the SQL statement EXPLAIN in the procedure body. Otherwise, the value of the EXPLAIN option applies to all explainable SQL statements in the procedure body, and to the fullselect portion of any DECLARE CURSOR statements.

WITHOUT EXPLAIN is the default.

WITH EXPLAIN

Specifies that information will be provided about how SQL statements in the procedure will execute. Information is inserted into the table *owner.PLAN_TABLE*. *owner* is the authorization ID of the owner of the procedure package. Alternatively, the authorization ID of the owner of the procedure can have an alias as *owner.PLAN_TABLE* that points to the base table, *PLAN_TABLE*. *owner* must also have the appropriate SELECT and INSERT privileges on that table. WITH EXPLAIN does not obtain information for statements that access remote objects. *PLAN_TABLE* must have a base table and can have multiple aliases with the same table name, *PLAN_TABLE*, but have different schema qualifiers; it cannot be a view or a synonym. It should exist before the version is added or replaced. In all inserts to *owner.PLAN_TABLE*, the value of QUERYNO is the statement number that is assigned by DB2.

The WITH EXPLAIN option also populates two optional tables, if they exist: *DSN_STATEMENT_TABLE* and *DSN_FUNCTION_TABLE*. *DSN_STATEMENT_TABLE* contains an estimate of the processing cost for an SQL statement. See *DB2 Application Programming and SQL Guide* for more information. *DSN_FUNCTION_TABLE* contains information about function resolution. See *DB2 Application Programming and SQL Guide* for more information.

For a description of the tables that are populated by the WITH EXPLAIN option, see “EXPLAIN” on page 1283.

WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE

Specifies whether immediate writes are to be done for updates that are made to group buffer pool dependent page sets or partitions. This option is only applicable for data sharing environments. The IMMEDIATEWRITE subsystem parameter has no effect of this option. *DB2 Command Reference* shows the implied hierarchy of the IMMEDIATEWRITE bind option (which is similar to this procedure option) as it affects run time.

WITHOUT IMMEDIATE WRITE

Specifies that normal write activity is performed. Updated pages that are group buffer pool dependent are written at or before phase one of commit or at the end of abort for transactions that have been rolled back.

WITHOUT IMMEDIATE WRITE is the default.

WITH IMMEDIATE WRITE

Specifies that updated pages that are group buffer pool dependent are immediately written as soon as the buffer update completes. Updated pages are written immediately even if the buffer is updated during forward progress or during the rollback of a transaction. WITH IMMEDIATE WRITE might impact performance.

ISOLATION LEVEL RR, RS, CS, or UR

Specifies how far to isolate the procedure from the effects of other running applications. For information about isolation levels, see *DB2 Performance Monitoring and Tuning Guide*.

RR Specifies repeatable read.

RS Specifies read stability.

CS Specifies cursor stability. CS is the default.

UR Specifies uncommitted read.

WITH KEEP DYNAMIC or WITHOUT KEEP DYNAMIC

Specifies whether DB2 keeps dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC

Specifies that DB2 does not keep dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC is the default.

WITH KEEP DYNAMIC

Specifies that DB2 keeps dynamic SQL statements after commit points. If you specify WITH KEEP DYNAMIC, the application does not need to prepare an SQL statement after every commit point. DB2 keeps the dynamic SQL statement until one of the following occurs:

- The application process ends
- A rollback operations occurs
- The application executes an explicit PREPARE statement with the same statement identifier as the dynamic SQL statement

If you specify WITH KEEP DYNAMIC, and the prepared statement cache is active, the DB2 subsystem keeps a copy of the prepared statement in the cache. If the prepared statement cache is not active, the subsystem keeps only the SQL statement string past a commit point. If the application executes an OPEN, EXECUTE, or DESCRIBE operation for that statement, the statement is implicitly prepared.

If you specify WITH KEEP DYNAMIC, DDF server threads that are used to execute procedures or packages that have this option in effect will remain active. Active DDF server threads are subject to idle thread time outs, as described in *DB2 Installation Guide* for install panel DSNTIPR.

If you specify WITH KEEP DYNAMIC, you must not specify REOPT ALWAYS. WITH KEEP DYNAMIC and REOPT ALWAYS are mutually exclusive. However, you can specify WITH KEEP DYNAMIC and REOPT ONCE.

Use WITH KEEP DYNAMIC to improve performance if your DRDA client application uses a cursor that is defined as WITH HOLD. The DB2 subsystem automatically closes a held cursor when there are no more rows to retrieve, which eliminates an extra network message.

OPTHINT *string-constant*

Specifies whether query optimization hints are used for static SQL statements that are contained within the body of the procedure.

string-constant is a character string of up to 128 bytes in length, which is used by the DB2 subsystem when searching the PLAN_TABLE for rows to use as input. The default value is an empty string, which indicates that the DB2 subsystem does not use optimization hints for static SQL statements.

Optimization hints are only used if optimization hints are enabled for your system. See *DB2 Installation Guide* for information about enabling optimization hints.

SQL PATH

Specifies the SQL path that the DB2 subsystem uses to resolve unqualified user-defined distinct types, functions, and procedure names (in CALL statements) in the body of the procedure.

The maximum length of the SQL path is 2048 bytes. DB2 calculates the length by taking each *schema-name* specified and removing any trailing blanks from it, adding two delimiters around it, and adding one comma after each schema name except for the last one. The length of the resulting string cannot exceed 2048 bytes.

schema-name

Specifies a schema. DB2 does not validate that the specified schema actually exists when the ALTER statement is processed.

schema-name-list

Specifies a comma separated list of schema names. The same schema name should not appear more than one time in the list of schema names. The number of schema names that you can specify is limited by the maximum length of the resulting SQL path.

SESSION_USER or USER

Specifies the value of the SESSION_USER (or USER) special register. At the time the ALTER statement is processed, the actual length is included in the total length of the list of schema names that is specified for the PATH option. If you specify SESSION_USER (or USER) in a list of schema names, do not use delimiters around the SESSION_USER (or USER) keyword.

DEFAULT

Specifies that the SQL Path should be set to "SYSIBM", "SYSFUN", "SYSPROC", *procedure-schema*. *procedure-schema* is the schema qualifier for the procedure that is being altered.

If you specify DEFAULT, you do not need to explicitly specify the SYSIBM, SYSFUN, and SYSPROC schemas; these schemas are implicitly added to the beginning of the SQL path in the order listed. If you do not specify the SYSIBM, SYSFUN, and SYSPROC, schemas, they are not included in the length of the SQL path.

RELEASE AT

Specifies when to release resources that the procedure uses: either at each commit point or when the procedure terminates.

COMMIT

Specifies that resources will be released at each commit point.

COMMIT is the default.

DEALLOCATE

Specifies that resources will be released only when the procedure terminates. DEALLOCATE has no effect on packages that run on a DB2 server through a DRDA connection with a client system. DEALLOCATE also has no effect on dynamic SQL statements, which always use RELEASE AT COMMIT, with this exception: When you use the RELEASE AT DEALLOCATE clause and the WITH KEEP DYNAMIC clause, and the subsystem is installed with a value of YES for the field CACHE DYNAMIC

SQL on installation panel DSNTIP8, the RELEASE AT DEALLOCATE option is honored for dynamic SELECT and SQL data change statements.

Locks that are acquired for dynamic statements are held until one of the following events occurs:

- The application process ends.
- The application process issues a PREPARE statement with the same statement identifier. (Locks are released at the next commit point).
- The statement is removed from the prepared statement cache because the statement has not been used. (Locks are released at the next commit point).
- An object that the statement is dependent on is dropped or altered, or a privilege that the statement needs is revoked. (Locks are released at the next commit point).

RELEASE AT DEALLOCATE can increase the package size because additional items become resident in the package. For more information about how the RELEASE clause affects locking and concurrency, see *DB2 Performance Monitoring and Tuning Guide*.

REOPT

Specifies if DB2 will determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE

Specifies that DB2 does not determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE is the default.

ALWAYS

Specifies that DB2 always determine the access path at run time each time an SQL statement is run.

ONCE

Specifies that DB2 determine the access path for any dynamic SQL statements only one time, at the first time the statement is opened. This access path is used until the prepared statement is invalidated or removed from the dynamic statement cache and needs to be prepared again.

VALIDATE RUN or VALIDATE BIND

Specifies whether to recheck, at run time, errors of the type "OBJECT not FOUND" and NOT AUTHORIZED" that are found during bind or rebind. The option has no effect if all objects and needed privileges exist.

VALIDATE RUN

Specifies that if needed objects or privileges do not exist when the ALTER PROCEDURE statement is processed, warning messages are returned, but the ALTER PROCEDURE statement succeeds. The DB2 subsystem rechecks for the objects and privileges at run time for those SQL statements that failed the checks during processing of the ALTER PROCEDURE statement. The authorization checks the use of the authorization ID of the owner of the procedure package.

VALIDATE RUN is the default.

VALIDATE BIND

Specifies that if needed objects or privileges do not exist at the time the ALTER PROCEDURE statement is processed, an error is issued and the ALTER PROCEDURE statement fails.

ROUNDING

Specifies the desired rounding mode for manipulation of DECFLOAT data.

DEC_ROUND_CEILING

Specifies numbers are rounded towards positive infinity.

DEC_ROUND_DOWN

Specifies numbers are rounded towards 0 (truncation).

DEC_ROUND_FLOOR

Specifies numbers are rounded towards negative infinity.

DEC_ROUND_HALF_DOWN

Specifies numbers are rounded to nearest; if equidistant, round down.

DEC_ROUND_HALF_EVEN

Specifies numbers are rounded to nearest; if equidistant, round so that the final digit is even.

DEC_ROUND_HALF_UP

Specifies numbers are rounded to nearest; if equidistant, round up.

DEC_ROUND_UP

Specifies numbers are rounded away from 0.

DATE FORMAT ISO, EUR, USA, JIS, or LOCAL

Specifies the date format for result values that are string representations of date or time values. See “String representations of datetime values” on page 89 for more information.

The default format is specified in the DATE FORMAT field of installation panel DSNTIP4 of the system where the procedure is defined. You cannot use the LOCAL option unless you have a date exit routine.

DECIMAL(15), DECIMAL(31), DECIMAL(15,s), or DECIMAL(31,s)

Specifies the maximum precision that is to be used for decimal arithmetic operations. See “Arithmetic with two decimal operands” on page 183 for more information. The default format is specified in the DECIMAL ARITHMETIC field of installation panel DSNTIPF of the system where the procedure is defined. If the form *pp.s* is specified, *s* must be a number between 1 and 9. *s* represents the minimum scale that is to be used for division.

FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED

Specifies whether the FOR UPDATE clause is required for a DECLARE CURSOR statement if the cursor is to be used to perform positioned updates.

FOR UPDATE CLAUSE REQUIRED

Specifies that a FOR UPDATE clause must be specified as part of the cursor definition if the cursor will be used to make positioned updates.

FOR UPDATE CLAUSE REQUIRED is the default.

FOR UPDATE CLAUSE OPTIONAL

Specifies that the FOR UPDATE clause does not need to be specified in order for a cursor to be used for positioned updates. The procedure body can include positioned UPDATE statements that update columns that the user is authorized to update.

If the resulting DBRM for the procedure is very large, you might need extra storage when you specify FOR UPDATE CLAUSE OPTIONAL.

The FOR UPDATE clause of the select-statement with no column list applies to static or dynamic SQL statements. You can specify the FOR UPDATE OF clause of the select-statement with a column list to restrict updates to only the columns that are named in the column list and to specify the acquisition of update locks.

TIME FORMAT ISO, EUR, USA, JIS, or LOCAL

Specifies the time format for result values that are string representations of date or time values. See “String representations of datetime values” on page 89 for more information.

The default format is specified in the TIME FORMAT field of installation panel DSNTIP4 of the system where the procedure is defined. You cannot use the LOCAL option unless you have a date exit routine.

SQL-routine-body

Specifies the statements that define the body of the SQL procedure. For information on the SQL control statements that are supported in native SQL procedures, see Chapter 6, “SQL control statements for native SQL procedures,” on page 1541. If an *SQL-procedure-statement* is the only statement in the procedure body, the statement must not end with a semicolon. For information on the SQL statements that are allowed in SQL procedures, see “SQL statements allowed in SQL procedures” on page 1608.

Notes

Changes are immediate: Any changes that the ALTER PROCEDURE statement causes to the definition of a procedure take effect immediately. The changed definition is used the next time that the procedure is called.

Considerations for altering a version of a procedure: To alter a version of a procedure, the environment settings that are in effect when the ALTER PROCEDURE statement is issued must be the same as the environment settings that are in effect when the version of the procedure is first created using the CREATE PROCEDURE or ALTER PROCEDURE statements if one of the following options is specified:

- **QUALIFIER**
- **PACKAGE OWNER**
- **WLM ENVIRONMENT FOR DEBUG MODE**
- **OPTHINT**
- **SQL PATH**
- **DECIMAL** (if the value includes a comma)

Changing to a native SQL procedure: You cannot change an external SQL procedure to a native SQL procedure. You can drop the external SQL procedure that you want to change by using the DROP statement and create a native SQL procedure with a similar definition using the CREATE PROCEDURE statement. Alternatively, you can create a native SQL procedure using a different schema.

Identifier resolution: See Chapter 6, “SQL control statements for native SQL procedures,” on page 1541 for information on how names are resolved to columns, SQL variables, or SQL routines for native SQL procedures. Name resolution is unchanged for external SQL procedures.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the procedure name for parameters, and the label name for SQL variables.

Characteristics of the package that is generated for a version of a procedure: The package that is associated with a version of a procedure is named as follows:

- *location* is set to the value of the CURRENT SERVER special register
- *collection-id* (schema) for the package is the same as the schema qualifier of the procedure
- *package-id* is the same as the specific name of the procedure
- *version-id* is the same as the version identifier for the initial version of the procedure

The package is generated using the bind options that correspond to the implicitly or explicitly specified procedure options. See Table 83 for more information. In addition to the corresponding bind options, the package is generated using the following bind options:

- DBPROTOCOL(DRDA)
- FLAG(1)
- SQLERROR(NOPACKAGE)
- ENABLE(*)

Considerations for a procedure that is defined using a TABLE LIKE name AS LOCATOR clause: If a procedure is defined with a table parameter (the **TABLE LIKE name AS LOCATOR** clause was specified in the CREATE PROCEDURE statement to indicate that one of the input parameters is a transition table), the procedure cannot be changed with an ALTER PROCEDURE statement if the change requires that the parameter list be specified. For example, to add or replace a version of a native SQL procedure, the procedure must be dropped and recreated.

Correspondence of procedure options to BIND options: The following table lists options for CREATE PROCEDURE and ALTER PROCEDURE and the corresponding options for the bind commands. See *DB2 Command Reference* for more information about the effects of the options of the bind commands.

Table 83. Correspondence of procedure options to bind options

CREATE PROCEDURE or ALTER PROCEDURE option	bind commands option
APPLICATION ENCODING SCHEME	ENCODING(ASCII) ENCODING(EBCDIC) ENCODING(UNICODE)
CURRENT DATA NO	CURRENTDATA(NO)
CURRENT DATA YES	CURRENTDATA(YES)
DEFER PREPARE	DEFER(PREPARE)
NODEFER PREPARE	NODEFER(PREPARE)
DEGREE	DEGREE(ANY) DEGREE(1)

Table 83. Correspondence of procedure options to bind options (continued)

CREATE PROCEDURE or ALTER PROCEDURE option	bind commands option
DYNAMICRULES	DYNAMICRULES(RUN) DYNAMICRULES(BIND) DYNAMICRULES(DEFINEBIND) DYNAMICRULES(DEFINERUN) DYNAMICRULES(INVOKEBIND) DYNAMICRULES(INVOKERUN)
ISOLATION LEVEL	ISOLATION(RR) ISOLATION(RS) ISOLATION(CS) ISOLATION(UR)
OPTHINT	OPTHINT
PACKAGE OWNER	OWNER
QUALIFIER	QUALIFIER
RELEASE AT COMMIT	RELEASE(COMMIT)
RELEASE AT DEALLOCATE	RELEASE(DEALLOCATE)
REOPT ALWAYS	REOPT(ALWAYS)
REOPT NONE	REOPT(NONE)
REOPT ONCE	REOPT(ONCE)
ROUNDING DEC_ROUND_CEILING	ROUNDING(CEILING)
ROUNDING DEC_ROUND_DOWN	ROUNDING(DOWN)
ROUNDING DEC_ROUNDING_FLOOR	ROUNDING(FLOOR)
ROUNDING DEC_ROUNDING_HALF_DOWN	ROUNDING(HALFDOWN)
ROUNDING DEC_ROUNDING_HALF_EVEN	ROUNDING(HALFEVEN)
ROUNDING DEC_ROUNDING_HALF_UP	ROUNDING(HALFUP)
ROUNDING DEC_ROUNDING_UP	ROUNDING(UP)
SQL PATH	PATH
VALIDATE BIND	VALIDATE(BIND)
VALIDATE RUN	VALIDATE(RUN)
WITH EXPLAIN	EXPLAIN(YES)
WITHOUT EXPLAIN	EXPLAIN(NO)
WITH IMMEDIATE WRITE	IMMEDWRITE(YES)
WITHOUT IMMEDIATE WRITE	IMMEDWRITE(NO)
WITH KEEP DYNAMIC	KEEP DYNAMIC(YES)
WITHOUT KEEP DYNAMIC	KEEP DYNAMIC(NO)

Invalidation of plans and packages: When a version of an SQL procedure is altered to change any option that is specified for the active version, all the plans and packages that refer to that procedure are marked invalid. Additionally, when

certain attributes of a native SQL procedure are changed, the body of the procedure might be rebound or regenerated. The following table summarizes when implicit rebind and regeneration occurs when specific options are changed. A value of Y in a row indicates that a rebind or regeneration will occur if the option is changed for a version of the procedure.

Table 84. CREATE PROCEDURE and ALTER PROCEDURE options that result in rebind or regeneration when changed.

CREATE PROCEDURE or ALTER PROCEDURE option	Change requires rebind of invoking applications?	Change results in implicit rebind of the non-control statements of the body of the procedure?	Change results in implicit regeneration of the entire body of the procedure?
ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE	Y ^{1, 2}	Y ¹	Y
APPLICATION ENCODING SCHEME	Y	Y	Y
ASUTIME	Y		
COMMIT ON RETURN	Y		
CURRENT DATA		Y	
DATE FORMAT	Y	Y	Y
DECIMAL	Y	Y	Y
DEFER PREPARE or NODEFER PREPARE		Y	
DEGREE		Y	
DYNAMIC RESULT SETS	Y		
DYNAMICRULES		Y	
FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED	Y	Y	Y
INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS	Y		
ISOLATION LEVEL		Y	
MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL	Y	Y	Y
NOT DETERMINISTIC or DETERMINISTIC			
OPTHINT		Y	
PACKAGE OWNER		Y	
QUALIFIER		Y	
RELEASE AT COMMIT or RELEASE AT DEALLOCATE		Y	
REOPT		Y	
SQL PATH		Y	

Table 84. CREATE PROCEDURE and ALTER PROCEDURE options that result in rebind or regeneration when changed. (continued)

CREATE PROCEDURE or ALTER PROCEDURE option	Change requires rebind of invoking applications?	Change results in implicit rebind of the non-control statements of the body of the procedure?	Change results in implicit regeneration of the entire body of the procedure?
STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER nn FAILURES, or CONTINUE AFTER FAILURES	Y		
TIME FORMAT	Y	Y	Y
VALIDATE RUN or VALIDATE BIND		Y	
WITH EXPLAIN or WITHOUT EXPLAIN		Y	
WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE		Y	
WITH KEEP DYNAMIC or WITHOUT KEEP DYNAMIC		Y	
WLM ENVIRONMENT FOR DEBUG MODE	Y		
Note:			
1. The procedure package is rebound or regenerated if a value of ALLOW DEBUG MODE is changed to DISALLOW DEBUG MODE.			
2. Invoking applications are invalidated if a value of DISALLOW DEBUG MODE is changed to DISABLE DEBUG MODE.			

Compatibilities: For compatibility with previous versions of DB2, the following clauses can be specified, but they will be ignored and an error or a warning will be issued. If ALTER is implicitly or explicitly specified with one of these options specified as part of *option-list*, a warning is issued and the option is ignored. If REPLACE or ADD VERSION is specified with one of these options specified as part of *option-list*, and error is issued. For example, ADD VERSION is specified and STAY RESIDENT is specified as part of option-list and error is issued.

- STAY RESIDENT
- PROGRAM TYPE
- RUN OPTIONS
- NO DBINFO
- COLLID or NOCOLLID
- SECURITY
- PARAMETER STYLE GENERAL WITH NULLS
- STOP AFTER SYSTEM DEFAULT FAILURES
- STOP AFTER *nn* FAILURES
- CONTINUE AFTER FAILURES

If WLM ENVIRONMENT is specified for a native SQL procedure, WLM ENVIRONMENT FOR DEBUG MODE must be specified.

For compatibility with the CREATE PROCEDURE statement, the following clause can be specified, but will be ignored:

- **LANGUAGE SQL**

Considerations for SYSENVIRONMENTS catalog table: An ALTER statement that specifies a new environment settings will result in a new row being added to the SYSENVIRONMENTS catalog table. The new row will be added even if an error is subsequently encountered during processing of the ALTER statement. Thus, a new SYSENVIRONMENTS row might be added even for an ALTER statement that fails.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET, RESULT SETS, and DYNAMIC RESULT SET as synonyms for **DYNAMIC RESULT SETS**.
- VARIANT as a synonym for **NOT DETERMINISTIC**
- NOT VARIANT as a synonym for **DETERMINISTIC**

Example

Example 1: The following statement changes the existing procedure options for the active version of the UPDATE_SALARY_1 native SQL procedure. If you need to change a different version of the procedure, you would specify *VERSION routine-version-id* in place of ACTIVE VERSION. Note that the ALTER clause that precedes the version specification can be omitted.

```
ALTER PROCEDURE UPDATE_SALARY_1
  ALTER ACTIVE VERSION
  NOT DETERMINISTIC
  CALLED ON NULL INPUT
  ALLOW DEBUG MODE
  ASUTIME LIMIT 10
```

Example 2: To change the procedure body of any existing version of a procedure, you need to use the REPLACE clause. The following statement changes both the procedure body and the existing SQL data access option for version V2 of the UPDATE_SALARY_1 SQL procedure. Note that the list of parameters is specified even though no changes are made to the list. To replace an existing version of a procedure, you must specify the list of parameters, any options that are to have non-default values (even if those options are specified in the version of the procedure that you are replacing), and the body of the procedure.

```
ALTER PROCEDURE UPDATE_SALARY_1
  REPLACE VERSION V2 (P1 INTEGER, P2 CHAR(5))
  MODIFIES SQL DATA
  UPDATE EMP SET SALARY = SALARY * RATE
  WHERE EMPNO = EMPLOYEE_NUMBER;
```

Example 3: To add a new version of an existing procedure, use the ADD VERSION clause. The following statement adds a new version of the UPDATE_SALARY_1 procedure to apply a larger salary increase. Note that the list of parameters is specified even though the new version of the procedure uses the same parameters as the existing version of the procedure. To add a new version of a procedure, you must specify the list of parameters, any options that will have non-default values, and the body of the procedure.

```
ALTER PROCEDURE UPDATE_SALARY_1
  ADD VERSION V3 (P1 INTEGER, P2 CHAR(5))
  UPDATE EMP SET SALARY = SALARY * (RATE*10)
  WHERE EMPNO = EMPLOYEE_NUMBER;
```

| *Example 4:* When the new version of the procedure has been defined, as in
| Example 3, you must use the ALTER PROCEDURE statement with the ACTIVATE
| VERSION clause if the new version of the procedure is to be the currently active
| version, as in the following example.

| ALTER PROCEDURE UPDATE_SALARY_1
| ACTIVATE VERSION V3;

| *Example 5:* To regenerate the currently active version of a procedure, use the
| following statement.

| ALTER PROCEDURE UPDATE_SALARY_1
| REGENERATE ACTIVE VERSION;

ALTER SEQUENCE

The ALTER SEQUENCE statement changes the attributes of a sequence at the current server. Only future values of the sequence are affected by the ALTER SEQUENCE statement.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

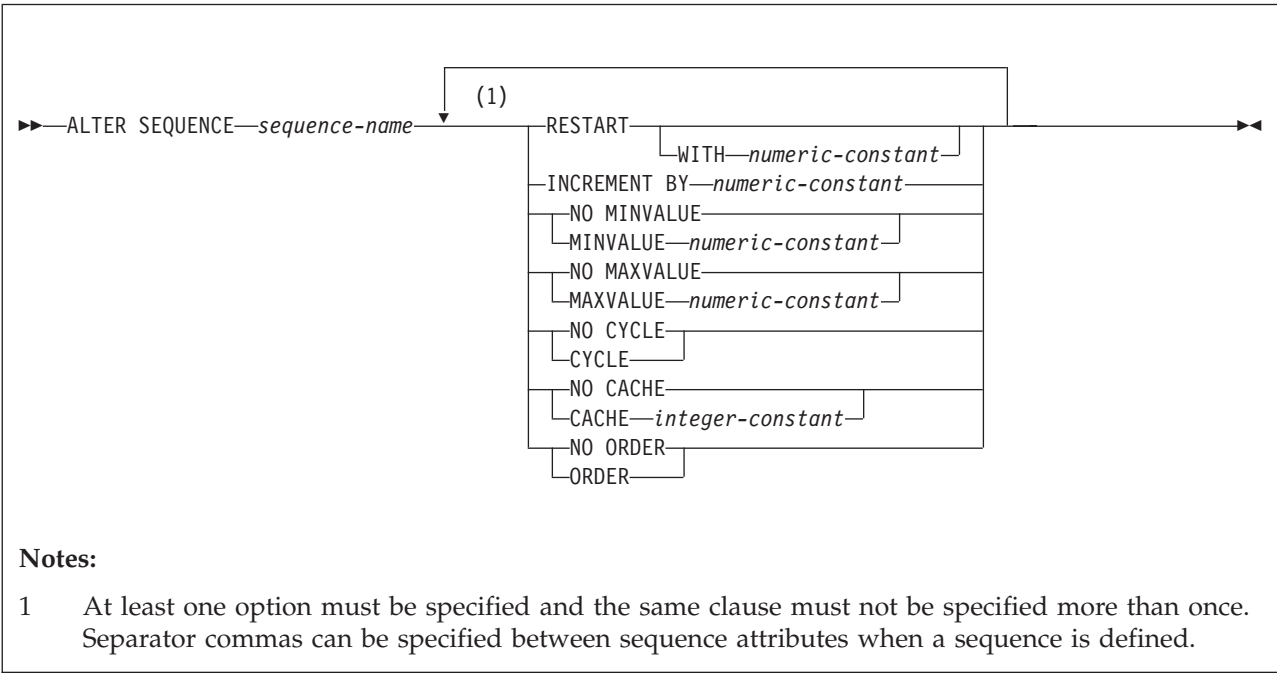
The privilege set that is defined below must include at least one of the following:

- Ownership of the sequence
- The ALTER privilege for the sequence
- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process.

Syntax



Description

sequence-name

Identifies the sequence. The combination of sequence name and the implicit or

explicit qualifier must identify an existing sequence at the current server.
sequence-name must not identify a sequence that is generated by DB2 for an identity column or a DB2_GENERATED_DOCID_FOR_XML column.

RESTART

Restarts the sequence. If *numeric-constant* is not specified, the sequence is restarted at the value specified implicitly or explicitly as the starting value on the CREATE SEQUENCE statement that originally created the sequence.

WITH *numeric-constant*

Specifies the value at which to restart the sequence. The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point.

If RESTART is not specified, the sequence is not restarted. Instead, it resumes with the current values in effect for all the options after the ALTER statement is issued.

After a sequence is restarted or changed to allow cycling, sequence numbers might be duplicates of values generated by the sequence previously.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the sequence. The value can be any positive or negative value (including 0) that could be assigned to a column of the data type that is associated with the sequence without any non-zero digits existing to the right of the decimal point.

If INCREMENT BY *numeric-constant* is positive, the sequence ascends. If INCREMENT BY *numeric-constant* is negative, the sequence descends. If INCREMENT BY *numeric-constant* is 0, the sequence is treated as an ascending sequence.

The absolute value of INCREMENT BY can be greater than the difference between MAXVALUE and MINVALUE.

NO MINVALUE or MINVALUE

Specifies whether or not there is a minimum end point of the range of values for the sequence.

NO MINVALUE

Specifies that the minimum end point of the range of values for the sequence has not been specified explicitly. In such a case, the value for MINVALUE becomes one of the following:

- For an ascending sequence, the value is the original starting value.
- For a descending sequence, the value is the minimum of the data type that is associated with the sequence.

MINVALUE *numeric-constant*

Specifies the minimum value at which a descending sequence either cycles or stops generating values, or an ascending sequence cycles to after reaching the maximum value. The last value that is generated for a cycle of a descending sequence will be equal to or greater than this value. MINVALUE is the value to which an ascending sequence cycles to after reaching the maximum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

NO MAXVALUE or MAXVALUE

Specifies whether or not there is a maximum end point of the range of values for the sequence.

NO MAXVALUE

Specifies either explicitly or implicitly that the minimum end point of the range of values for the sequence has not be set. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending sequence, the value is the maximum value of the data type that is associated with the sequence
- For a descending sequence, the value is the original starting value.

If NO MAXVALUE is explicitly specified in the ALTER SEQUENCE statement, the value of the MAXVALUE column in the catalog table is reset to the maximum value of the data type associated with the sequence if the sequence is ascending or the value stored in the START column of the catalog table if the sequence is descending. Whether the sequence is ascending or descending depends on whether or not the INCREMENT BY option is reset. If it is, the new INCREMENT BY VALUE determines if the sequence is ascending or descending. If it is not explicitly reset, the value stored in the INCREMENT column of the catalog table determines if the sequence is ascending or descending.

MAXVALUE *numeric-constant*

Specifies the maximum value at which an ascending sequence either cycles or stops generating values or a descending sequence cycles to after reaching the minimum value. The last value that is generated for a cycle of an ascending sequence will be less than or equal to this value. MAXVALUE is the value to which a descending sequence cycles to after reaching the minimum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

NO CYCLE or CYCLE

Specifies whether or not the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition or by overshooting it.

NO CYCLE

Specifies that the sequence cannot generate more values once the maximum or minimum value for the sequence has been reached.

CYCLE

Specifies that the sequence continue to generate values after either the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum value, it generates its maximum value. The maximum and minimum values for the sequence defined by the MINVALUE and MAXVALUE options determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by the sequence. When a sequence is defined with CYCLE, any application

conversion tools for converting applications from other vendor platforms to DB2 should also explicitly specify MINVALUE, MAXVALUE, and START WITH values.

NO CACHE or CACHE

Specifies whether or not to keep some preallocated values in memory for faster access. This is a performance and tuning option.

NO CACHE

Specifies that values of the sequence are not to be preallocated. This option ensures that there is not a loss of values in the case of a system failure. When NO CACHE is specified, the values of the sequence are not stored in the cache. In this case, every request for a new value for the sequence results in synchronous I/O.

CACHE *integer-constant*

Specifies the maximum number of sequence values that DB2 can preallocate and keep in memory. Preallocating values in the cache reduces synchronous I/O when values are generated for the sequence. The actual number of values that DB2 caches is always the lesser of the number in effect for the CACHE option and the number of remaining values within the logical range. Thus, the CACHE value is essentially an upper limit for the size of the cache.

In the event the system is shut down (either normally or through a system failure), all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of sequence values that could be lost when the system is shut down.

The minimum value is 2.

In a data sharing environment, you can use the CACHE and NO ORDER options to allow multiple DB2 members to cache sequence values simultaneously.

NO ORDER or ORDER

Specifies whether the sequence numbers must be generated in order of request.

NO ORDER

Specifies that the sequence numbers do not need to be generated in order of request.

ORDER

Specifies that the sequence numbers are generated in order of request. Specifying ORDER might disable the caching of values. There is no guarantee that values are assigned in order across the entire server unless NO CACHE is also specified. ORDER applies only to a single-application process.

In a data sharing environment, if the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously, and the requests for next value assignments from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the same sequence, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested for next value first, then DB2B requested, and then DB2A again requested. Therefore, to

guarantee that sequence numbers are generated in strict numeric order among multiple DB2 members using the same sequence concurrently, specify the ORDER option.

Notes

Altering a sequence: The changes to the attributes of a sequence take effect after the ALTER SEQUENCE statement is committed. Only future sequence numbers are affected by the ALTER SEQUENCE statement. If the ALTER SEQUENCE request results in an error or is rolled back, nothing is changed; however, unused cache values might be lost.

- The data type of a sequence cannot be changed. Instead, drop and recreate the sequence specifying the desired data type for the new sequence.
- All cached values are lost when a sequence is altered.
- After restarting a sequence or changing it to cycle, it is possible that a generated value will duplicate a value previously generated for that sequence.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- NOCACHE (single key word) as a synonym for NO CACHE
- NOCYCLE (single key word) as a synonym for NO CYCLE
- NOMINVALUE (single key word) as a synonym for NO MINVALUE
- NOMAXVALUE (single key word) as a synonym for NO MAXVALUE
- NOORDER (single key word) as a synonym for NO ORDER

Examples

Example 1: Reset a sequence to the START WITH value to generate the numbers from 1 up to the number of rows in the table:

```
ALTER SEQUENCE org_seq  
  RESTART;
```

ALTER STOGROUP

The ALTER STOGROUP statement changes the description of a storage group at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

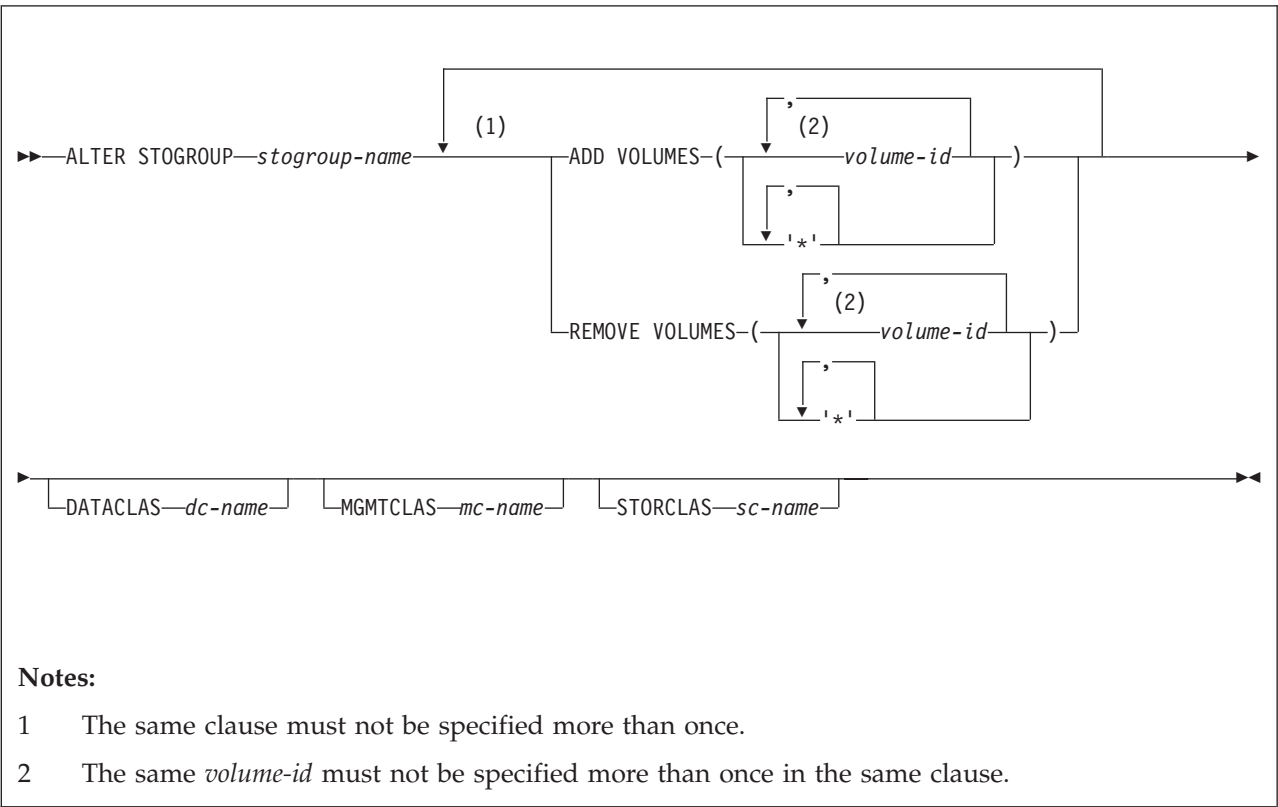
Authorization

The privilege set that is defined below must include one of the following:

- Ownership of the storage group
- SYSADM or SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Syntax



Description

stogroup-name
Identifies the storage group to be altered. The name must identify a storage group that exists at the current server.

ADD VOLUMES(*volume-id*,...) or ADD VOLUMES('*',...)

Adds volumes to the storage group. Each *volume-id* is the volume serial number of a storage volume to be added. It can have a maximum of six characters and is specified as an identifier or a string constant.

A *volume-id* must not be specified if any volume of the storage group is designated by an asterisk (*). An asterisk must not be specified if any volume of the storage group is designated by a *volume-id*.

You cannot add a volume that is already in the storage group unless you first remove it with REMOVE VOLUMES.

Asterisks are recognized only by Storage Management Subsystem (SMS). If the data set that is associated with the storage group is non SMS managed, either ADD VOLUMES or REMOVE VOLUMES must be specified. Neither ADD VOLUMES or REMOVE VOLUMES is required if DATACLAS, MGMTCLAS, or STORCLAS is specified. SMS usage is recommended, rather than using DB2 to allocate data to specific volumes. Having DB2 select the volume requires non-SMS usage or assigning an SMS Storage Class with guaranteed space. However, because guaranteed space reduces the benefits of SMS allocation, it is not recommended.

If you do choose to use specific volume assignments, additional manual space management must be performed. Free space must be managed for each individual volume to prevent failures during the initial allocation and extension. This process generally requires more time for space management and results in more space shortages. Guaranteed space should be used only where the space needs are relatively small and do not change.

REMOVE VOLUMES(*volume-id*,...) or REMOVE VOLUMES('*',...)

Removes volumes from the storage group. Each *volume-id* is the volume serial number of a storage volume to be removed. Each *volume-id* must identify a volume that is in the storage group.

The REMOVE VOLUMES clause is applied to the current list of volumes before the ADD VOLUMES clause is applied. Removing a volume from a storage group does not affect existing data, but a volume that has been removed is not used again when the storage group is used to allocate storage for table spaces or index spaces.

Asterisks are recognized only by Storage Management Subsystem (SMS). If the data set that is associated with the storage group is non SMS managed, either ADD VOLUMES or REMOVE VOLUMES must be specified. Neither ADD VOLUMES or REMOVE VOLUMES is required if DATACLAS, MGMTCLAS, or STORCLAS is specified.

DATACLAS *dc-name*

Identifies the name of the SMS data class to associate with the DB2 storage group. The SMS data class name must be from 1-8 characters in length. The SMS storage administrator defines the data class that can be used. DATACLAS must not be specified more than one time.

MGMTCLAS *mc-name*

Identifies the name of the SMS management class to associate with the DB2 storage group. The SMS management class name must be from 1-8 characters in length. The SMS storage administrator defines the management class that can be used. MGMTCLAS must not be specified more than one time.

STORCLAS *sc-name*

Identifies the name of the SMS storage class to associate with the DB2 storage group. The SMS storage class name must be from 1-8 characters in length. The

SMS storage administrator defines the storage class that can be used.
STORCLAS must not be specified more than one time.

Notes

Work file databases: If the storage group altered contains data sets in a work file database, the database must be stopped and restarted for the effects of the ALTER to be recognized. To stop and restart a database, issue the following commands:

```
-STOP DATABASE(database-name)
-START DATABASE(database-name)
```

Device types: When the storage group is used at run time, an error can occur if the volumes in the storage group are of different device types, or if a volume is not available to z/OS for dynamic allocation of data sets.

When a storage group is used to extend a data set, all volumes in the storage group must be of the same device type as the volumes used when the data set was defined. Otherwise, an extend failure occurs if an attempt is made to extend the data set.

Number of volumes: There is no specific limit on the number of volumes that can be defined for a storage group. However, the maximum number of volumes that can be managed for a storage group is 133.

If the VOLUMES clause is specified, the maximum number of volumes is 59.

Verifying the existence of volumes and classes: When processing the VOLUMES, DATACLAS, MGMTCLAS, or STORCLAS clauses, DB2 does not check the existence of the volumes or classes or determine the types of devices that are identified or if SMS is active. Later, when the storage group allocates data sets, the list of volumes is passed in the specified order to Data Facilities (DFSMSdfp). See *DB2 Administration Guide* for more information about creating DB2 storage groups.

SMS data set management: You can have Storage Management Subsystem (SMS) manage the storage needed for the objects that the storage group supports. To do so, specify ADD VOLUMES(*) and REMOVE VOLUMES(current-vols) in the ALTER statement, where *current-vols* is the list of the volumes currently assigned to the storage group. SMS manages every data set created later for the storage group. SMS does not manage data sets created before the execution of the statement.

You can also specify ADD VOLUMES(volume-id) and REMOVE VOLUMES(*) to make the opposite change.

See *DB2 Administration Guide* for considerations for using SMS to manage data sets.

Examples

Example 1: Alter storage group DSN8G910. Add volumes DSNV04 and DSNV05.

```
ALTER STOGROUP DSN8G910
ADD VOLUMES (DSNV04,DSNV05);
```

Example 2: Alter storage group DSN8G910. Remove volumes DSNV04 and DSNV05.

```
ALTER STOGROUP DSN8G910
REMOVE VOLUMES (DSNV04,DSNV05);
```

ALTER TABLE

The ALTER TABLE statement changes the description of a table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The ALTER privilege on the table
- Ownership of the table
- DBADM authority for the database
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

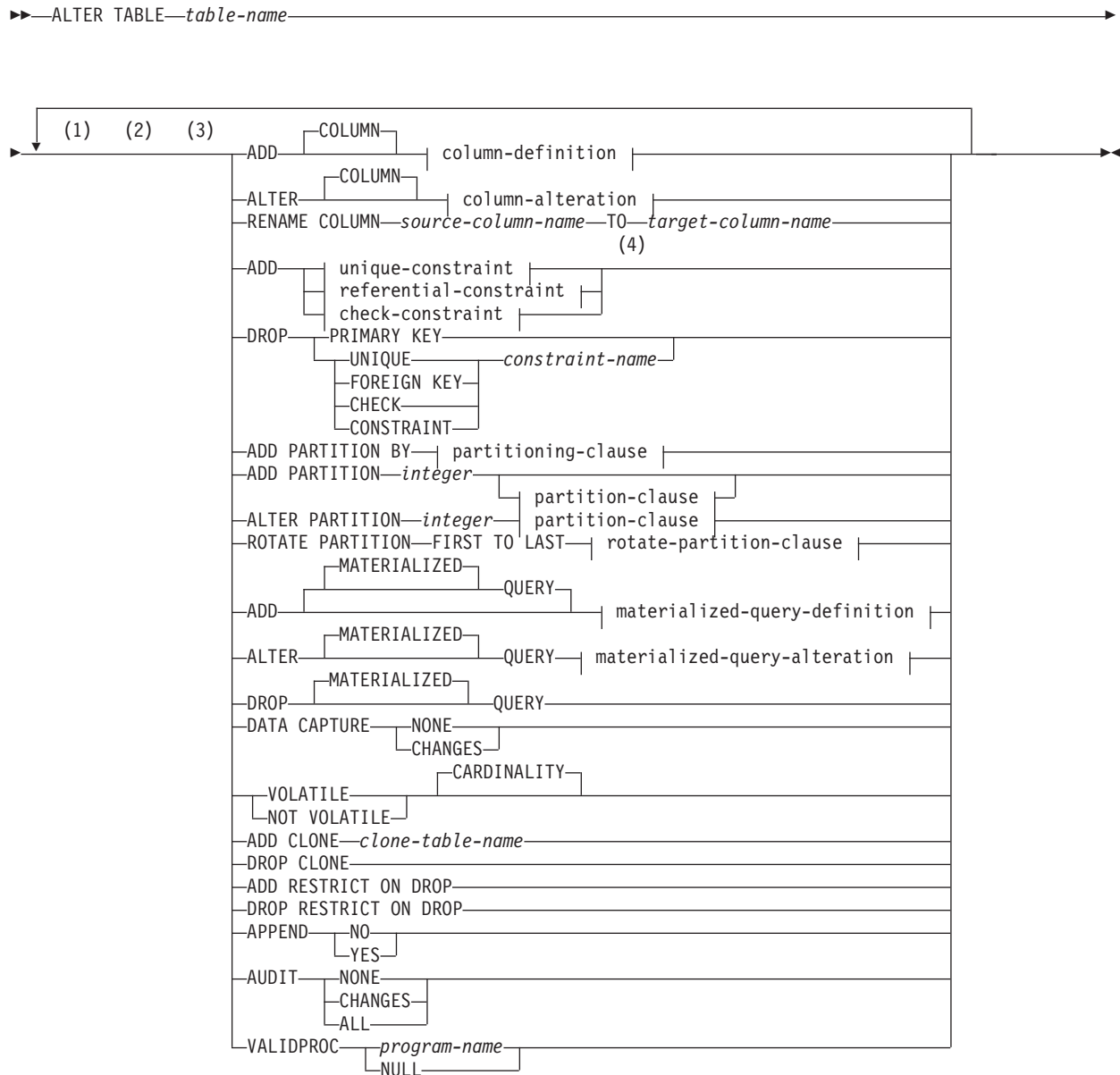
Additional privileges might be required in the following situations:

- FOREIGN KEY, ADD PRIMARY KEY, ADD UNIQUE, DROP PRIMARY KEY, DROP FOREIGN KEY, or DROP CONSTRAINT is specified.
- The data type of a column that is added to the table is a distinct type.
- A fullselect is specified.
- A column is defined as a security label column.
- A column is defined as ROWID GENERATED BY DEFAULT.

See the description of the appropriate clauses for the details about these privileges.

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and each role of the process.

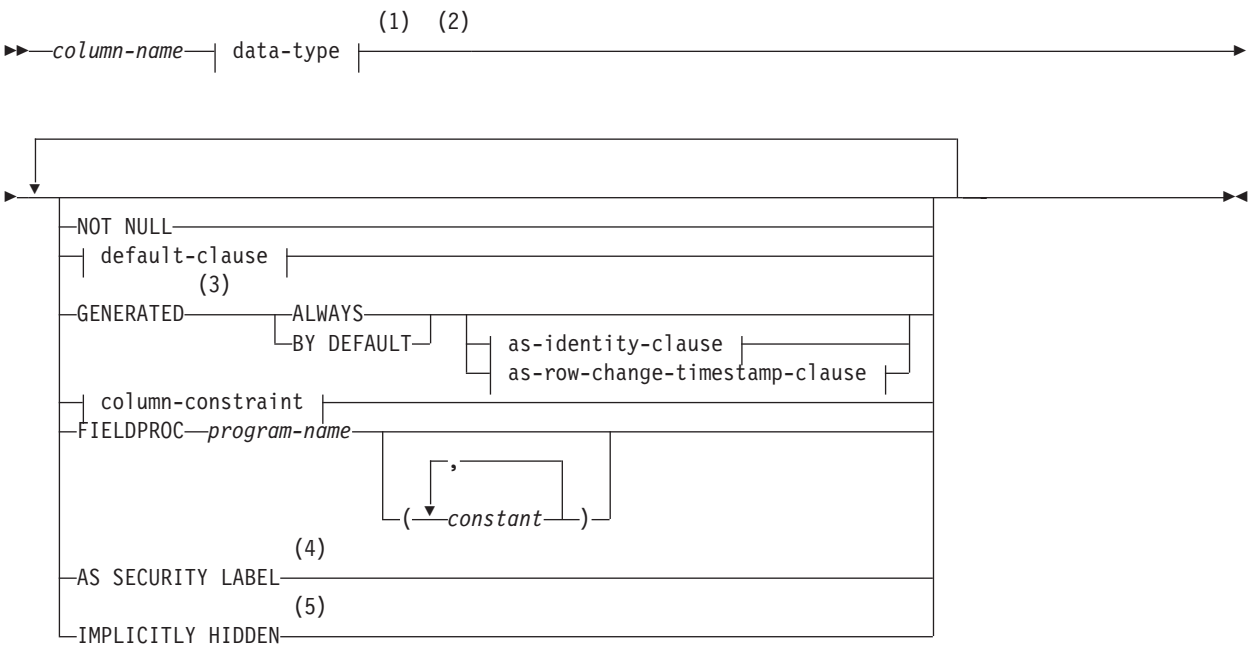
Syntax



Notes:

- 1 The same clause must not be specified more than one time, except for the ALTER COLUMN clause. If ALTER COLUMN SET DATA TYPE is specified, it must be specified first.
- 2 The ADD PARTITION, ALTER COLUMN, ALTER PARTITION, and ROTATE PARTITION FIRST TO LAST clauses are mutually exclusive with each other.
- 3 If ADD CLONE, DROP CLONE, or RENAME COLUMN is specified, no other clause is allowed on the ALTER TABLE statement.
- 4 The ADD keyword is optional for *referential-constraint* or *unique-constraint* if it is the first clause specified in the statement. Otherwise, ADD is required.

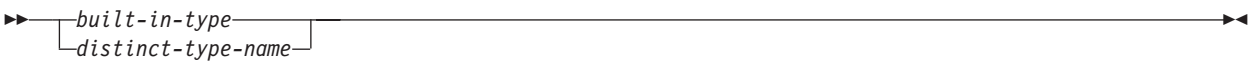
column-definition:



Notes:

- 1 *data-type* is optional if *as-row-change-timestamp-clause* is specified
- 2 The same clause must not be specified more than one time.
- 3 GENERATED must be specified if the column is to be an identity column. A column that has a ROWID data type (or a distinct type that is based on a ROWID data type) defaults to GENERATED ALWAYS.
- 4 AS SECURITY LABEL can be specified only for a CHAR(8) data type and requires that the NOT NULL and WITH DEFAULT clauses be specified.
- 5 IMPLICITLY HIDDEN must not be specified for a column defined as a ROWID, or a distinct type that is based on a ROWID.

data-type:



```

graph TD
    Root[SQL DATA TYPES] --> Small[SMALL DATA TYPES]
    Root --> Numeric[NUMERIC DATA TYPES]
    Root --> Real[REAL DATA TYPES]
    Root --> Decimal[DECIMAL DATA TYPES]
    Root --> Character[CHARACTER DATA TYPES]
    Root --> Graphic[GRAPHIC DATA TYPES]
    Root --> Binary[BINARY DATA TYPES]
    Root --> Date[DATE AND TIME DATA TYPES]
    Root --> Rowid[ROWID]
    Root --> Xml[XML]

    Small --> SmallInt[SMALLINT]
    Small --> Integer[INTEGER]
    Small --> Int[INT]
    Small --> BigInt[BIGINT]

    Numeric --> Decimal[DECIMAL]
    Numeric --> Dec[DEC]
    Numeric --> Numeric[NUMERIC]
    Numeric --> PrecisionScale["(precision, scale)"]

    Real --> Float[FLOAT]
    Real --> Real[REAL]
    Real --> Double[DOUBLE]
    Real --> Precision["(precision)"]

    Decimal --> DecFloat[DECFLOAT]
    Decimal --> PrecisionScale2["(precision, scale)"]

    Character --> Character[CHARACTER]
    Character --> Char[CHAR]
    Character --> Varchar[VARCHAR]
    Character --> Varying[VARYING]
    Character --> Clob[CLOB]
    Character --> LargeObject[LARGE OBJECT]
    Character --> Length["(length)"]
    Character --> Encoding["FOR SBCS MIXED BIT DATA"]

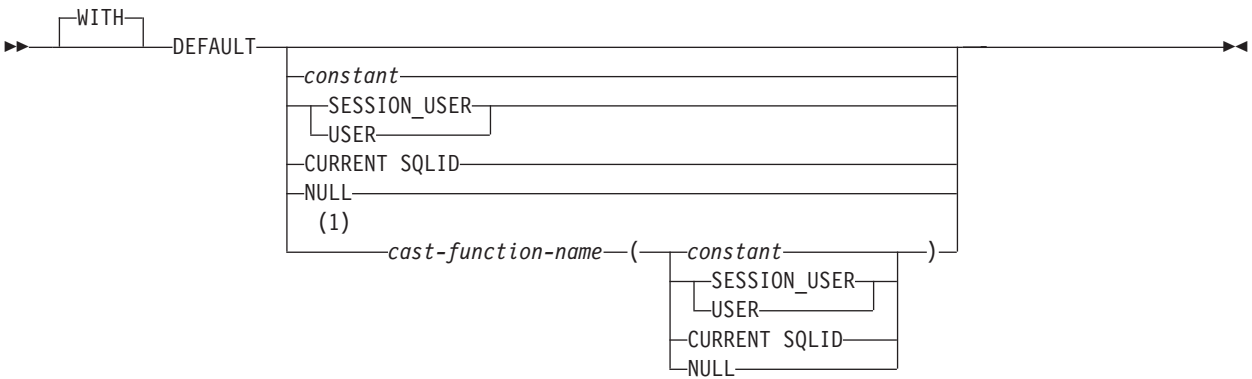
    Graphic --> Graphic[GRAPHIC]
    Graphic --> Vargraphic[VARGRAPHIC]
    Graphic --> Dbclob[DBCLOB]
    Graphic --> Length2["(length)"]
    Graphic --> Encoding2["FOR SBCS MIXED DATA"]

    Binary --> Binary[BINARY]
    Binary --> BinaryVarying[BINARY VARYING]
    Binary --> Varbinary[VARBINARY]
    Binary --> BinaryLargeObject[BINARY LARGE OBJECT]
    Binary --> Blob[BLOB]
    Binary --> Length3["(length)"]
    Binary --> Encoding3["FOR SBCS MIXED DATA"]

    Date --> Date[DATE]
    Date --> Time[TIME]
    Date --> Timestamp[TIMESTAMP]

    Rowid --> Rowid[ROWID]
    Xml --> Xml[XML]
  
```

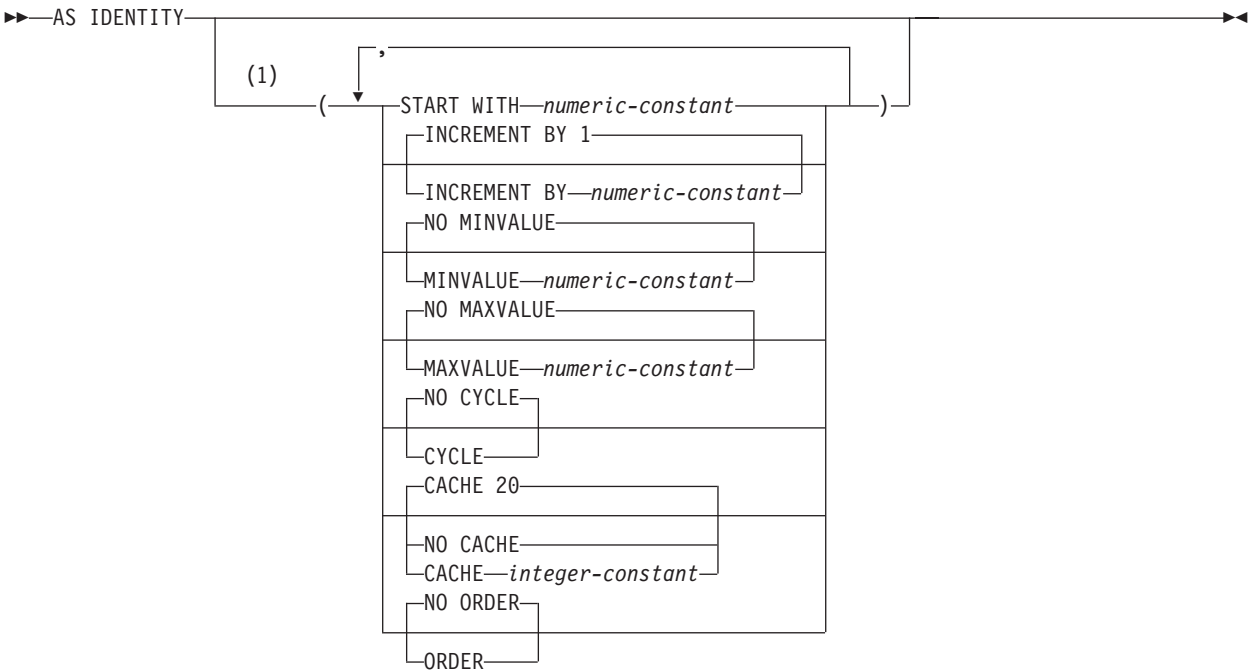
default-clause:



Notes:

- 1 The *cast-function-name* form of the DEFAULT value can only be used with a column that is defined as a distinct type.

as-identity-clause:



Notes:

- 1 Separator commas can be specified between attributes when an identity column is defined.

as-row-change-timestamp-clause:

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP

column-constraint

references-clause
check-constraint

column-alteration:

column-name SET DATA TYPE *altered-data-type* (1)
default-clause
DROP DEFAULT
SET GENERATED ALWAYS BY DEFAULT
SET GENERATED ALWAYS BY DEFAULT identity-alteration

Notes:

- 1 Each clause can be specified only one time. If DATA TYPE is specified, it must be specified first.

```

graph LR
    Root[DATA TYPE] --> Small[SMALL DATA TYPE]
    Root --> Precision[PRECISION DATA TYPE]
    Root --> Character[CHARACTER DATA TYPE]
    Root --> Graphic[GRAPHIC DATA TYPE]
    Root --> Binary[BINARY DATA TYPE]

    Small --> SmallInt[SMALLINT]
    Small --> Integer[INTEGER]
    Small --> Int[INT]
    Small --> BigInt[BIGINT]

    Precision --> Decimal[DECIMAL]
    Precision --> Dec[DEC]
    Precision --> Numeric[NUMERIC]
    Decimal --- D1["(5,0)"]
    Dec --- D2["(-integer, integer)"]
    Numeric --- D3["(53)"]
    Numeric --- D4["(integer)"]

    Character --> Float[FLOAT]
    Character --> Real[REAL]
    Character --> Double[DOUBLE]
    Float --- C1["(34)"]
    Float --- C2["(integer)"]
    Double --- C3["PRECISION"]

    Character --> DecFloat[DECFLOAT]
    DecFloat --- CF1["(34)"]
    DecFloat --- CF2["(16)"]

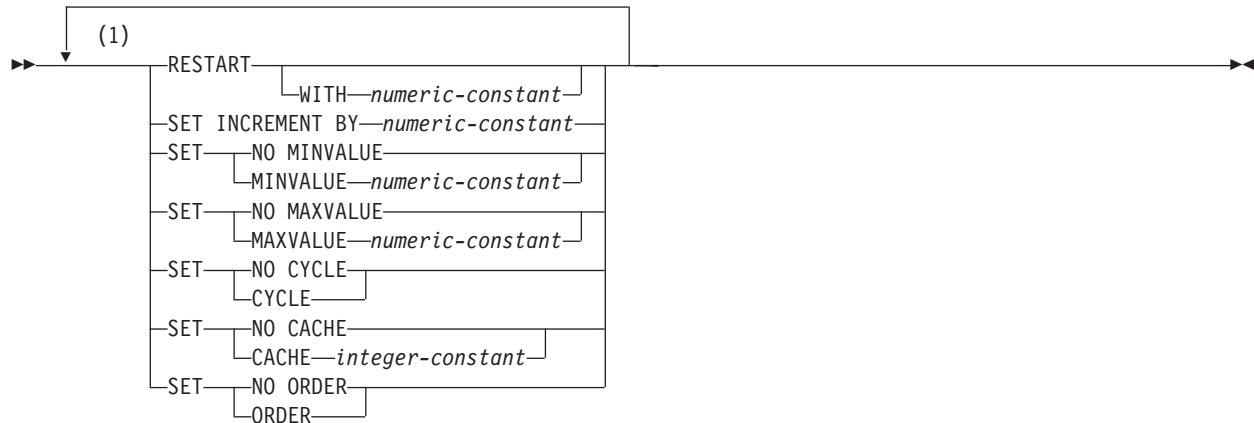
    Character --> Character[CHARACTER]
    Character --> Char[CHAR]
    Character --- CH1["(1)"]
    Character --- CH2["(integer)"]
    Character --> VaryingCharacter[VARYING CHARACTER]
    VaryingCharacter --- VC1["(integer)"]
    Character --> Varchar[VARCHAR]

    Character --> For[FOR]
    For --> SBCS[SBCS]
    For --> Mixed[MIXED]
    For --> Bit[BIT]
    For --> Data[DATA]

    Graphic --> Graphic[GRAPHIC]
    Graphic --- G1["(1)"]
    Graphic --- G2["(integer)"]
    Graphic --> Vargraphic[VARGRAPHIC]
    Vargraphic --- VG1["(integer)"]

    Binary --> Binary[BINARY]
    Binary --- B1["(1)"]
    Binary --- B2["(integer)"]
    Binary --> BinaryVarying[BINARY VARYING]
    BinaryVarying --- BV1["(integer)"]
    Binary --> Varbinary[VARBINARY]
  
```

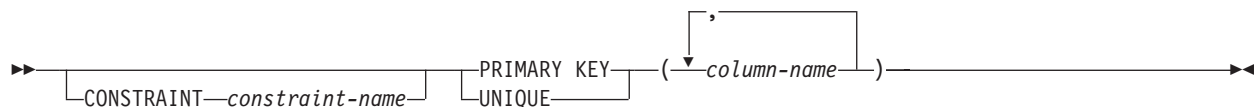
identity-alteration:



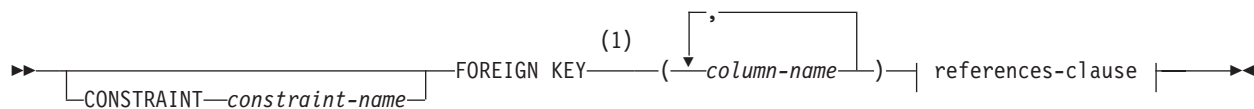
Notes:

- 1 At least one option must be specified and the same clause must not be specified more than once.

unique-constraint:



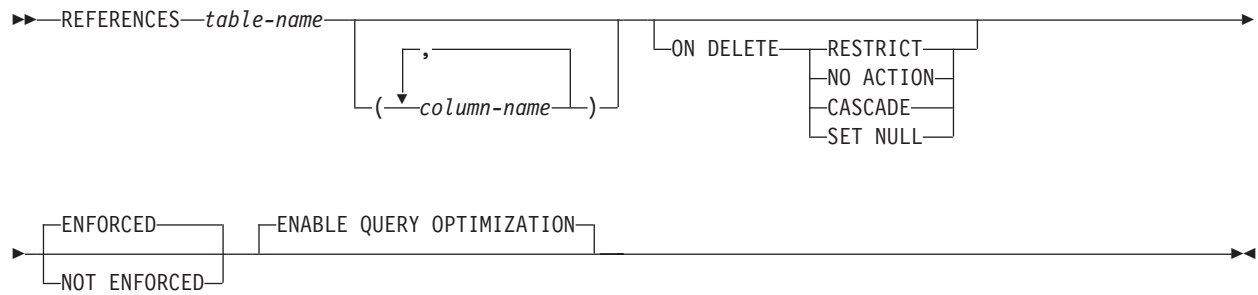
referential-constraint:



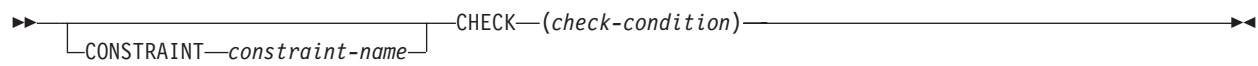
Notes:

- 1 For compatibility with prior releases, when the `CONSTRAINT` clause (shown above) is not specified, a *constraint-name* can be specified following `FOREIGN KEY`.

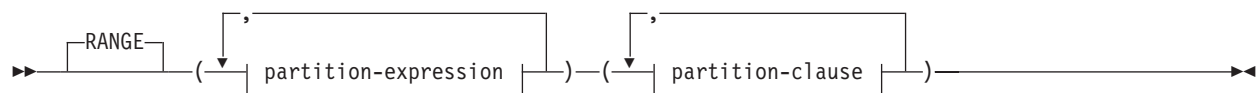
references-clause:



check-constraint:



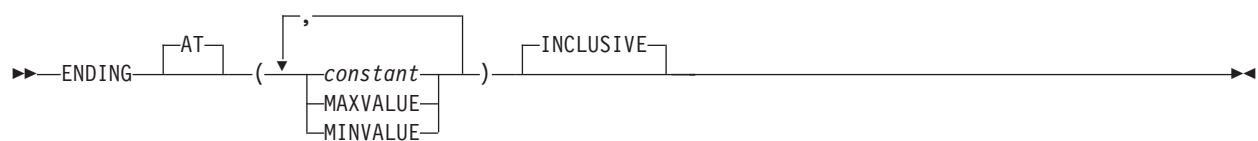
partitioning-clause:



partition-expression:



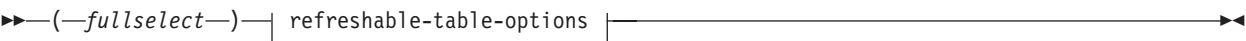
partition-clause:



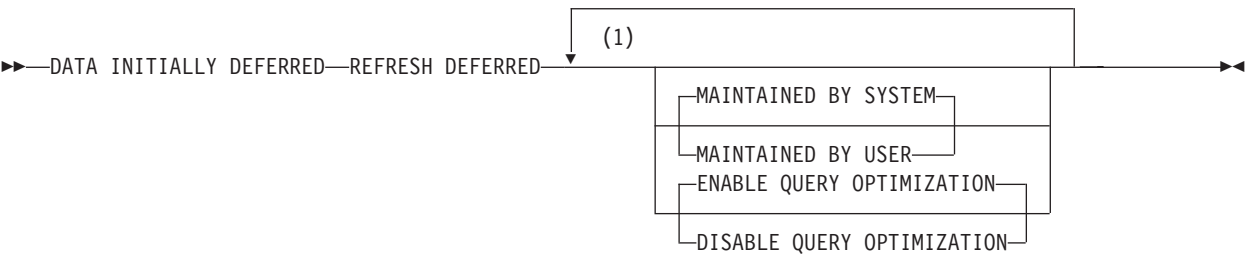
partition-rotation:



materialized-query-definition:



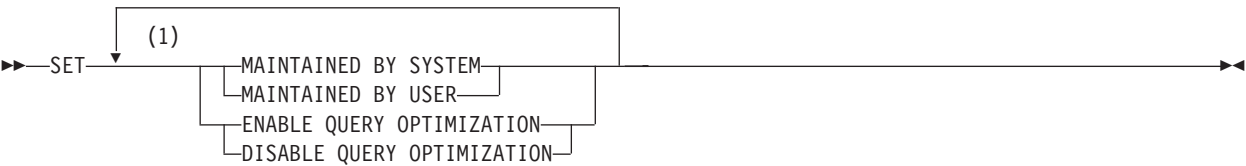
refreshable-table-options:



Notes:

- 1 The same clause must not be specified more than one time.

materialized-query-table-alteration:



Notes:

- 1 The same clause must not be specified more than one time.

Description

table-name

Identifies the table to be altered. The name must identify a table that exists at the current server. The name must not identify a declared temporary table,

view, or a table that was implicitly created for an XML column. If the name identifies a catalog table, DATA CAPTURE CHANGES is the only clause that can be specified.

If *table-name* identifies an auxiliary table, alterations are limited to the following clauses:

- APPEND

If *table-name* identifies a materialized query table, alterations are limited to the following clauses:

- AUDIT
- DATA CAPTURE
- ALTER MATERIALIZED QUERY
- DROP MATERIALIZED QUERY
- ADD RESTRICT ON DROP
- DROP RESTRICT ON DROP

ADD COLUMN:

ADD *column-definition*

Adds a column to the table. Except for a ROWID column and an identity column, all values of the column in existing rows are set to its default value. If the table has n columns, the ordinality of the new column is $n+1$. The value of n cannot be greater than 749. For a dependent table, n cannot be greater than 748.

The column cannot be added if the increase in the total byte count of the columns exceeds the maximum row size. The maximum row size for the table is eight less than the maximum record size as described in Maximum record size.

If you add a LOB column and the table does not already have a ROWID column, DB2 creates an implicitly hidden ROWID column. For details about adding a LOB column, such as the other objects that might be implicitly created or need to be explicitly created, see Creating a table with LOB columns. For more information about adding a ROWID column, see Adding a ROWID column.

For implicitly created LOB objects, the privilege set requires CREATETAB and CREATETS privileges on the database that contains the table (DSNDB04 if the database is implicitly created) and the USE privilege on the buffer pool and the storage group that is used by the auxiliary table and the LOB table space. The implicitly created objects are owned by the owner of the base table.

If you add an XML column, the privilege set requires the CREATETAB and CREATETS privileges on the database that contains the table (DSNDB04 if the database is implicitly created), INDEX on the base table for the first DOCID column that is added, and USE privilege on the buffer pool and the storage group that is used by the XML objects. These privileges are required for implicitly created XML objects. The implicitly created objects are owned by the owner of the base table.

When you add a column to a table, the table space is placed into advisory REORG-pending status.

You cannot add the following columns:

- A column to a table that has an edit procedure that is defined as WITH ROW ATTRIBUTES.

- A ROWID column to a table that already has an explicitly defined ROWID column
- An identity column to a table that has an identity column
- A security label column to a table that already has a security label column
- A row change timestamp column to a table that already has a row change timestamp column
- A LOB, ROWID, identity column, or row change timestamp column to a created temporary table
- A GRAPHIC, VARGRAPHIC, DBCLOB, or CHAR FOR MIXED DATA column, when the setting for installation option MIXED DATA is NO

column-name

Names of the column you want to add to the table. Do not use the name of an existing column of the table. Do not qualify *column-name*.

built-in-type

Specifies the data type of the column is one of the built-in data types. See built-in-type for information about the built-in data types that can be used when adding a column to a table.

distinct-type-name

Specifies the distinct type (user-defined data type) of the column. The length and scale of the column are respectively the length and scale of the source type of the distinct type. The privilege set must implicitly or explicitly include the USAGE privilege on the distinct type.

The encoding scheme of the distinct type must be the same as the encoding scheme of the table.

If the column is to be used in the definition of the foreign key of a referential constraint, the data type of the corresponding column of the parent key must have the same distinct type.

DEFAULT

The default value assigned to the column in the absence of a value specified in a data change statement, or LOAD. Do not specify DEFAULT for the following types of columns:

- A ROWID column (DB2 generates default values)
- An identity column (DB2 generates default values)
- An XML column
- A row change timestamp column

Do not specify a value after the DEFAULT keyword for a security label column. DB2 provides the default for a security label column.

If a value is not specified after the DEFAULT keyword, the default value depends on the data type of the column as indicated in the following table:

Data Type

Default Value

Numeric

0

Fixed-length character or graphic string

Blanks

Fixed-length binary string

Hexadecimal zeros

Varying-length string

A string of length 0

Date For existing rows, a date corresponding to 1 January 0001. For added rows, CURRENT DATE.

Time For existing rows, a time corresponding to 0 hours, 0 minutes, and 0 seconds. For added rows, CURRENT TIME.

Timestamp

For existing rows, a date corresponding to 1 January 0001, and a time corresponding to 0 hours, 0 minutes, 0 seconds, and 0 microseconds. For added rows, CURRENT TIMESTAMP.

In a given column definition:

- DEFAULT and FIELDPROC cannot both be specified.
- NOT NULL and DEFAULT NULL cannot both be specified.
- Omission of NOT NULL and DEFAULT for a column other than an identity column is an implicit specification of DEFAULT NULL. For an identity column, it is an implicit specification of NOT NULL, and DB2 generates default values.

A default value other than the one that is listed above can be specified in one of the following forms, except for a LOB column. The only form that can be specified for a LOB column is DEFAULT NULL. Unlike other varying-length strings, a LOB column can have the default value of only a zero-length string or null. Specify:

- WITH DEFAULT for a default value of an empty string
- DEFAULT NULL for a default value of null

constant

Specifies a constant as the default value for the column. The value of the constant must conform to the rules for assigning that value to the column.

A character or string constant must be short enough so that its UTF-8 representation requires no more than 1536 bytes. A hexadecimal graphic string (GX) constant cannot be specified.

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register at the time of an SQL data change statement or LOAD, as the default for the column. If SESSION_USER is specified, the data type of the column must be a character string with a length attribute greater than or equal to 8 characters when the value is expressed in CCSID 37. For existing rows, the value is that of the SESSION_USER special register at the time the ALTER TABLE statement is processed.

CURRENT SQLID

Specifies the value of the SQL authorization ID of the process at the time of an SQL data change statement or LOAD, as the default for the column. If CURRENT SQLID is specified, the data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the CURRENT SQLID special register. For existing rows, the value is the SQL authorization ID of the process at the time the ALTER TABLE statement is processed.

NULL

Specifies null as the default value for the column.

cast-function-name

The name of the cast function that matches the name of the distinct

type for the column. A cast function can be specified only if the data type of the column is a distinct type.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type.

constant

Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type.

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register at the time a row is inserted as the default for the column. The source type of the distinct type of the column must be a CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the SESSION_USER special register.

CURRENT SQLID

Specifies the value of the CURRENT SQLID special register at the time a row is inserted as the default for the column. The source type of the distinct type of the column must be a CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL

Specifies the NULL value as the argument.

GENERATED

Specifies that DB2 generates values for the column. GENERATED is applicable only to ROWID columns, identity columns, and row change timestamp columns. If the data type of the column is a ROWID (or a distinct type that is based on a ROWID), the default is GENERATED ALWAYS.

ALWAYS

Specifies that DB2 will generate a value for the column when a row is inserted into the table. ALWAYS is the recommended value unless you are using data propagation.

BY DEFAULT

Specifies that DB2 will generate a value for the column when a row is inserted unless a value was specified for the column on the data change statement.

If a user-supplied value is specified for a ROWID column, DB2 uses the value only if it is a valid row ID value that was previously generated by DB2 and the column has a unique, single-column index. Until this index is created on the ROWID column, the insert, and update operations and the LOAD utility cannot be used to add rows to the table. If the table space name is not specified on the CREATE TABLE statement, DB2 implicitly creates the necessary object to make the table complete, including the index. The name of this index is 'I' followed by the first ten characters of the column name followed by seven randomly generated characters. If the column name is less than ten characters, DB2 adds underscore characters to the end of the name until it has ten characters. The implicitly created index has the COPY NO attribute.

For an identity column, DB2 inserts a specified value but does not verify that it is a unique value for the column unless the identity column has a unique, single-column index.

If a user-supplied value is specified for an identity column, DB2 inserts the specified value but does not perform any special validation on that value beyond the normal validation that is performed for any column. DB2 does not check how the specified value affects the sequential properties that are defined for the identity column. To ensure the uniqueness of an identity column that is defined as GENERATED BY DEFAULT, define a unique index on the identity column.

BY DEFAULT is the recommended value only when you are using data propagation.

AS IDENTITY

Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero, or a distinct type based on one of these types). Separator commas between identity column attribute specifications are optional when the identity column is defined.

An identity column is implicitly NOT NULL. When adding an identity column to a table, you must also specify GENERATED ALWAYS or GENERATED BY DEFAULT.

Defining a column AS IDENTITY does not necessarily guarantee uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the identity column.

START WITH *numeric-constant*

Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending identity column and the MAXVALUE for a descending identity column. This value is not necessarily the value that would be cycled to after reaching the maximum or minimum value for the identity column. The START WITH clause can be used to start the generation of values outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value (including 0) that does not exceed the value of a large integer constant and could be assigned to the column without any non-zero digits existing to the right of the decimal point. The default is 1.

If the value is positive or zero, the sequence of values for the identity column ascends. If the value is negative, the sequence of values descends.

MINVALUE or NO MINVALUE

Specifies the minimum value at which a descending identity

column either cycles or stops generating values or an ascending identity column cycles to after reaching the maximum value.

NO MINVALUE

Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MINVALUE becomes one of the following:

- For an ascending identity column, the value is the START WITH value or 1 if START WITH was not specified.
- For a descending identity column, the value is the minimum value of the data type of the column.

MINVALUE *numeric-constant*

Specifies the numeric constant that is the minimum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

MAXVALUE or NO MAXVALUE

Specifies the maximum value at which a ascending identity column either cycles or stops generating values or a descending identity column cycles to after reaching the minimum value.

NO MAXVALUE

Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending identity column, the value is the maximum value of the data type of the column.
- For a descending identity column, the value is the START WITH value or -1 if START WITH is not specified.

MAXVALUE *numeric-constant*

Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

CYCLE or NO CYCLE

Specifies whether this identity column should continue to generate values after reaching either its maximum or minimum value.

NO CYCLE

Specifies that values will not be generated for the identity column after the maximum or minimum value has been reached. This is the default.

CYCLE

Specifies that values continue to be generated for this column after the maximum or minimum value has been reached. If this option is used, after an ascending identity column reaches the maximum value, it generates its minimum value. After a descending identity column reaches its minimum value, it

generates its maximum value. The maximum and minimum values for the identity column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by DB2 for an identity column. However, if a unique index exists on the identity column and a non-unique value is generated for it, an error occurs.

CACHE or NO CACHE

Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table. The default is CACHE 20.

NO CACHE

Specifies that values for the identity column are not preallocated and stored in the cache, ensuring that values will not be lost in the case of a system failure. In this case, every request for a new value for the identity column results in synchronous I/O.

CACHE *integer-constant*

Specifies the maximum number of values of the identity column sequence that DB2 can preallocate and keep in memory.

During a system failure, all cached identity column values that are yet to be assigned might be lost and will not be used. Therefore, the value that is specified for CACHE also represents the maximum number of values for the identity column that could be lost during a system failure.

The minimum value is 2.

In a data sharing environment, you can use the CACHE and NO ORDER options to allow multiple DB2 members to cache sequence values simultaneously.

ORDER or NO ORDER

Specifies whether the identity column values must be generated in order of request. The default is NO ORDER.

NO ORDER

Specifies that the values do not need to be generated in order of request.

ORDER

Specifies that the values are generated in order of request. Specifying ORDER might disable the caching of values. ORDER applies only to a single-application process.

In a data sharing environment, if the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously, and the requests for identity values from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the identity column, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested a value first, then DB2B requested, and then DB2A again requested.

Therefore, to guarantee that identity values are generated in strict numeric order among multiple DB2 members using the same identity column, specify the ORDER option.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP

Specifies that the column is a timestamp and the values will be generated by DB2. DB2 generates a value for the column for each row as a row is inserted, and for any row for which any column is updated. The value that is generated for a row change timestamp column is a timestamp that corresponds to the time of the insert or update of the row. If multiple rows are inserted or updated with a single statement, the value of the row change timestamp column might be different for each row.

If *data-type* is specified, it must be **TIMESTAMP**. You must specify **NOT NULL** with a row change timestamp column.

NOT NULL

Prevents the column from containing null values. If **NOT NULL** is specified, the **DEFAULT** clause must be used to specify a nonnull default value for the column unless the column has a row ID data type or is an identity column. For a ROWID column, **NOT NULL** must be specified, and **DEFAULT** must not be specified. For an identity column, although **NOT NULL** can be specified, **DEFAULT** must not be specified.

IMPLICITLY HIDDEN

Specifies that the column is not visible in the results of SQL statements unless you refer explicitly to the column by name. For example, assume that table T1 includes a column that is defined with the **IMPLICITLY HIDDEN** clause. The result of **SELECT * FROM T1** would not include the implicitly hidden column. However, the result of a **SELECT** statement that explicitly refers to the name of the implicitly hidden column would include that column in the result table.

IMPLICITLY HIDDEN must not be specified for a column that is defined as a ROWID, or a distinct type that is based on a ROWID.

references-clause

The *references-clause* of a *column-definition* provides a shorthand method of defining a foreign key composed of a single column. Thus, if *references-clause* is specified in the definition of column C, the effect is the same as if that *references-clause* were specified as part of a **FOREIGN KEY** clause in which C is the only identified column.

Do not specify *references-clause* in the definition of a LOB, ROWID, XML, DECFLOAT column, or a row change timestamp column, or a security label column because these types of columns cannot be a foreign key.

check-constraint

The *check-constraint* of a *column-definition* has the same effect as specifying a check constraint in a separate **ADD check-constraint** clause. For conformance with the SQL standard, a check constraint specified in the definition of column C should not reference any columns other than C.

Do not specify a check constraint in the definition of a LOB, ROWID, DECFLOAT, XML, or security label column.

FIELDPROC *program-name*

Designates *program-name* as the field procedure exit routine for the column. Writing a field procedure exit routine is described in *DB2 Administration Guide*. A field procedure can be specified only for a column with a length

attribute that is not greater than 255 bytes. FIELDPROC can only be specified for columns that are a built-in character string or graphic string data type that is not a LOB. The column must not be one of the following:

- a security label column
- a row change timestamp column

The field procedure encodes and decodes column values. Before a value is inserted in the column, it is passed to the field procedure for encoding. Before a value from the column is used by a program, it is passed to the field procedure for decoding. A field procedure could be used, for example, to alter the sorting sequence of values entered in the column.

The field procedure is also invoked during the processing of the ALTER TABLE statement. When so invoked, the procedure provides DB2 with the column's *field description*. The field description defines the data characteristics of the encoded values. By contrast, the information you supply for the column in the ALTER TABLE statement defines the data characteristics of the decoded values.

constant

Is a parameter that is passed to the field procedure when it is invoked. A parameter list is optional. The *n*th parameter specified in the FIELDPROC clause on ALTER TABLE corresponds to the *n*th parameter of the specified field procedure. The maximum length of the parameter list is 255 bytes, including commas but excluding insignificant blanks and the delimiting parentheses.

If you omit FIELDPROC, the column has no field procedure.

AS SECURITY LABEL

Specifies that the table is defined with multilevel security with row level granularity and specifies that the column will contain the security label values. A table can have only one security label column. To define a table with a security label column, the primary authorization ID of the statement must have a valid security label, and the RACF SECLABEL class must be active. In addition, the following conditions are also required:

- The data type of the column must be CHAR(8).
- The subtype of the column must be SBCS.
- The column does not have any field procedures, check constraints, or referential constraints.
- The column must be defined as NOT NULL and WITH DEFAULT clauses.
- The WITH DEFAULT clause must not be specified with a default value (DB2 provides the default value).
- The table does not have an edit procedure that is defined as WITH ROW ATTRIBUTES.
- The table is not the source table for a materialized query table.

For existing rows in the table, the value of the security label column defaults to the security label of the user at the time the ALTER statement is executed.

ALTER COLUMN:

ALTER COLUMN *column-alteration*

Alters the definition of an existing column, including the attributes of an existing identity column. Only the attributes specified are altered. Other attributes remain unchanged. Only future values of the column are affected by the changes made with an ALTER TABLE ALTER COLUMN statement.

The table being altered must not be in an incomplete state because of a missing unique index on a unique constraint (primary or unique key). An ALTER TABLE ALTER COLUMN statement might not be processed in the same unit of work as a data change statement. A column cannot be altered if any of the following conditions are true:

- The table has an edit procedure that is defined as WITH ROW ATTRIBUTES or a validation exit procedure
- The table is used in a materialized query table definition
- The table is a materialized query table
- There is an extended index that depends on that column
- The column is referenced in a field procedure
- The column is referenced in a referential constraint
- The column is a LOB column
- The column is defined as a security label column
- The column is defined as a row change timestamp column

You can modify all the attributes of an existing identity column, except for the data type of the column. To change the data type of an identity column, drop the table containing the column and recreate it. When the attributes of an identity column are altered, the column of the specified *column-name* must exist in the specified table and must have been defined with the IDENTITY attribute.

column-name

Identifies the column to be altered. The name must not be qualified and must identify an existing column in the table being altered when the ALTER statement is processed. The name must not identify a column that is being added in the same ALTER TABLE statement.

A column can only be referenced in one ALTER COLUMN clause in a single ALTER TABLE statement. However, that same column can be referenced multiple times for adding or dropping constraints in the same ALTER TABLE statement.

SET DATA TYPE (*altered-data-type*)

Specifies the new data type of the column to be altered. For a character column, you can also use the clause to change the definition of the subtype that is stored in the DB2 catalog and OBD.

The column cannot be an identity column. The new data type must be compatible with the existing data type of the column. The existing data type of the column cannot be a ROWID, LOB, date, time, timestamp, or distinct type. If the column is a partitioning column, and the existing data type is CHAR or VARCHAR FOR BIT DATA, the new data type cannot be VARBINARY or BINARY. If the column is CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, or BINARY, the new data type cannot be VARBINARY if the column is part of an index and is defined with the DESC attribute. For more information on the compatibility of data types, see “Assignment and comparison” on page 102.

If any numeric data type is being converted to DECFLOAT, the ALTER statement will fail if there is a partitioning key, check constraints, index, or a unique constraint on the column.

If the data type is a character or graphic string, the new length attribute must be at least as large as the existing length attribute of the column. If the data type is a numeric data type, the specified precision and scale must

be at least as large as the existing precision and scale. If a decimal fraction is being converted to floating point, the ALTER statement will fail if there is a unique index or a unique constraint on the column.

If the specified column has a default value, the existing default value must represent a value that could be assigned to a column with the new data type in accordance with the rules for assignment. The default value is updated to reflect the new data type.

If the column is specified in a unique constraint (unique key or primary key) or unique index, the new column length must not exceed the limit on an index size. For PADDED indexes, the sum of the length attributes of the columns must not be greater than $2000-n$, where n is the number of columns that can contain null values. For NOT PADDED indexes, the sum of the length attributes of the columns must not be greater than $2000-n-2m$, where n is the number of nullable columns and m is the number of varying length columns.

The total byte count of columns after the alteration must not exceed the maximum row size. If the column is in the partitioning key, the new partitioning key cannot exceed $255-n$.

Table 85 shows the numeric data type alterations that are supported for SET DATA TYPE:

Table 85. Supported numeric data type alterations for SET DATA TYPE

From/To	SMALLINT	INTEGER	BIGINT	DECIMAL (q,t)	REAL	DOUBLE	DECFLOAT (16)	DECFLOAT (34)
SMALLINT	Y	Y	Y	(q-t)>4	Y	Y	Y	Y
INTEGER	N	Y	Y	(q-t)>9	N	Y	Y	Y
BIGINT	N	N	Y	(q-t)>18	N	N	N	Y
DECIMAL (p,s)	s=0 p<5	s=0 p<10	s=0 p<=19	q>=p (q-t)>=(p-s)	p<7	p<16	p<17	Y
DECFLOAT (16)	N	N	N	N	N	N	Y	Y
DECFLOAT (34)	N	N	N	N	N	N	N	Y
FLOAT (1-21)	N	N	N	N	Y	Y	Y	Y
FLOAT (22-53)	N	N	N	N	N	Y	Y	Y

When a SMALLINT, INTEGER, or DECIMAL column is altered to a BIGINT data type, and there is an index defined on that column, the index will be put in RBDP status.

In releases of DB2 prior to Version 9.1, use of the DECIMAL(19,0) data type for applications that work with BIGINT data was encouraged. For performance reasons, the DECIMAL(19,0) columns should be altered to BIGINT. Note that altering from DECIMAL(19,0) to BIGINT is provided only for DECIMAL(19,0) columns that are used for applications that work with BIGINT (thus, the data in those columns is within the range of the BIGINT).

When altering from DECIMAL(19,0) to BIGINT you should ensure that all values in the DECIMAL(19,0) column are within the range of BIGINT

before the alter. The following query or a similar query can be run to determine which rows (if any) contain values that are outside of the range of BIGINT:

```
SELECT * FROM table_name
WHERE dec19_0_column > 9223372036854775807
OR dec19_0_column < -9223372036854775808;
```

When a partitioning key column with a numeric data type is altered to a larger numeric data type, and the limit key value for the original numeric data type of the column is X'FF', the limit key value for the new numeric data type of the column is left-padded with X'FF'. For example, if a column is converted from SMALLINT to INTEGER, and a limit key value for the SMALLINT column is 32767 (which is 2 bytes of X'FF'), the limit key for the INTEGER column is 2147483647 (which is 4 bytes of X'FF').

When a partitioning key column with a character data type is altered to a longer character data type, and the limit key value for the original character data type of the column (excluding the first NULL byte if the column is nullable) is neither all X'FF' nor all X'00', the limit key value for the new character data type of the column is right-padded with blank(s) of the encoding scheme of the table. For example, if a column is converted from CHAR(1) to VARCHAR(2), and a limit key value for the CHAR(1) column is 'A' (which is X'C1'), the limit key for the VARCHAR(2) column is 'A ' (which is X'C140' when the encoding scheme of the table is EBCDIC, or is X'C120' when the encoding scheme of the table is UNICODE or ASCII).

When a partitioning key column with a character data type is altered to a longer character data type, and the limit key value for the original character data type of the column (excluding the first NULL byte if the column is nullable) is all X'FF', the limit key value for the new character data type of the column is right-padded with X'FF' and the table space that contains the table being altered is left in REORG-pending (REORP) status.

When a partitioning key column with a character data type is altered to a longer character data type, and the limit key value for the original character data type of the column (excluding the first NULL byte if the column is nullable) is all X'00', the limit key value for the new character data type of the column is right-padded with X'00' and the table space that contains the table being altered is left in REORG-pending (REORP) status.

Table 86 shows the character data type alterations that are supported for SET DATA TYPE:

Table 86. Supported character data type alterations for SET DATA TYPE (x >=0).

From/To	CHARACTER (n+x)	VARCHAR (n+x)	LONG VARCHAR	GRAPHIC (n+x)	VARGRAPHIC (n+x)	LONG VARGRAPHIC
CHARACTER(n)	Y	Y	N	N	N	N
VARCHAR(n)	Y	Y	N	N	N	N
LONG VARCHAR	N	Y	N	N	N	N
GRAPHIC(n)	N	N	N	Y	Y	N
VARGRAPHIC(n)	N	N	N	Y	Y	N
LONG VARGRAPHIC	N	N	N	N	Y	N

When columns are converted from CHAR to VARCHAR, normal assignment rules apply, which means that trailing blanks are kept instead of being stripped out. If you want varying length character strings without trailing blanks, use the STRIP function for data in the column after changing the data type to VARCHAR.

When a CHAR FOR BIT DATA column is converted to a BINARY data type, the following applies:

- The existing space characters in the table will not be changed to hexadecimal zeros (X'00')
- If the new length attribute is greater than current length attribute of the column, the values in the table are padded with hexadecimal zeros (X'00')

When a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA column is converted to a BINARY or VARBINARY data type, the existing default value will be cast as a binary string. The resulting binary string will be at least twice the original size. The alter will fail if the resulting binary string length exceeds 1536 UTF-8 bytes.

When a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA column is converted to a BINARY or VARBINARY data type, and there is an index defined on that column, the index will be put in RBDP.

Table 87. Supported binary data type alterations for SET DATA TYPE ($x \geq 0$)

From/To	BINARY(n+x)	VARBINARY(n+x)
CHAR(n) FOR BIT DATA	Y	Y
VARCHAR(n) FOR BIT DATA	Y	Y
BINARY(n)	Y	Y
VARBINARY(n)	Y ¹	Y

Note: ALTER from VARBINARY to BINARY is not allowed when the column is part of a unique index.

The table being altered must not be defined with an edit procedure that is defined as WITH ROW ATTRIBUTES or a valid procedure. There must not be a materialized query table defined on this table, and this table must not be defined as a materialized query table.

If the alteration results in the generation of a new table version, the table space that contains the table that is being changed is left in an advisory REORG-pending (AREO) status. If the column that is being changed is part of an index, an exception state might be set for the index as shown in Table 88:

Table 88. Informational settings for ALTER COLUMN when the column is in an index

Alteration type	Exception state for index	Plans and packages invalidation
VARCHAR to CHAR	AREO*	Yes
VARGRAPHIC to GRAPHIC	AREO*	Yes
CHAR to VARCHAR	AREO*	Yes
GRAPHIC to VARGRAPHIC	AREO*	Yes
VARCHAR to VARCHAR	AREO* (for padded only)	No
VARGRAPHIC to VARGRAPHIC	AREO* (for padded only)	No
CHAR to CHAR	AREO*	Yes

Table 88. Informational settings for ALTER COLUMN when the column is in an index (continued)

Alteration type	Exception state for index	Plans and packages invalidation
GRAPHIC to GRAPHIC	AREO*	Yes
DECIMAL to DECIMAL	RBDP	Yes

For information on resetting informational or restrictive exception states, see *DB2 Utility Guide and Reference*.

FOR subtype DATA

Alters the *subtype* of a character column. This clause does not change the data. The clause only updates the definition of the subtype as it is stored in the DB2 catalog and the OBD. The length and data type that are specified must match the existing length and data type of the column.

Only character strings are valid when subtype is BIT.

For more information on the subtype values (SBCS, MIXED, and BIT), see the subtype information under built-in-type.

SET default-clause

Specifies the new default value of the column to be altered. The new default value must conform to the current rules for assigning that value to the column. Existing rows will retain their current value. The new default value will only be reflected in the rows that are inserted after the alter. Sections that are dependent on the table that is being altered will be invalidated.

The table must not be referenced by an view.

If the column is specified in a unique constraint (unique key or primary key) or unique index, the default value might be altered to the same value as an existing row of that column. However, subsequent data change operations will fail in the absence of a value specified for that column on the insert operation.

DROP DEFAULT

Drops the current default value of the column. For columns that are not nullable, the specified column must be defined with a default value. For columns that are nullable, the specified column cannot have a null default value. For columns that are nullable, the new default value is the null value.

The table that contains the specified column must not be referenced in a view.

Follow these steps to remove the default value for a column that was defined using ALTER TABLE with the ADD COLUMN clause:

1. Run the REORG utility on the table space that contains the table
2. Issue the ALTER TABLE statement that specifies the DROP DEFAULT clause

If the REORG is not processed, an error is returned for the ALTER TABLE statement.

SET GENERATED ALWAYS or BY DEFAULT

Specifies when DB2 is to generate values for the column. GENERATED BY DEFAULT specifies that a value is only to be generated when a value is not provided. or the DEFAULT keyword is used in an assignment to the

column. GENERATE ALWAYS specifies that DB2 is to always generate a value for the column. This clause can only be specified for an identity column.

RESTART

Specifies the next value for the identity column. If *numeric-constant* is not specified, the sequence is restarted at the value that is specified implicitly or explicitly as the starting value when the identity column was originally created.

WITH *numeric-constant*

Specifies that, when it is time to generate the next value for this identity column, *numeric-constant* will be used as the next value for the column. This value can be any positive or negative value (including 0) that could be assigned to this column without nonzero digits existing to the right of the decimal point.

If RESTART is not specified, the sequence is not restarted. Instead, it resumes with the current values that are in effect for all the options after the ALTER statement is issued.

After an identity column is restarted or changed to allow cycling, sequence numbers might be duplicates of values generated previously.

SET INCREMENT BY *numeric-constant*

For a definition, see the description of INCREMENT BY *numeric-constant* for defining an identity column.

SET MINVALUE or NO MINVALUE

For a definition, see the description of MINVALUE or NO MINVALUE for defining an identity column.

SET MAXVALUE or NO MAXVALUE

For a definition, see the description of MAXVALUE or NO MAXVALUE for defining an identity column.

SET CYCLE or NO CYCLE

For a definition, see the description of CYCLE or NO CYCLE for defining an identity column.

SET CACHE or NO CACHE

For a definition, see the description of CACHE or NO CACHE for defining an identity column.

SET ORDER or NO ORDER

For a definition, see the description of ORDER or NO ORDER for defining an identity column.

RENAME COLUMN:

RENAME COLUMN *source-column-name* **TO** *target-column-name*

Renames the specified column. The names must not be qualified.

source-column-name

Identifies the column that is to be renamed. The name must identify an existing column of the table.

target-column-name

Specifies the new name for the column. The name must not identify a column that already exists in the table.

You cannot rename a column if any of the following conditions apply:

- The column is referenced in a view
- The column is referenced in the expression of an index definition
- The column has a check constraint defined
- The column has a field procedure defined
- The table has a trigger
- The table is a materialized query table or is referenced by a materialized query table
- The table has a valid procedure, or an edit procedure that is defined as WITH ROW ATTRIBUTES
- The table is a DB2 catalog table

ADD unique-constraint:

CONSTRAINT *constraint-name*

Names the primary key or unique key constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table. If the table space is implicitly created, the enforcing primary key and unique key indexes are also implicitly created.

PRIMARY KEY(*column-name*,...)

Defines a primary key composed of the identified columns. Each column name must be an unqualified name that identifies a column of the table except a LOB, ROWID, DECFLOAT (including a distinct type that is based on a LOB, ROWID, or DECFLOAT data type), XML column, or a row change timestamp column, and the same column must not be identified more than one time. The number of identified columns must not exceed 64. In addition, the sum of the length attributes of the columns must not be greater than $2000 - 2m$, where m is the number of varying-length columns in the key. The table must not have a primary key and the identified columns must be defined as NOT NULL.

The set of columns in the primary key cannot be the same as the set of columns of another unique key.

The table must have a unique index with a unique key that is identical to the primary key. The keys are identical only if they have the same number of columns and the n th column name of one is the same as the n th column name of the other. If the table is in a table space that is implicitly created, and no unique index is defined on the identified columns, DB2 will automatically create a primary index. The privilege set must include the INDEX privilege on the table and the USE privilege on the buffer pool and the storage group. The implicitly created primary key index is owned by the owner of the base table.

The identified columns are defined as the primary key of the table. The description of the index is changed to indicate that it is a primary index. If the table has more than one unique index with a key that is identical to the primary key, the selection of the primary index is arbitrary.

UNIQUE(*column-name*,...)

Defines a unique key composed of the identified columns with the specified *constraint-name*. If a *constraint-name* is not specified, a name is generated. Each column name must be an unqualified name that identifies a column of the table except a LOB, ROWID, XML, or DECFLOAT column (including a distinct type that is based on a LOB, ROWID, or DECFLOAT data type), and the same column must not be identified more than one time. Each identified column must be defined as NOT NULL. The number of identified columns must not

exceed 64. In addition, the sum of the length attributes of the columns must not be greater than $2000 - n$ for padded indexes and $2000 - n - 2m$ for nonpadded indexes, where n is the number of columns that can contain null values and m is the number of varying-length columns in the key.

The set of columns in the unique key cannot be the same as the set of columns of the primary key or another unique key. A unique key is a duplicate if it is the same as the primary key or a previously defined unique key. The specification of a duplicate unique key is ignored with a warning.

The table must have a unique index with a key that is identical to the unique key. The keys are identical only if they have the same number of columns and the n th column name of one is the same as the n th column name of the other. If the table is in a table space that is implicitly created, and no unique index is defined on the identified columns, DB2 will automatically create a unique index to enforce the unique key constraint. The privilege set must include the INDEX privilege on the table and the USE privilege on the buffer pool and the storage group. The implicitly created unique key index is owned by the owner of the base table.

The identified columns are defined as a unique key of the table. The description of the index is changed to indicate that it is enforcing a unique key constraint. If the table has more than one unique index with a key that is identical to the unique key, the selection of the enforcing index is arbitrary.

ADD referential-constraint:

CONSTRAINT *constraint-name*

Names the referential constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

FOREIGN KEY (*column-name*,...) *references-clause*

Specifies a referential constraint with the specified *constraint-name*.

Let T1 denote the object table of the ALTER TABLE statement.

The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T1 except a LOB, ROWID, XML, DECFLOAT column, security label column, or a row change timestamp column, and the same column must not be identified more than one time. The number of identified columns must not exceed 64 and the sum of their length attributes must not exceed 255 minus the number of columns that allow null values. The referential constraint is a duplicate if the FOREIGN KEY and the parent table are the same as the FOREIGN KEY and parent table of an existing referential constraint on T1. The specification of a duplicate referential constraint is ignored with a warning.

REFERENCES *table-name* (*column-name*,...)

The table name specified after REFERENCES must identify a table that exists at the current server, but it must not identify a catalog table or a declared global temporary table. Let T2 denote the identified parent table and let T1 denote the table that is being changed (T1 and T2 can be the same table).

T2 must have a unique index and the privilege set on T2 must include the ALTER or REFERENCES privilege on the parent table, or the REFERENCES privilege on the columns of the nominated parent key.

The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a

column of T2. The identified column cannot be a LOB, ROWID, XML, DECFLOAT, security label, or row change timestamp column. The same column must not be identified more than one time.

The list of column names in the parent key must be identical to the list of column names in a primary key or unique key in the parent table T2. The column names must be specified in the *same order* as in the primary key or unique key. If any of the referenced columns in T2 has a non-numeric data type, T2 and T1 must use the same encoding scheme.

If a list of column names is not specified, then T2 must have a primary key. Omission of a list of column names is an implicit specification of the columns of the primary key for T2.

The specified foreign key must have the same number of columns as the parent key of T2 and, except for their names, default values, null attributes and check constraints, the description of the *n*th column of the foreign key must be identical to the description of the *n*th column of the nominated parent key. If the foreign key includes a column defined as a distinct type, the corresponding column of the nominated parent key must be the same distinct type. If a column of the foreign key has a field procedure, the corresponding column of the nominated parent key must have the same field procedure and an identical field description. A *field description* is a description of the encoded value as it is stored in the database for a column that has been defined to have an associated field procedure.

The table space that contains T1 must be available to DB2. If T1 is populated, its table space is placed in a check pending status. A table in a segmented table space is populated if the table is not empty. A table in a table space that is not segmented is considered populated if the table space has ever contained any records.

The referential constraint specified by the FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent. A description of the referential constraint is recorded in the catalog.

ON DELETE

The delete rule of the relationship is determined by the ON DELETE clause. For more on the concepts used here, see “Referential constraints” on page 20.

If T1 and T2 are the same table, CASCADE or NO ACTION must be specified. SET NULL must not be specified unless some column of the foreign key allows null values. Also, SET NULL must not be specified if any nullable column of the foreign key is a column of the key of a partitioning index. The default value for the rule depends on the value of the CURRENT RULES special register when the ALTER TABLE statement is processed. If the value of the register is 'DB2', the delete rule defaults to RESTRICT; if the value is 'SQL', the delete rule defaults to NO ACTION.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

A cycle involving two or more tables must not cause a table to be delete-connected to itself. Thus, if the relationship would form a cycle:

- The referential constraint cannot be defined if each of the existing relationships that would be part of the cycle have a delete rule of CASCADE.
- CASCADE must not be specified if T2 is delete-connected to T1.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. For example, assume that T1 is a dependent of T3 in a relationship with a delete rule of *r* and that one of the following is true:

- T2 and T3 are the same table.
- T2 is a descendent of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendents of the same table and the deletion of rows from that table cascades to both T2 and T3.

In this case, the referential constraint cannot be defined when *r* is SET NULL. When *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

ENFORCED or NOT ENFORCED

Indicates whether or not the referential constraint is enforced by DB2 during normal operations, such as insert, update, or delete.

ENFORCED

Specifies that the referential constraint is enforced by DB2 during normal operations (such as data change operations) and that it is guaranteed to be correct. ENFORCED is the default.

NOT ENFORCED

Specifies that the referential constraint is not enforced by DB2 during normal operations (such as data change operations). NOT ENFORCED should only be used when the data that is stored in the table is verified to conform to the constraint by some other method than relying on DB2.

ENABLE QUERY OPTIMIZATION

Specifies that the constraint can be used for query optimization. DB2 uses the information in query optimization using materialized query tables with the assumption that the constraint is correct. This is the default.

ADD check-constraint:

CONSTRAINT *constraint-name*

Names the check constraint. If *constraint-name* is not specified, a unique constraint name is derived from the name of the first column in the *check-condition* specified in the definition of the check constraint. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

CHECK (*check-condition*)

Defines a check constraint. At any time, *check-condition* must be true or unknown for every row of the table. A *check-condition* can evaluate to unknown if a column that is an operand of the predicate is null. A *check-condition* that evaluates to unknown does not violate the check constraint. A *check-condition* is a search condition, with the following restrictions:

- It can refer only to the columns of table *table-name*; however, the columns cannot be LOB, ROWID, XML, DECFLOAT, or security label columns (including distinct types that are based on LOB, ROWID, and DECFLOAT data types).
- It can be up to 7400 bytes long, not including redundant blanks.
- It must not contain any of the following:
 - Subselects
 - Built-in or user-defined functions
 - CAST specifications
 - Cast functions other than those created when the distinct type was created
 - Host variables
 - Parameter markers
 - Special registers
 - Columns that include a field procedure
 - CASE expressions
 - ROW CHANGE expressions
 - row expressions
 - DISTINCT predicates
 - GX constants (hexadecimal graphic string constants)
 - sequence references
 - OLAP specifications
- If a *check-condition* refers to a LOB column (including a distinct type that is based on a LOB), the reference must occur within a LIKE predicate.
- The AND and OR logical operators can be used between predicates. The NOT logical operator cannot be used.
- The first operand of every predicate must be the column name of a column in the table.
- The second operand in the *check-condition* must be either a constant or a column name of a column in the table.
 - If the second operand of a predicate is a constant, and if the constant is:
 - A floating-point number, then the column data type must be floating point.
 - A decimal number, then the column data type must be either floating point or decimal.
 - A big integer number, then the column data type must not be an integer or a small integer
 - An integer number, then the column data type must not be a small integer.
 - A small integer number, then the column data type must be small integer.
 - A decimal constant, then its precision must not be larger than the precision of the column.
 - If the second operand of a predicate is a column, then both columns of the predicate must have:
 - The same data type
 - Identical descriptions with the exception that the specification of the NOT NULL and DEFAULT clauses for the columns can be different, and that string columns with the same data type can have different length attributes

Effects of defining a check constraint on a populated table: When a check constraint is defined on a populated table and the value of the special register CURRENT RULES is 'DB2', the check constraint is not immediately enforced on the table. The check constraint is added to the description of the table, and

the table space that contains the table is placed in a check pending status. For a description of the check pending status and the implications for utility operations, see *DB2 Utility Guide and Reference*.

When a check constraint is defined on a populated table and the value of the special register CURRENT RULES is 'STD', the check constraint is checked against all rows of the table. If no violations occur, the check constraint is added to the table. If any rows violate the new check constraint, an error occurs and the description of the table is unchanged.

DROP constraint:

DROP PRIMARY KEY

Drops the definition of the primary key and all referential constraints in which the primary key is a parent key. The table must have a primary key and the privilege set must include the ALTER or REFERENCES privilege on every dependent table of the table.

The description of the primary index is changed to indicate that it is not a primary index. If the table space was implicitly created, the corresponding enforcing index is dropped if the primary key is dropped.

DROP UNIQUE *constraint-name*

Drops the definition of the unique key constraint and all referential constraints in which the unique key is a parent key. The table must have a unique key. The privilege set must include the ALTER or REFERENCES privilege on every dependent table of the table. The description of the enforcing index is changed to indicate that it is not enforcing a unique key constraint. If the table space is implicitly created, the corresponding enforcing index is dropped if the unique key is dropped.

DROP FOREIGN KEY *constraint-name*

Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint in which the table is the dependent table, and the privilege set must include the ALTER or REFERENCES privilege on the parent table of that relationship, or the REFERENCES privilege on the columns of the parent table of that relationship.

DROP CHECK *constraint-name*

Drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint defined on the table.

DROP CONSTRAINT *constraint-name*

Drops the constraint *constraint-name*. The *constraint-name* must identify an existing primary key, unique key, check, or referential constraint defined on the table.

DROP CONSTRAINT must not be used on the same ALTER TABLE statement as DROP PRIMARY KEY, DROP UNIQUE KEY, DROP FOREIGN KEY or DROP CHECK.

ADD partitioning:

ADD PARTITION BY RANGE

Specifies the range partitioning scheme for the table (the columns used to partition the data). When this clause is specified, the table uses table-controlled partitioning. The number of partitions specified in the ADD PARTITION BY RANGE clause has to be the same as the number of partitions defined in the table space.

This clause applies only to tables in a partitioned table space. If the table is already complete by having established either table-controlled partitioning or index-controlled partitioning, the ADD PARTITION BY RANGE clause is not allowed. If this clause is used, then the ENDING AT clause cannot be used on a subsequent CREATE INDEX statement for this table.

partition-expression

Specifies the key data over which the range is defined to determine the target data partition of the data.

column-name

Specifies the columns of the key. Each *column-name* must identify a column of the table. Do not specify more than 64 columns, the same column more than one time, a BINARY, VARBINARY, LOB, XML, or DECFLOAT column, a column with a distinct type that is based on a these data type, a qualified column name, or a row change timestamp column. The sum of length attributes of the columns must not be greater than 255 - *n*, where *n* is the number of columns that can contain null values.

NULLS LAST

Specifies that null values are treated as positive infinity for purposes of comparison.

ASC

Puts the entries in ascending order by the column. ASC is the default.

DESC

Puts the entries in descending order by the column.

partition-element

Specifies ranges for a data partitioning key and the table space where rows of the table in the range will be stored.

PARTITION integer

Specifies a number of a physical partition in the table space. A PARTITION clause must be specified for every partition of the table space. In the context, highest means highest in the sorting sequence of the columns. In a column that is defined as ascending (ASC), highest and lowest have the usual meanings. In a column that is defined as descending (DESC), the lowest actual value is the highest in the sorting sequence.

ENDING AT (constant, MAXVALUE, or MINVALUE...)

Specifies the limit key for a partition boundary. Specify at least one value (constant, MAXVALUE, or MINVALUE) after ENDING AT in each PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key for ascending and the lowest for descending.

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant

must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the rules listed for the ENDING AT clause for a partition definition. See list of rules.

INCLUSIVE

Specifies that the specified range values are included in the data partition.

ADD PARTITION:

ADD PARTITION

Specifies that a partition is added to the table and each partitioned index on the table. The new partition is the next physical partition not being used until the maximum for the table space has been reached. ADD PARTITION must not be specified for nonpartitioned tables or for tables in a partition-by-growth table space. Adding a partition is not allowed if the table is a materialized query table or a materialized query table is defined on the table. However, adding a partition is allowed if an accelerated query table is defined on the table. A partition cannot be added if the table space definition is incomplete because a partitioning key or partitioning index is missing. If the table uses index-controlled partitioning, it is converted to use table-controlled partitioning.

The maximum number of partitions allowed depends on how the table space was originally created. If DSSIZE was specified when the table space was created, it is non-zero in the catalog. The maximum number of partitions allowed is shown in Table 89.

Table 89. Maximum number of partitions allowed

DSSIZE	Page size 4	Page size 8 KB	Page size 16	Page size 32 KB
	KB		KB	
1GB-4GB	4096	4096	4096	4096
8GB	2048	4096	4096	4096
16GB	1024	2048	4096	4096
32GB	512	1024	2048	4096
64GB	256	512	1024	2048

If LARGE was specified when the table space was created, the maximum number of partitions is shown in the fourth row of Table 90. For more than 254 partitions when LARGE or DSSIZE is not specified, the maximum number of partitions is determined by the page size of the table space.

Table 90. Maximum number of partitions when DSSIZE = 0

Type of table space	Number of existing partitions	Maximum partitions
non-large	1 to 16	16
non-large	17 to 32	32
non-large	33 to 64	64
large	N/A	4096

The existing table space PRIQTY and SECQTY attributes of the previous logical partition are used for the space attributes of the new partition. For each partitioned index, the existing PRIQTY and SECQTY attributes of the previous partition are used.

To specify specific space attributes for the new partition, use additional ALTER TABLESPACE and ALTER INDEX statements.

ENDING AT (*constant*, MAXVALUE, or MINVALUE, ...)

Specifies the high key limit for the new partition. The new partition's key limit must be higher when partitioning is ascending and lower when it is descending. Specify at least one value (*constant*, MAXVALUE, or MINVALUE) after ENDING AT in the PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the corresponding partition of the index.

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on. Using fewer values than there are columns in the key has the same effect as using the highest or lowest values for the omitted columns, depending on whether they are ascending or descending.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition.
- The values specified for the last partition are enforced. The value specified for the last partition is the highest value of the key that can be placed in the table. If the limit was not previously enforced, any existing key values that are greater than the value that is specified for the added partition are placed into the discard data set when REORG is run.
- If a key includes a ROWID column or a column with a distinct type that is based on a ROWID data type, 17 bytes of the constant that is specified for the corresponding ROWID column are considered.
- The combination of the number of table space partitions and the corresponding limit key size cannot exceed the number of partitions * (106 + limit key size in bytes) < 65394
- If the concatenation of all the values exceeds 255 bytes, only the first 255 bytes are considered.

INCLUSIVE

Specifies that the specified range values are included in the data partition.

ALTER PARTITION:

ALTER PARTITION

Specifies that the partitioning limit key for the identified partition is to be changed.

This clause applies only to tables in a partitioned table space. ALTER PARTITION must not be specified for a table in a partition-by-growth table space or for tables that have XML columns or LOB columns (or distinct type columns that are based on LOBs). ALTER PARTITION must also not be specified if the table is a materialized query table or if a materialized query table is defined on the specified table. However, adding a partition is allowed if an accelerated query table is defined on the table.

integer

If *integer* is specified, it must be in the range 1 to *n*, where *n* is the number of partitions in the table. Changing a partition boundary is not allowed if the table is a materialized query table or if a materialized query table is defined from this table. When this option is specified for any partition except for the last, both the identified partition and the partition following are placed in REORG-pending (REORP) status.

ENDING AT (*constant*, MAXVALUE, or MINVALUE...)

Specifies the highest value of the partitioning key for the identified partition.

In this context, highest means highest in the sorting sequences of the columns. In a column defined as ascending (ASC), highest and lowest have their usual meanings. In a column defined as descending (DESC) the lowest actual value is highest in the sorting sequence.

Specify at least one value after ENDING AT in each ALTER PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the

corresponding partition. The length of each highest key value (the limit key) is the same as the length of the partitioning key.

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the rules listed for the ENDING AT clause for a partition definition. See list of rules.

The value that is specified must not be equal to or beyond the range of the partition boundaries of the adjacent partitions.

INCLUSIVE

Specifies that the specified range values are included in the data partition.

If the table uses index-controlled partitioning, it is converted to use table-controlled partitioning. The high limit key for the last partition is set to the highest possible value for ascending key columns or the lowest possible value for descending key columns.

ROTATE PARTITION:

ROTATE PARTITION FIRST TO LAST

Specifies that the first logical partition should be rotated to become the last logical partition. The table definition must be complete and must contain more than one partition. This clause must be followed by the ENDING AT clause, which specifies the new high key limit for this partition, which is now logically last. The new partitioning key value must be higher than the current high key limit if partition values are ascending. If partition values are descending, the new key limit must be lower than the current low key limit.

Rotating a partition occurs immediately. If there is a referential constraint with DELETE RESTRICT on the table, the ROTATE might fail. If the table uses index-controlled partitioning, it is converted to use table-controlled partitioning.

After an ALTER TABLE statement with the ROTATE PARTITION clause is run, the RUNSTATS utility with REORG should be run on the table space to ensure effective access paths are available for selection.

If the table has a security label column, the user must have a valid security label to rotate partitions. In addition, if write-down is in effect, the user must have the write-down privilege.

ROTATE PARTITION must not be specified in the following situations:

- The table is a materialized query table or a materialized query table is defined on the table
- The table is in a partition-by-growth table space
- The table that has XML columns

ENDING AT (*constant*, MAXVALUE, or MINVALUE...)

The ENDING AT clause specifies the new high key limit for the existing partition holding the oldest data.

In this context, highest means highest in the sorting sequences of the columns. In a column defined as ascending (ASC), highest and lowest have their usual meanings. In a column defined as descending (DESC) the lowest actual value is highest in the sorting sequence.

Specify at least one value after ENDING AT. You can use as many values as there are columns in the key. The concatenation of all the values is the highest value of the key in the corresponding partition. The length of each highest key value (the limit key) is the same as the length of the partitioning key.

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must specify MAXVALUE.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the rules listed for the ENDING AT clause for a partition definition. See list of rules.

INCLUSIVE

Specifies that the specified range values are included in the data partition.

RESET

Specifies that the existing data in the oldest partition is deleted. In a partitioned table with limit values that are in ascending sequence, ALTER TABLE ROTATE PARTITION FIRST TO LAST logically operates as if the partition with the lowest high key limit were dropped and then a new partition was added with the specified high key limit. The new key limit for the partition must be higher than any other partition in the table. For descending limit keys, the rotation operates as the partition with the highest limit values becomes the partition with the lowest limit values.

If the partition contains referential integrity parent relationships, has DATA CAPTURE logging enabled, or has a delete row trigger, then each data row in the partition must be deleted individually. If a table does not have any of these attribute settings, then the data rows are removed by deleting and redefining the underlying data sets.

ADD MATERIALIZED QUERY:

ADD MATERIALIZED QUERY *materialized-query-definition*

Changes a base table to a materialized query table. Supplies a definition for a regular table to make it a materialized query table. The table specified by *table-name* and the result columns of the fullselect must not have the following characteristics:

- Be already defined as a materialized query table
- Have any primary keys, unique constraints (unique indexes), referential constraints (foreign keys), check constraints, or triggers defined
- Be referenced in the definition of another materialized query table
- Be directly or indirectly referenced in the *fullselect*
- Be in an incomplete state

If *table-name* does not meet these criteria, an error occurs.

The ADD MATERIALIZED QUERY clause cannot be specified in a Common Criteria environment.

fullselect

Defines the query on which the table is based. The columns of the existing table must meet the following characteristics:

- Have the same number of columns
- Have exactly the same column definitions
- Have the same column names in the same ordinal positions

If fullselect is specified, the owner of the table being altered must have the SELECT privilege on the tables or views referenced in the fullselect. Having SELECT privilege means that the owner has at least one of the following authorizations:

- Ownership of the tables or views referenced in the fullselect
- The SELECT privilege on the tables and views referenced in the fullselect
- SYSADM authority
- DBADM authority for the database in which the table of the fullselect reside

If the owner of the table does not have the SELECT privilege, the following authorization IDs must have SYSADM authority or DBADM authority for the database in which the tables of the fullselect reside:

- For embedded statements, the authorization ID of the owner of the plan or package
- For dynamically prepared statements, the SQL authorization ID of the process

For details about specifying *fullselect* for a materialized query table, see the definition of *fullselect* in the “CREATE TABLE” on page 1079 statement.

Altering a table to change it from a base table to a materialized query table with REFRESH DEFERRED causes any plans and packages dependent on the table to be invalidated.

refreshable-table-options

Specifies the materialized query table options for altering a regular table to a materialized query table. The ORDER BY clause is allowed, but it is used only by REFRESH. The ORDER BY clause can improve the locality of reference of data in the materialized query table.

DATA INITIALLY DEFERRED

Specifies that the data in the table is not validated as part of the ALTER TABLE statement. A REFRESH TABLE statement can be used to make sure the data in the materialized query table is the same as the result of the query in which the table is based.

REFRESH DEFERRED

Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or as updated by the user for a user-maintained materialized query table.

MAINTAINED BY SYSTEM or MAINTAINED BY USER

Specifies how the data in the materialized query table is maintained.

MAINTAINED BY SYSTEM

Specifies that the data in the materialized query table *table-name* is to be maintained by the system. Only the REFRESH TABLE statement is allowed on the table.

MAINTAINED BY USER

Specifies that the data in materialized query table *table-name* is to be maintained by the user, who can use LOAD utility or SQL data change statements and REFRESH TABLE statements on the table.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION

Specifies whether this materialized query table can be used for optimization.

ENABLE QUERY OPTIMIZATION

Specifies that the materialized query table can be used for query optimization. If the fullselect specified does not satisfy the restrictions for query optimization, an error occurs. For detailed rules to satisfy query optimization, see *materialized-query-definition* in the “CREATE TABLE” on page 1079 statement.

DISABLE QUERY OPTIMIZATION

Specifies that the materialized query table cannot be used for query optimization. The table can still be queried directly.

ALTER MATERIALIZED QUERY:

ALTER MATERIALIZED QUERY *materialized-query-table-alteration*

Changes attributes of a materialized query table. The *table-name* must identify a materialized query table.

SET *refreshable-table-alteration*

Changes how the table is maintained or whether the table can be used in query optimization.

MAINTAINED BY SYSTEM

Specifies that the data in a materialized query table *table-name* is to be maintained by the system.

MAINTAINED BY USER

Specifies that the data in the materialized query table *table-name* is to be maintained by the user.

ENABLE QUERY OPTIMIZATION

Specifies that materialized query table *table-name* can be used in query optimization. If the fullselect specified for the materialized query table does not satisfy the restrictions for automatic query optimization, an error occurs. For detailed rules to satisfy query optimization, see “CREATE TABLE” on page 1079.

DISABLE QUERY OPTIMIZATION

Specifies that materialized query table *table-name* cannot be used for query optimization. The table can still be queried directly.

DROP MATERIALIZED QUERY:

DROP MATERIALIZED QUERY

Changes a materialized query table so that it is no longer considered a materialized query table. The table specified by *table-name* must be defined as a materialized query table. The definition of columns and data of the name are not changed, but the table can no longer be used for query optimization and is no longer valid for use with the REFRESH TABLE statement.

Altering a table to change from a materialized query table to a base table with the DROP MATERIALIZED QUERY clause causes any plans and packages dependent on the table to be invalidated.

DATA CAPTURE:

DATA CAPTURE

Specifies whether the logging of the following actions on the table is augmented by additional information:

- SQL data change operations
- Adding columns (using the ADD COLUMN clause)
- Changing columns (using the ALTER COLUMN clause)

For guidance on intended uses of the expanded log records, see:

- The description of data propagation to IMS in *IMS DataPropagator: An Introduction*
- The instructions for using Remote Recovery Data Facility (RRDF) in *Remote Recovery Data Facility Program Description and Operations*
- The instructions for reading log records in *DB2 Administration Guide*

NONE

Do not record additional information to the log.

CHANGES

Write additional data about SQL updates to the log. Information about the values that are represented by any LOB or XML columns is not available. Do not specify DATA CAPTURE CHANGES for tables that reside in table spaces that specify NOT LOGGED.

For details about the recording of additional data for logged updates to catalog tables, see “Notes” on page 785.

VOLATILE:

VOLATILE or NOT VOLATILE

Specifies how DB2 is to choose access to the table.

VOLATILE

Specifies that DB2 is to use index access to the table whenever possible for SQL operations. However, be aware that list prefetch and certain other optimization techniques are disabled when VOLATILE is used.

One instance in which use of VOLATILE might be desired is for a table whose size can vary greatly. If statistics are taken when the table is empty or has only a few rows, those statistics might not be appropriate when the table has many rows. Another instance in which use of VOLATILE might be desired is for a table that contains groups of rows, as defined by the primary key on the table. All but the last column of the primary key of such a table indicate the group to which a given row belongs. The last column of the primary key is the sequence number indicating the order in which the rows are to be read from the group. VOLATILE maximizes concurrency of operations on rows within each group, since rows are usually accessed in the same order for each operation.

NOT VOLATILE

Specifies that DB2 is to base SQL access to the table on the current statistics.

CARDINALITY

An optional keyword that currently has no effect, but that is provided for DB2 family compatibility.

ADD CLONE:

ADD CLONE *clone-table-name*

Specifies that a clone table, identified by *clone-table-name*, is created for the table that is being altered. The clone table is created in the same table space as the base table and has the same structure as the base table. This includes, but is not limited to, column names, data types, null attributes, check constraints, indexes. When ADD CLONE is used to create a clone of the specified base table, the base table must conform to the following rules:

- Reside in a DB2-managed universal table space
- Be the only table in the table space
- Not be defined with a clone table
- Not be involved in any referential constraint
- Not be defined with any after triggers
- Not be a materialized query table
- If the table space is created with the DEFINE NO clause, all data sets must already be created
- Not have any pending changes

- Not have any active versioning
- Not have an incomplete definition
- Not be a created global temporary table or a declared global temporary table

DROP CLONE:

DROP CLONE

Specifies that the clone table that is associated with the specified base table is dropped. *table-name* must identify a base table that exists at the current server and the table must have a clone table defined.

RESTRICT ON DROP:

ADD RESTRICT ON DROP

Restricts dropping the table and the database and table space that contain the table.

DROP RESTRICT ON DROP

Removes the restriction on dropping the table and the database and table space that contain the table.

APPEND:

APPEND NO or APPEND YES

Specifies whether append processing is used for the table. The APPEND clause must not be specified for a table in a work file table space.

If the base table is in a range-partitioned table space, the APPEND option on the LOB table might be different for each partition (depending if the LOB table space and associated objects for each partition are created explicitly or implicitly). If the base table is in a partition by growth table space, the APPEND attributes of LOB table will be inherited by each partition.

NO Specifies that append processing is not used for the table. For insert and LOAD operations, DB2 will attempt to place data rows in a well clustered manner with respect to the value in the row's cluster key columns.

YES

Specifies that data rows are placed into the table without regard to clustering during the insert and LOAD operations.

AUDIT:

AUDIT

Alters the auditing attribute of the table. For information about audit trace classes, see *DB2 Administration Guide*.

NONE

Specifies that no auditing is to be done when the table is accessed.

CHANGES

Specifies that auditing is to be done when the table is accessed during the first insert, update, or delete operation. However, the auditing is done only if the appropriate audit trace class is active.

ALL

Specifies that auditing is to be done when the table is accessed during the first operation of any kind performed by a utility or application process. However, the auditing is done only if the appropriate audit trace class is active and the access is not performed with COPY, RECOVER, REPAIR, or any stand-alone utility.

The ALTER TABLE statement is audited for successful and failed attempts in the following cases, if the appropriate audit trace class is active:

- **AUDIT** attribute is changed to **NONE**, **CHANGES**, or **ALL** on an audited or non-audited table.
- **AUDIT CHANGES** or **AUDIT ALL** is in effect.

VALIDPROC:

VALIDPROC

Names a validation procedure for the table or inhibits the execution of any existing validation procedure.

program-name

Designates *program-name* as the new validation exit routine for the table. Validation exit routines are described in *DB2 Administration Guide*.

The validation procedure can inhibit a data change operation on any row of the table. Before the operation takes place, the row is passed to the procedure. The values that are represented by any LOB or XML columns in the table are not passed to the validation procedure. On an insert or update operation, if the table has a security label column and the user does not have write-down privilege, the user's security label value is passed to the validation routine as the value of the column. After examining the row, the procedure returns a value that indicates whether the operation should proceed. A typical use is to impose restrictions on the values that can appear in various columns.

A table can have only one validation procedure at a time. When you name a new procedure, any existing procedure is no longer used. The new procedure is not used to validate existing table rows. It is used only to validate rows that are loaded, inserted, updated, or deleted after execution of the ALTER TABLE statement.

NULL

Discontinues the use of any validation routine for the table.

Notes

Order of processing of clauses: When there is more than one clause, they are processed in the following order:

1. VALIDPROC
2. AUDIT
3. DATA CAPTURE
4. ROTATE
5. VOLATILE clauses
6. APPEND clauses
7. DROP clauses
8. ALTER clauses
9. RENAME clause
10. ADD clauses

Within each of these stages, the order in which the user specifies the clauses is the order in which they are performed.

Altering the data type, length, precision, or scale of a column: When you change the data type, length, precision, or scale of a column, the following information applies to indexes, limit keys, check constraints, and invalidation:

- *Restrictions.* The ALTER TABLE statement is not allowed if any of the following conditions are true:
 - The column is referenced in a referential constraint.
 - The column has a field procedure routine.
 - The column is defined as an identity column.
 - The column is defined as an security label column
 - The table has an edit or validation routine.
 - The table is defined with DATA CAPTURE CHANGES and the ALTER TABLE statement causes a new version of the table to be generated.
 - The table is a created temporary table.
 - The table is a materialized query table or the table is referenced by a materialized query table.
 - The data type changed is not to a compatible data type.
 - The new length or data type specification could result in a loss of significance because of a shorter length or less precision in the data type.
 - For a conversion from decimal to float, a unique index or a unique constraint exists on the column.
 - For a conversion from other numeric data type to DECFLOAT, a partitioning key, check constraint, index, or a unique constraint exists on the column.
 - The existing default value for a column cannot be assigned to the new data type.
 - Increasing the column length results in an existing index that references the column exceeding the maximum size of an index.
 - Increasing the column length results in the partitioning key using that column exceeding the maximum size for a partitioning key.
 - Table definition is incomplete because unique index for enforcing a unique constraint (primary key or unique key) is missing.
- *Indexes.*
 - If the index has a changed character column, the index is in advisory REORG-pending (AREO*) status.
 - If the index has a changed numeric column, the index remains in REBUILD-pending (RBDP) status.
- *Length of partitioned index keys.* When a table is altered and the length of a column in the PARTITIONING KEY is changed, DB2 changes the length of the limit key (the highest key value) for a partition too. The length of the limit key is increased by the same amount that the length of the column is increased.
- *Check constraints.* If a check constraint refers to the column being altered, the length of the column is also changed in the check constraint.
- *Statistics.* The RUNSTATS utility should be run to collect new COLUMN statistics for all altered columns. Even though the COLCARD value is valid, the HIGH2KEY and LOW2KEY values are invalid, and any SYSCOLSTATS catalog entries for the column are removed. Any frequencies or histogram statistics which include this column should also be collected again.

When you change a column from a fixed to varying length or change the length of a varying-length column, process the ALTER TABLE statements in the same unit of

work or do a reorganization between the ALTER TABLE statements to avoid anomalies with the lengths and padding of individual values.

Referencing columns in ADD, ALTER, and RENAME clauses: A column can only be referenced once in an ADD COLUMN, an ALTER COLUMN, or a RENAME COLUMN clause in a single ALTER TABLE statement. However, that same column can be referenced multiple times for adding or dropping constraints in the same ALTER TABLE statement.

Because a distinct type is subject to the same restrictions as its source type, all the syntactic rules that apply to LOB, ROWID, and DECFLOAT columns apply to distinct type columns that are based on LOBs, row IDs, and DECFLOATs. For example, if a table has an explicitly created ROWID column, you cannot add a column with a distinct type that is sourced on a row ID.

Adding a column to table T only changes the description of T. If the catalog description of T is used to create a table T' and a facility such as DSN1COPY is used to effectively copy T into T', queries that refer to the added column in T' will fail because the data does not match its description. To avoid this problem, run the REORG utility against the table space of T before making the copy.

Restrictions on a clone table: Tables that are involved in a clone relationship (base tables and their associated clone tables) have the following restrictions:

- You cannot use the RUNSTATS utility on a clone table. When a data exchange is done using the EXCHANGE statement, real time statistics for the base table are invalidated.
- Objects that are involved in a clone relationship do not use the FASTSWITCH naming convention when the REORG utility is run. This includes both the base table and the clone table objects (data and index), as well as LOB and XML objects.
- For a partitioned table, if a mixture of 'I' and 'J' data sets exists when a clone table is created, the mixture of 'I' and 'J' data sets can be changed only by first dropping the clone table.
- Catalog and directory tables cannot have clone tables.
- Indexes cannot be created on a clone table. When an index is created on a base table that is involved in a clone relationship, the index on the clone table will be created implicitly and will be put into rebuild-pending status.
- Before triggers cannot be created on a clone table. Before triggers that are created on a base table apply to both the base table and the clone table.
- You cannot rename a base table that has a clone and you cannot rename a clone table.
- Real time statistics tables cannot have clone tables.
- You cannot drop an auxiliary table or an auxiliary index of an object that is involved in a clone relationship.

If the table is involved in a clone relationship, no other table altering can take place. If a table change is required, the clone table objects must be dropped so that the base table object attributes can be modified. After the desired table, index, etc. changes are completed, the clone table objects can be recreated.

Altering the attributes of an existing identity column: Existing values for the identity column are unaffected by the ALTER TABLE statement. The changed identity column attributes affect values generated after the ALTER statement has executed. DB2 does not validate any of the existing identity column values against

the new identity column attributes. For example, duplicate values might be generated even if NO CYCLE is in effect, such as when an ascending identity column altered to become a descending identity column.

Any existing values in the cache that have not yet been used might be lost. Loss of cached values can also occur if the ALTER statement returns an error or is rolled back.

Adding a LOB column: If the table space that contains the table is implicitly created and you add a LOB column to the table, the following object are implicitly created:

- A LOB table space
- An auxiliary table
- An auxiliary index

Adding a ROWID column: When you add a ROWID column to an existing table, DB2 ensures that the same, unique row ID value is returned for a row whenever it is accessed. If the table already has an implicitly hidden ROWID column, DB2 also ensures that the values in the two ROWID columns are identical.

If the table space that contains the table is implicitly created and you add a ROWID column that is defined as GENERATED BY DEFAULT to the table, an enforcing index for the ROWID column is implicitly created. If the table already has an implicitly hidden ROWID column and the ROWID column that you add is defined as GENERATED BY DEFAULT, DB2 changes the implicitly hidden ROWID column to have the GENERATED BY DEFAULT attribute and does not implicitly create an enforcing index for the ROWID column.

When you add a ROWID column that is defined as GENERATED BY DEFAULT and the ROWID index is implicitly created, the privilege set requires the INDEX privilege on the table and the USE privilege on the buffer pool and the storage group. The implicitly created ROWID index is owned by the owner of the table.

Reorganizing a table space has no effect on the values in a ROWID column.

Adding an identity column: When you add an identity column to a table that is not empty, DB2 places the table space that contains the table in the REORG pending state. When the REORG utility is subsequently run, DB2 generates the values for the identity column in all existing rows and then removes the REORG pending status. These values are guaranteed to be unique, and their order is system-determined.

Adding a row change timestamp column: When you add a row change timestamp column to an existing table, the initial value for existing rows is not stored at the time of the ALTER statement. DB2 places the table space into an advisory-REORG pending state. When the REORG utility is subsequently run, DB2 generates the values for the row change timestamp column in all existing rows and then removes the REORG pending status. These values will not change unless the row is updated.

Effect of adding a column on views: Adding a column to a table has no effect on existing views.

Considerations for implicitly hidden columns: A column that is defined as implicitly hidden can be explicitly referenced on the ALTER statement. For

example, an implicitly hidden column can be altered, can be specified as part of a referential constraint or a check constraint, or a materialized query table definition.

Cascaded effects of adding or altering a column: Adding a column to a table has no cascaded effects to views that reference the table. For example, adding a column to a table does not cause the column to be added to any dependent views, even if those views were created with a SELECT clause. But altering a column can cause other cascaded effects. Table 91 lists the cascaded effect of altering the data type, precision, scale, or length of a column.

Table 91. Cascaded effect of altering a column's data type, precision, scale, or length

Operation	Effect
Alter of a column referenced by a view	If the data type, length, precision, or scale for a column is altered, all the views that are dependent on the altered table are reevaluated at alter time with the new column attributes. If errors are encountered during the view regeneration process, the ALTER TABLE statement fails. The new internal structure of each dependent view is not saved at alter time, and subsequent references to a dependent view will cause the view to be regenerated again. Use the ALTER VIEW statement to regenerate a dependent view and have the new internal structure saved.
Alter of a column referenced in the key of an index or a unique constraint (unique key or primary key)	The alter is allowed unless DECIMAL with a fraction is being converted to a floating value. In this case, the loss of precision can result in a loss of uniqueness. For numeric data type conversions, the index is placed in REBUILD-pending status. For character data type conversions, the index key columns are converted on first-write access. The index is not placed in REBUILD-pending status.
Alter of a column referenced in a package or plan	The alter is allowed. All plans and packages dependent on the table in which the column is being altered are invalidated.
Alter of a column referenced in the body of a user-defined function or procedure	Alter is allowed. If there is a package associated with the function or procedure, it is invalidated.
Alter of a column referenced in the parameter list of a user-defined function or procedure	Alter is allowed. The attributes of the existing function or procedure are unchanged. To access the new definition of the column, the function or procedure must be dropped and recreated.
Alter of a column referenced by a trigger	Alter is allowed. All trigger packages dependent on the table of the column are invalidated.
Alter of a column referenced in a CHECK constraint	Alter is not allowed.

Adding a partition: When you add a partition to a table, if the boundary for the last partition was not previously enforced, it is enforced after the partition is added, and the last two logical partitions are left in REORG-pending (REORP)

status. If the last partition before the new one is added was in REORG-pending status, the added partition is also placed in REORG-pending status.

Row format for newly added partitions: When the value of the SPRMRRF subsystem parameter is ENABLE, newly added partitions that are created using the ADD PARTITION clause (or partitions that are added because the table space is partition-by-growth) will be created in re-ordered row format. When the value of the SPRMRRF subsystem parameter is DISABLE, newly added partitions will be created in basic row format, except for the following:

- For table spaces that are already using basic row format and that contain tables with edit procedures, newly created partition will always be in basic row format regardless of value of the SPRMRRF parameter.
- For table spaces that are already using re-ordered row format and that contain tables with edit procedures, newly created partition will always be in re-ordered row format regardless of value of the SPRMRRF parameter.
- Newly created partitions of an XML table space will always be in re-ordered format.

Rotating a partition from first to last: Running ALTER TABLE to rotate the first logical partition to become the last logical partition can be very time consuming. During the reset operation, all rows from the partition are deleted. In addition, the keys for the deleted rows are also deleted from all nonpartitioned indexes, which requires that each nonpartitioned index must be scanned.

When you rotate partitions, if the boundary for the last partition was not previously enforced, it is enforced after ROTATE FIRST TO LAST is issued, and the last two logical partitions are left in REORG-pending (REORP) status. If the last partition before ROTATE FIRST TO LAST was issued was in REORG-pending status, the last two logical partitions are left in REORG-pending status.

Effect of changes on applications: Applications might need to be changed to correspond to changes to the columns in a table. For example, if you increase the length of a column, you need to increase the length of host variables into which that column is fetched. If you change the data type of a column, you also might need to change the data type of the corresponding host variable to avoid performance degradation. If you rename a column, you need to change any references to that column to avoid unexpected results.

Invalidation of plans and packages: When a table is altered, all the plans, packages, and dynamic cached statements that refer to the table are invalidated if any one of the following conditions is true:

- The table is a created temporary table or a materialized query table.
- The table is changed to add or drop a materialized query definition.
- The AUDIT attribute of the table is changed.
- A DATE, TIME, or TIMESTAMP column is added and its default value for added rows is CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, respectively.
- A security label is added.
- The length attribute of a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY column has changed. See Table 88 on page 811.
- The column data type, precision, scale, or subtype is changed.
- The column is renamed.

- The table is partitioned and a partition is added or one of the existing partitions is changed or rotated
- An identity attribute of an identity column has changed

When a referential constraint is defined with a delete rule of CASCADE or SET NULL, all plans and packages that refer to the parent table of the constraint are invalidated. Furthermore, all plans and packages that refer to tables from which deletes cascade to this parent table are also invalidated.

Altering a base table or a user-maintained materialized query table to change it to a system-maintained materialized query table causes any plans and packages dependent on the table to be invalidated because data change statements are not allowed on system-maintained materialized query tables. Altering a materialized query table to change it to a base table causes any plans and packages dependent on the table to be invalidated because the REFRESH TABLE statement is invalid on a base table.

Invalidation of plans and packages by RENAME COLUMN:

ALTER TABLE RENAME COLUMN will invalidate any plan or package that is dependent on the table in which the column is renamed. Any attempt to execute the invalidated plan or package will trigger an automatic rebind of the plan or package.

The automatic rebind will fail if the column is referenced in the plan or package because the referenced column no longer exists in the table. In this case, applications that reference the plan or package need to be modified, recompiled, and rebound to return the expected result.

The automatic rebind will succeed in either of the following cases:

- The plan or package does not reference the column. In this case, the renaming of the column does not affect the query results that are returned by the plan or package. The application does not need to be modified as a result of renaming the column.
- The plan or package does reference the column, but after the column is renamed, another column with the name of the original column is added to the table. In this case, any query that references the name of the original column might return a different result set. In order to restore the expected results, the application would need to be modified to specify the new column name.

Example: The following scenario shows how renaming a column can cause a plan or package to return unexpected results:

```
CREATE TABLE MYTABLE (MYCOL1 INT);
INSERT INTO TABLE MYTABLE
VALUES (1);
SELECT MYCOL1 FROM MYTABLE -- this is the statement in the package MYPACKAGE,
                           -- the query returns a value of 1

ALTER TABLE MYTABLE
  RENAME MYCOL1 TO MYCOL2; -- MYPACKAGE is invalidated and automatic rebind
                           -- of MYPACKAGE will fail at this point

ALTER TABLE MYTABLE
  ADD COLUMN MYCOL1 VARCHAR(10); -- automatic rebind of MYPACKAGE
                                -- will be successful

INSERT INTO TABLE MYTABLE (MYCOL1)
VALUES ('ABCD');
-- at this point an application executes MYPACKAGE which results in a
-- successful automatic rebind. However, the statement in the package
-- will return 'ABCD' instead of the expected '1'
```

Dropping constraints and check pending status: If a table space or partition is in check pending status because it contains a table with rows that violate constraints, dropping the constraints removes the check pending status.

Altering materialized query tables: The ALTER TABLE statement can be used to register an existing table at the current server as a materialized query table, change the attributes of an existing materialized query table, or change an existing materialized query table into a base table.

The isolation level at the time when a base table is first altered to become a materialized query table by the ALTER TABLE statement is the isolation level for the materialized query table.

Altering a table to change it to a materialized query table with query optimization enabled makes the table eligible for use in query rewrite immediately. Therefore, pay attention to the accuracy of the data in the table. If necessary, the table should be altered to a materialized query table with query optimization disabled, and then the table should be refreshed and enabled with query optimization.

When a base table is altered into a materialized query table or a user-maintained query table is altered into a system-maintained one, the REFRESH_TIME column of the row for the table in SYSIBM.SYSVIEWS contains the current timestamp. When a system-maintained materialized query table is altered into a user-maintained materialized query table, the REFRESH_TIME column of the row for the table in SYSIBM.SYSVIEWS does not change.

The LOAD utility is not allowed on a system-maintained query table, but it is allowed on a user-maintained materialized query table.

Considerations for running utilities while altering tables: You cannot execute the ALTER TABLE statement while a utility has control of the table space that contains the table.

Restrictions on field procedures, edit procedures, and validation exit procedures: Field procedures, edit procedures that are defined as WITH ROW ATTRIBUTES, and validation exit procedures cannot be used on tables that have column names that are larger than 18 EBCDIC bytes. If you have tables that have field procedures or validation exit procedures and you add a column where the column name is larger than 18 bytes, the field procedures and validation exit procedures for the table will be invalidated.

Consider using triggers to replace the functionality on field procedures, edit procedures that are defined as WITH ROW ATTRIBUTES, and validation exit procedures on tables where the column names are larger than 18 EBCDIC bytes.

Restrictions on SQL data change statements in the same commit scope as ALTER TABLE: SQL data change statements that affect an index cannot be performed in the same commit scope as ALTER TABLE statements that affect that index.

Restrictions on DATA CAPTURE CHANGES: If the table is in advisory REORG-pending state, you cannot alter the table to use the DATA CAPTURE CHANGES clause.

Capturing changes to the DB2 catalog: To have logged changes to a DB2 catalog table augmented with information for data capture, specify ALTER TABLE xxx DATA CAPTURE CHANGES where xxx is the name of a catalog table

(SYSIBM.xxx). Data capture of catalog table changes provides the possibility of creating and managing a shadow of the catalog.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following clauses:

- NOCACHE (single clause) as a synonym for NO CACHE
- NOCYCLE (single clause) as a synonym for NO CYCLE
- NOMINVALUE (single clause) as a synonym for NO MINVALUE
- NOMAXVALUE (single clause) as a synonym for NO MAXVALUE
- NOORDER (single clause) as a synonym for NO ORDER
- PART *integer* VALUES can be specified as an alternative to PARTITION *integer* ENDING AT.
- VALUES as a synonym for ENDING AT
- DEFINITION ONLY as a synonym for WITH NO DATA
- SET MATERIALIZED QUERY AS DEFINITION ONLY as a synonym for DROP MATERIALIZED QUERY
- SET SUMMARY AS DEFINITION ONLY as a synonym for DROP MATERIALIZED QUERY
- SET MATERIALIZED QUERY AS (fullselect) as a synonym for ADD MATERIALIZED QUERY (fullselect)
- SET SUMMARY AS (fullselect) as a synonym for ADD MATERIALIZED QUERY (fullselect)

Examples

Example 1: Column DEPTNAME in table DSN8910.DEPT was created as a VARCHAR(36). Increase its length to 50 bytes. Also, add the column BLDG to the table DSN8910.DEPT. Describe the new column as a character string column that holds SBCS data.

```
ALTER TABLE DSN8910.DEPT
  ALTER COLUMN DEPTNAME SET DATA TYPE VARCHAR(50)
  ADD BLDG CHAR(3) FOR SBCS DATA;
```

Example 2: Assign a validation procedure named DSN8EAEM to the table DSN8910.EMP.

```
ALTER TABLE DSN8910.EMP
  VALIDPROC DSN8EAEM;
```

Example 3: Disassociate the current validation procedure from the table DSN8910.EMP. After the statement is executed, the table no longer has a validation procedure.

```
ALTER TABLE DSN8910.EMP
  VALIDPROC NULL;
```

Example 4: Define ADMRDEPT as the foreign key of a self-referencing constraint on DSN8910.DEPT.

```
ALTER TABLE DSN8910.DEPT
  FOREIGN KEY(ADMRDEPT) REFERENCES DSN8910.DEPT ON DELETE CASCADE;
```

Example 5: Add a check constraint to the table DSN8910.EMP which checks that the minimum salary an employee can have is \$10,000.

```
ALTER TABLE DSN8910.EMP
  ADD CHECK (SALARY >= 10000);
```

Example 6: Alter the PRODINFO table to define a foreign key that references a non-primary unique key in the product version table (PRODVER_1). The columns of the unique key are VERNAME, RELNO.

```
ALTER TABLE PRODINFO
  FOREIGN KEY (PRODNAME,PRODVERNO)
    REFERENCES PRODVER_1 (VERNAME,RELNO) ON DELETE RESTRICT;
```

Example 7: Assume that table DEPT has a unique index defined on column DEPTNAME. Add a unique key constraint named KEY_DEPTNAME consisting of column DEPTNAME to the DEPT table:

```
ALTER TABLE DSN8910.DEPT
  ADD CONSTRAINT KEY_DEPTNAME UNIQUE( DEPTNAME );
```

Example 8: Register the base table TRANSCOUNT as a materialized query table. The result of the fullselect must provide a set of columns that match the columns in the existing table (same number of columns, same column definitions, and same names). So that you can maintain the table with insert, update, and delete operations as well as the REFRESH TABLE statement, define the materialized query table as user-maintained.

```
ALTER TABLE TRANSCOUNT ADD MATERIALIZED QUERY
  (SELECT ACCTID, LOCID, YEAR, COUNT(*) as cnt
   FROM TRANS
   GROUP BY ACCTID, LOCID, YEAR )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER;
```

Example 9: Assume that table TB1 has a column, COL1 that is defined as CHAR(4) FOR BIT DATA WITH DEFAULT 'AB'. The value that is stored in the table will be X'C1C24040'. After the following ALTER TABLE statement is run, the resulting value that is stored in the table will be BX'C1C240400000':

```
ALTER TABLE TB1
  ALTER COLUMN COL1
  SET DATA TYPE BINARY(6);
```

ALTER TABLESPACE

The ALTER TABLESPACE statement changes the description of a table space at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

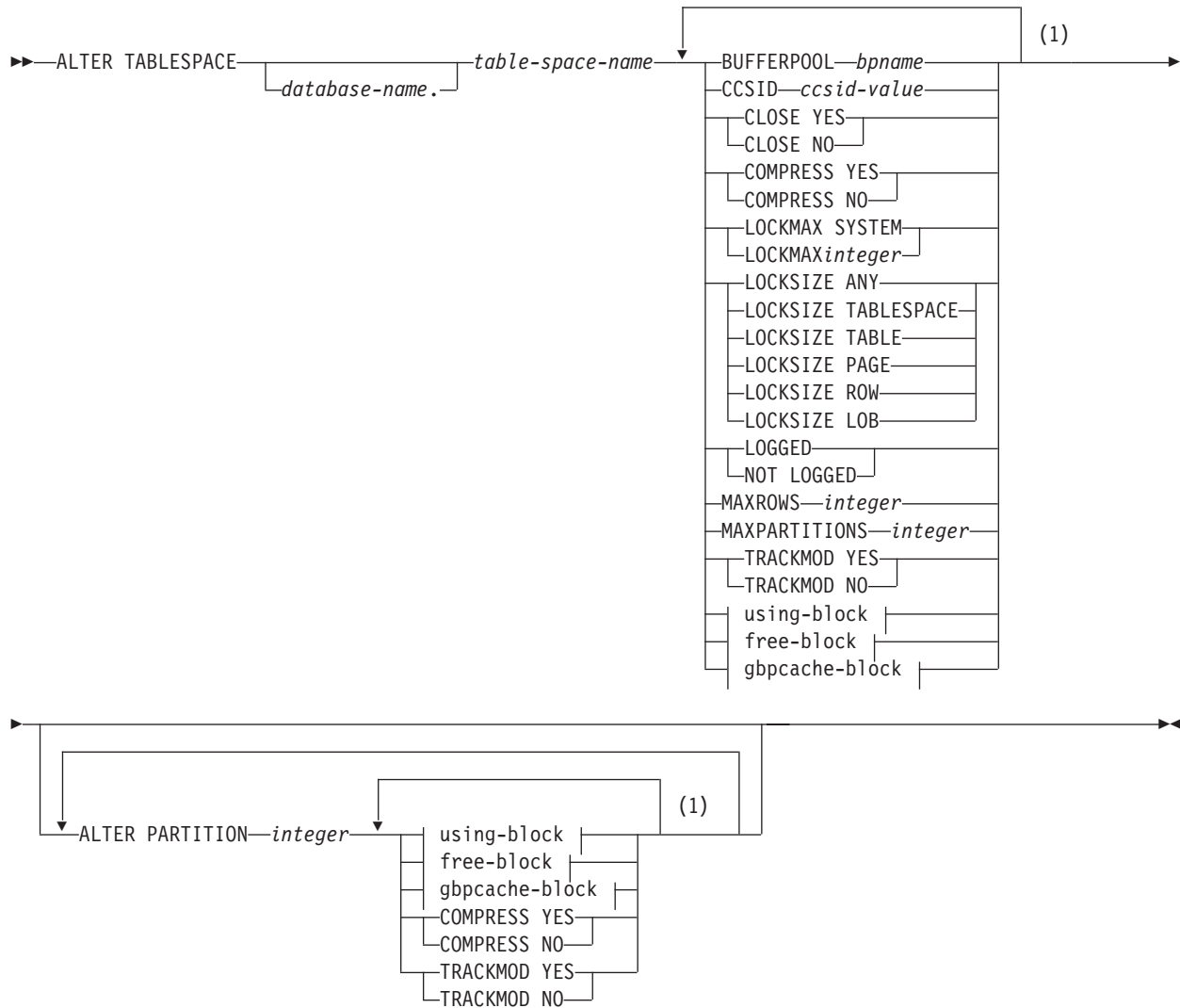
- Ownership of the table space
- DBADM authority for its database
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If BUFFERPOOL or USING STOGROUP is specified, additional privileges might be required, as explained in the description of those clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID and role of the process.

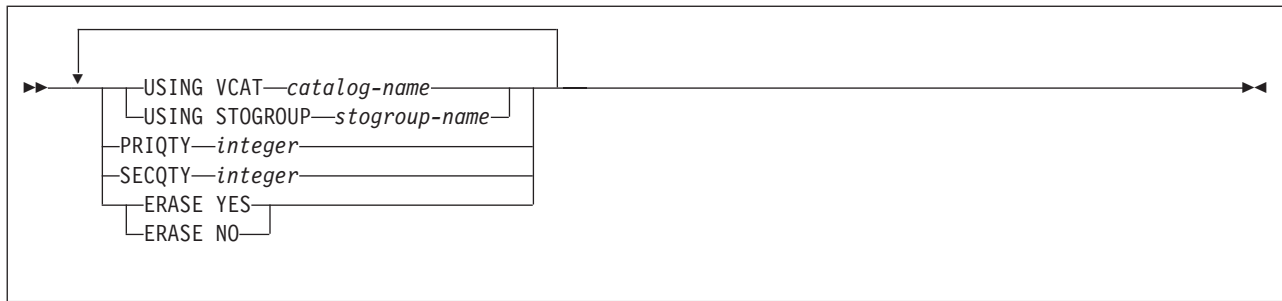
Syntax



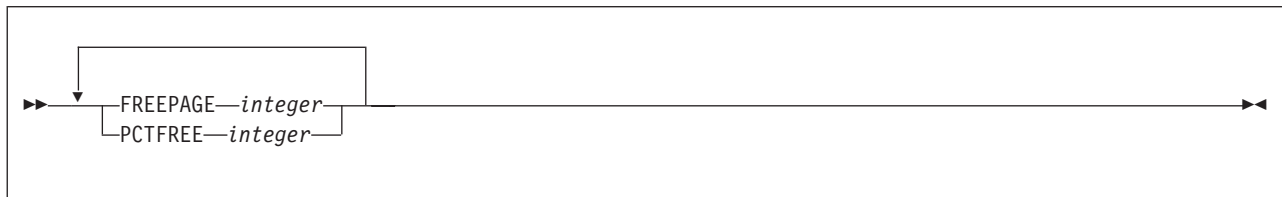
Notes:

- 1 The same clause must not be specified more than once in a single ALTER TABLESPACE statement. For example, if TRACKMOD YES is specified at the table space level, it must not be specified after ALTER PARTITION.

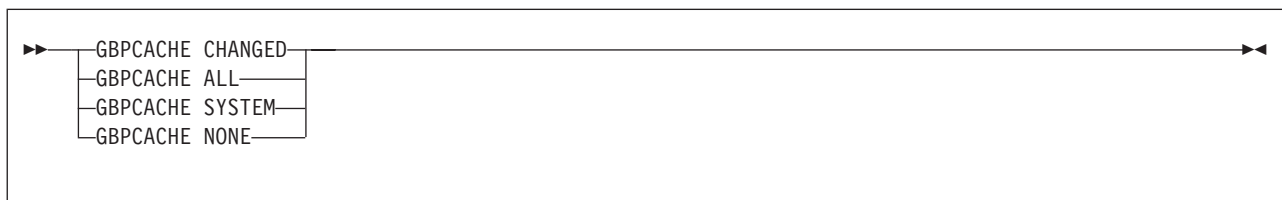
using-block:



free-block:



gbpcache-block:



Description

database-name.table-space-name

Identifies the table space that is to be altered. The name must identify a table space that exists at the current server. Omission of *database-name* is an implicit specification of DSNDB04.

If you identify a partitioned table space, you can use the PARTITION clause.

BUFFERPOOL *bpname*

Identifies the buffer pool that is to be used for the table space. *bpname* must identify an activated buffer pool with the same page size as the table space.

The privilege set must include SYSADM or SYSCTRL authority or the USE privilege for the buffer pool.

The change to the description of the table space takes effect the next time the data sets of the table space are opened. The data sets can be closed and reopened by using a STOP DATABASE command to stop the table space followed by a START DATABASE command to start the table space.

In a data sharing environment, if you specify BUFFERPOOL, the table space must be in the stopped state when the ALTER TABLESPACE statement is executed.

CCSID *ccsid-value*

Identifies the CCSID value to be used for the table space. *ccsid-value* must identify a CCSID value that is compatible with the current value of the CCSID

for the table space. See “Notes” on page 699 for a list that shows the CCSID to which a given CCSID can be changed and details about changing it.

Do not specify CCSID for a LOB table space, a table space that is implicitly created for an XML column, or a table space in a work file database.

The CCSID of a table space cannot be changed if the table space contains any table that has an index that contains expressions.

CLOSE

When the limit on the number of open data sets is reached, specifies the priority in which data sets are closed.

YES

Eligible for closing before CLOSE NO data sets. This is the default unless the table space is in a work file database.

NO Eligible for closing after all eligible CLOSE YES data sets are closed.

For a table space in a work file database, DB2 uses CLOSE NO regardless of the value specified

COMPRESS

Specifies whether data compression applies to the rows of the table space or partition. Do not specify COMPRESS for a LOB table space or a table space in a work file database.

YES

Specifies data compression. The rows are not compressed until the LOAD or REORG utility is run on the table in the table space or partition.

NO Specifies no data compression. Inserted rows will not be compressed. Updated rows will be decompressed. The dictionary used for compression will be erased when the LOAD REPLACE, LOAD RESUME NO, or REORG utility is run. See *DB2 Performance Monitoring and Tuning Guide* for more information about the dictionary and data compression.

LOCKMAX

Specifies the maximum number of page, row, or LOB locks an application process can hold simultaneously in the table space. If a program requests more than that number, locks are escalated. The page, row, or LOB locks are released and the intent lock on the table space or segmented table is promoted to S or X mode. If you specify LOCKMAX a for table space in a work file database, DB2 ignores the value because these types of locks are not used.

For an application that uses Sysplex query parallelism, a lock count is maintained on each member.

integer

Specifies the number of locks allowed before escalating, in the range 0 to 2 147 483 647.

Zero (0) indicates that the number of locks on the table or table space are not counted and escalation does not occur.

SYSTEM

Indicates that the value of field LOCKS PER TABLE(SPACE) on installation panel DSNTIPJ specifies the maximum number of page, row, or LOB locks a program can hold simultaneously in the table or table space.

If you change LOCKSIZE and omit LOCKMAX, the following results occur:

LOCKSIZE	Resultant LOCKMAX
TABLESPACE or TABLE	0
PAGE, ROW, or LOB	Unchanged
ANY	SYSTEM

If the lock size is TABLESPACE or TABLE, LOCKMAX must be omitted, or its operand must be 0.

LOCKSIZE

Specifies the size of locks used within the table space and, in some cases, also the threshold at which lock escalation occurs. Do not specify LOCKSIZE for a table space in a work file database.

ANY

Specifies that DB2 can use any lock size. Currently, DB2 never chooses row locks, but reserves the right to do so.

In most cases, DB2 uses LOCKSIZE PAGE LOCKMAX SYSTEM for non-LOB table spaces and LOCKSIZE LOB LOCKMAX SYSTEM for LOB table spaces. However, when the number of locks acquired for the table space exceeds the maximum number of locks allowed for a table space (an installation parameter), the page or LOB locks are released and locking is set at the next higher level. If the table space is segmented, the next higher level is the table. If the table space is not segmented, the next higher level is the table space.

TABLESPACE

Specifies table space locks.

TABLE

Specifies table locks. Use TABLE only for a segmented table space. Do not use TABLE for a universal table space.

PAGE

Specifies page locks. Do not use PAGE for a LOB table space.

ROW

Specifies row locks. Do not use ROW for a LOB table space.

LOB

Specifies LOB locks. Use LOB only for a LOB table space.

Let S denote an SQL statement that refers to a table in the table space:

- The LOCKSIZE change affects S if S is prepared and executed after the change. This includes dynamic statements and static statements that are not bound because of VALIDATE(RUN).
- If the size specified by the new LOCKSIZE is greater than the size of the old LOCKSIZE, the change affects S if S is a static statement that is executed after the change.

The hierarchy of lock sizes, starting with the largest, is as follows:

- table space lock
- table lock (only for segmented table spaces)
- page lock, row lock, and LOB lock (which are at the same level)

LOGGED or NOT LOGGED

Specifies whether changes that are made to the data in the specified table space are recorded in the log.

LOGGED

Specifies that changes that are made to the data in the specified table space are recorded in the log. This applies to all tables in the specified table space and to all indexes of those tables. Table spaces and indexes that are created for XML columns inherit the logging attribute from the associated base table space. Auxiliary indexes inherit the logging attribute from the associated base table space. This can affect the logging attribute of associated LOB table spaces. See “Notes” on page 852 for more information.

If the base table space is in informational copy-pending status (meaning updates have been made to the table space) when you change from NOT LOGGED to LOGGED, the base table space is placed in copy-pending status. All indexes of tables in the table space are unchanged from their current state; that is, if an index is currently in informational copy-pending status, it will remain in information copy-pending status.

Specifying LOGGED for a LOB table space requires that the base table space also specifies the LOGGED parameter.

LOGGED cannot be specified for XML table spaces. The logging attribute of an XML table space is inherited from its base table space.

LOGGED cannot be specified for table spaces in DSNDB06 (the DB2 catalog) or in a work file database.

NOT LOGGED

Specifies that changes that are made to data in the specified table space are not recorded in the log. This applies to all tables in the specified table space and to all indexes of those tables. Table spaces and indexes that are created for XML columns inherit the logging attribute from the associated base table space. Auxiliary indexes inherit the logging attribute from the associated base table space. This parameter can affect the logging attribute of associated LOB table spaces. See “Notes” on page 852 for more information.

NOT LOGGED prevents undo and redo information from being recorded in the log for the base table space; however, control information for the specified base table space will continue to be recorded in the log. For a LOB table space, changes to system pages and to auxiliary indexes are logged.

NOT LOGGED is mutually exclusive with the DATA CAPTURE CHANGES parameter of CREATE TABLE and ALTER TABLE. NOT LOGGED will not be applied to the table space if any table in the table space specifies DATA CAPTURE CHANGES.

NOT LOGGED cannot be specified for XML table spaces.

NOT LOGGED cannot be specified for table spaces in the following databases:

- DSNDB06 (the DB2 catalog)
- a work file database

MAXROWS *integer*

Specifies the maximum number of rows that DB2 will consider placing on each data page. The integer can range from 1 through 255.

The change takes effect immediately for new rows added. However, the space class settings for some pages might be incorrect and could cause unproductive page visits. It is highly recommended to reorganize the table space after altering MAXROWS.

If you specify MAXROWS, the table space must be in the stopped state when the ALTER TABLESPACE statement is executed. Do not specify MAXROWS for a LOB table space, a table space that is implicitly created for an XML column, a table space in a work file database, or the DB2 catalog table spaces that are listed under “SQL statements allowed on the catalog” on page 1689.

MAXPARTITIONS *integer*

Specifies that the table space is partition-by-growth. *integer* specifies the maximum number of partitions to which the table space can grow. *integer* must be in the range of 1 to 4096, depending on the value that is in effect for DSSIZE and the page size of the table space, and must not be less than the current maximum number of partitions for the table space. See “CREATE TABLESPACE” on page 1128 for more information about how DSSIZE and the page size are related.

MAXPARTITIONS can only be specified for a partition-by-growth table space, including XML partition-by-growth table spaces.

TRACKMOD

Specifies whether DB2 tracks modified pages in the space map pages of the table space or partition. Do not specify TRACKMOD for a LOB table space or a table space in a work file database.

YES

DB2 tracks changed pages in the space map pages to improve the performance of incremental image copy. For data sharing, changing TRACKMOD to YES causes additional SCA (shared communication area) storage to be used until after the next full or incremental image copy is taken or until TRACKMOD is set back to NO.

NO DB2 does not track changed pages in the space map pages. It uses the LRSN value in each page to determine whether a page has been changed.

FREEPAGE *integer*

Specifies how often to leave a page of free space when the table space is loaded or reorganized. One free page is left after every *integer* pages; *integer* can range from 0 to 255. FREEPAGE 0 leaves no free pages. Do not specify FREEPAGE for a LOB table space, a table space that is implicitly created for an XML column, or a table space in a work file database.

If the table space is segmented, the number of pages left free must be less than the SEGSIZE value. If the number of pages to be left free is greater than or equal to the SEGSIZE value, then the number of pages is adjusted downward to one less than the SEGSIZE value.

This change to the description of the table space or partition has no effect until data in the table space or partition is loaded or reorganized. For XML table spaces, this change has no effect until data in the table space is reorganized.

PCTFREE *integer*

Specifies what percentage of each page to leave as free space when the table space is loaded or reorganized. The first record on each page is loaded without restriction. When additional records are loaded, at least *integer* percent of free space is left on each page. *integer* can range from 0 to 99. Do not specify PCTFREE for a LOB table space, a table space that is implicitly created for an XML column, or a table space in a work file database.

This change to the description of the table space or partition has no effect until data in the table space or partition is loaded or reorganized. For XML table spaces, this change has no effect until data in the table space is reorganized.

USING

Specifies whether a data set for the table space or partition is managed by the user or is managed by the DB2 system. If the table space is partitioned, USING applies to the data set for the partition that is identified in the PARTITION clause. If the table space is a partition-by-growth table space, USING can only be specified at the table space level. If the table space is not partitioned, USING applies to every data set that is eligible for the table space. (A nonpartitioned table space can have more than one data set if PRIQTY+118 × SECQTY is at least 2 gigabytes.)

If the USING clause is specified, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed. See Altering storage attributes to determine how and when changes take effect. Do not specify the USING clause if the table space is in a work file database.

VCAT *catalog-name*

Specifies a user-managed data set with a name that starts with *catalog-name*. The VCAT clause must not be specified if the table space is a partition-by-growth table space. You must specify the catalog name in the form of an SQL identifier. You must specify an alias²⁴ if the name of the integrated catalog facility catalog is longer than eight characters. When the new description of the table space is applied, the integrated catalog facility catalog must contain an entry for the data set that conforms to the DB2 naming conventions set forth in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

STOGROUP *stogroup-name*

Specifies a DB2-managed data set that resides on a volume of the identified storage group. *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group. When the new description of the table space is applied, the description of the storage group must include at least one volume serial number, each volume serial number must identify a volume that is accessible to z/OS for dynamic allocation of the data set, and all identified volumes must be of the same device type. Furthermore, the integrated catalog facility catalog used for the storage group must not contain an entry for the data set.

If you specify USING STOGROUP and the current data set for the table space or partition is managed by DB2:

- Omission of the PRIQTY clause is an implicit specification of the current PRIQTY value.
- Omission of the SECQTY clause is an implicit specification of the current SECQTY value.
- Omission of the ERASE clause is an implicit specification of the current ERASE rule.

24. The alias of an integrated catalog facility catalog

If you specify USING STOGROUP to convert from user-managed data sets to DB2-managed data sets:

- Omission of the PRIQTY clause is an implicit specification of the default value. For information on how DB2 determines the default value, see Rules for primary and secondary space allocation.
- Omission of the SECQTY clause is an implicit specification of the default value. For information on how DB2 determines the default value, see Rules for primary and secondary space allocation.
- Omission of the ERASE clause is an implicit specification of ERASE NO.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set of the table space or partition. This clause can be specified only if the data set is managed by DB2, and if one of the following is true:

- USING STOGROUP is specified.
- A USING clause is not specified.

If PRIQTY is specified (with a value other than -1), the primary space allocation is at least *n* kilobytes, where *n* is the value of *integer* with the following exceptions:

- For 4 KB page sizes, if *integer* is less than 12, *n* is 12.
- For 8 KB page sizes, if *integer* is less than 24, *n* is 24.
- For 16 KB page sizes, if *integer* is less than 48, *n* is 48.
- For 32 KB page sizes, if *integer* is less than 96, *n* is 96.
- For any page size, if *integer* is greater than 4194304, *n* is 4194304.

For LOB table spaces, the exceptions are:

- For 4 KB pages sizes, if *integer* is less than 200, *n* is 200.
- For 8 KB pages sizes, if *integer* is less than 400, *n* is 400.
- For 16 KB pages sizes, if *integer* is less than 800, *n* is 800.
- For 32 KB pages sizes, if *integer* is less than 1600, *n* is 1600.
- For any page size, if *integer* is greater than 4194304, *n* is 4194304.

If the DB2 subsystem is under DFP 1.5, the maximum value allowed for PRIQTY is 64GB (67108864 kilobytes). Otherwise, the existing maximum value of 4GB (4194304 kilobytes) applies.

If PRIQTY -1 is specified, DB2 uses a default value for the primary space allocation. For information on how DB2 determines the default value for primary space allocation, see Rules for primary and secondary space allocation.

If PRIQTY is omitted and USING STOGROUP is specified, the value of PRIQTY is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of USING STOGROUP.)

If you specify PRIQTY and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of *p* KB not less than *n*, where *p* is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. To more closely estimate the actual amount of storage, see the description of the DEFINE CLUSTER command in *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

At least one of the volumes of the identified storage group must have enough available space for the primary quantity. Otherwise, the primary space allocation will fail.

See Altering storage attributes to determine how and when changes to PRIQTY take effect.

SECQTY *integer*

Specifies the minimum secondary space allocation for a DB2-managed data set of the table space or partition. This clause can be specified only if the data set is managed by DB2, and if one of the following is true:

- USING STOGROUP is specified.
- A USING clause is not specified.

If SECQTY -1 is specified, DB2 uses a default value for the secondary space allocation.

If USING STOGROUP is specified and SECQTY is omitted, the value of SECQTY is its current value. (However, if the current data set is being changed from being user-managed to DB2-managed, the value is its default value. See the description of USING STOGROUP.)

For information on the actual value that is used for secondary space allocation, whether you specify a value or DB2 uses a default value, see Rules for primary and secondary space allocation.

If you specify SECQTY and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of p KB not less than n , where p is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. To more closely estimate the actual amount of storage, see the description of the DEFINE CLUSTER command for z/OS DFSMS Access Method Services for Catalogs.

See Altering storage attributes to determine how and when changes to SECQTY take effect.

ERASE

Indicates whether the DB2-managed data sets for the table space or partition are to be erased before they are deleted during the execution of a utility or an SQL statement that drops the table space.

NO Does not erase the data sets. Operations involving data set deletion will perform better than ERASE YES. However, the data is still accessible, though not through DB2.

YES

Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

This clause can be specified only if the data set is managed by DB2, and if one of the following is true:

- USING STOGROUP is specified.
- A USING clause is not specified.

If you specify ERASE, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed. If you specify ERASE for a partitioned table space, you must also specify the ALTER PARTITION clause. See Altering storage attributes to determine how and when changes take effect.

GBPCACHE

In a data sharing environment, specifies what pages of the table space or partition are written to the group buffer pool in a data sharing environment. In

a non-data-sharing environment, you can specify GBPCACHE for a table space other than one in a work file database, but it is ignored. Do not specify GBPCACHE for a table space in a work file database in either environment (data sharing or not). In addition, you cannot alter the GBPCACHE value of some DB2 catalog table spaces; for a list of these table spaces, see “SQL statements allowed on the catalog” on page 1689.

CHANGED

When there is inter-DB2 R/W interest on the table space or partition, updated pages are written to the group buffer pool. When there is no inter-DB2 R/W interest, the group buffer pool is not used. Inter-DB2 R/W interest exists when more than one member in the data sharing group has the table space or partition open, and at least one member has it open for update.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), CHANGED is ignored and no pages are cached to the group buffer pool.

ALL

Indicates that pages are to be cached in the group buffer pool as they are read in from DASD.

Exception: In the case of a single updating DB2 when no other DB2 subsystems have any interest in the page set, no pages are cached in the group buffer pool.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), ALL is ignored and no pages are cached to the group buffer pool.

SYSTEM

Indicates that only changed system pages within the LOB table space are to be cached to the group buffer pool. A system page is a space map page or any other page that does not contain actual data values.

SYSTEM is the default for a LOB table space. Use SYSTEM only for a LOB table space.

NONE

Indicates that no pages are to be cached to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If you specify NONE, the table space or partition must not be in recover pending status when the ALTER TABLESPACE statement is executed.

If you specify GBPCACHE in a data sharing environment, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed.

ALTER PARTITION *integer*

Specifies that the identified partition of the table space is to be changed. For a table space that has *n* partitions, you must specify an integer in the range 1 to *n*. You must not use this clause for a nonpartitioned table space, for a LOB table space, or a partition-by-growth table space. At least one of the following clauses must be specified:

- COMPRESS
- ERASE
- FREEPAGE
- GBPCACHE
- PCTFREE

- PRIQTY
- SECQTY
- TRACKMOD
- USING

The changes specified by these clauses affect only the identified partition.

Do not specify the following clauses for ALTER PARTITION for partitions of a table space that is implicitly created for an XML column.

- CCSID
- FREEPAGE
- MAXROWS
- PCTFREE

Notes

Running utilities: You cannot execute the ALTER TABLESPACE statement while a DB2 utility has control of the table space.

Altering more than one partition: To change FREEPAGE, PCTFREE, USING, PRIQTY, SECQTY, COMPRESS, ERASE, or GBPCACHE for more than one partition, you must use separate ALTER TABLESPACE statements.

Altering storage attributes: The USING, PRIQTY, SECQTY, and ERASE clauses define the storage attributes of the table space or partition. If you specify USING or ERASE when altering storage attributes, the table space or partition must be in the stopped state when the ALTER TABLESPACE statement is executed. You can use a STOP DATABASE...SPACENAM... command to stop the table space or partition.

If the catalog name changes, the changes take effect after you move the data and start the table space or partition using the START DATABASE...SPACENAM... command. The catalog name can be implicitly or explicitly changed by the ALTER TABLESPACE statement. The catalog name also changes when you move the data to a different device. See the procedures for moving data in *DB2 Administration Guide*.

Changes to the secondary space allocation (SECQTY) take effect the next time DB2 extends the data set; however, the new value is not reflected in the integrated catalog until you use the REORG, RECOVER, or LOAD REPLACE utility on the table space or partition. The changes to the other storage attributes take effect the next time the page set is reset. For a non-LOB table space, the page set is reset when you use the REORG, RECOVER, or LOAD REPLACE utilities on the table space or partition. For a LOB table space, the page set is reset when RECOVER is run on the LOB table space or LOAD REPLACE is run on its associated base table space. If there is not enough storage to satisfy the primary space allocation, a REORG might fail. If you change the primary space allocation parameters or erase rule, you can have the changes take effect earlier if you move the data before you start the table space or partition.

Recommended GBPCACHE setting for LOB table spaces: For LOB table spaces, use the GBPCACHE CHANGED option instead of the GBPCACHE SYSTEM option. Due to the usage patterns of LOBs, the use of GBPCACHE CHANGED can help avoid excessive and synchronous writes to disk and the group buffer pool.

Altering the logging attribute of a table space: If the logging attribute (specified with the LOGGED or NOT LOGGED parameter) of a table space is altered frequently, the size of SYSIBM.SYSCOPY might need to be increased.

The logging attribute of the table space cannot be altered if the table space has been updated in the same unit of recovery.

A full image copy of the table space should be taken:

- Before altering a table space to NOT LOGGED
- After altering a table space to LOGGED

If a table space has data changes after an image copy is taken (the table space is in informational COPY-pending state), and the table space is altered from NOT LOGGED to LOGGED, the table space is marked COPY-pending and a full image copy of the table space must be taken.

An XML table space with the LOGGED logging attribute has its logging attribute altered to NOT LOGGED when the logging attribute of the associated base table space is altered from LOGGED to NOT LOGGED. When this happens, the logging attribute of the XML table space is said to be *linked* to the logging attribute of the base table space. When the logging attribute of the base table space is altered back to LOGGED, all logging attributes that are linked for the associated XML table spaces are altered back to LOGGED, and all of these links are dissolved.

A LOB table space with the LOGGED logging attribute has its logging attribute altered to NOT LOGGED when the logging attribute of the associated base table space is altered from LOGGED to NOT LOGGED. When this happens, the logging attribute of the LOB table space is said to be *linked* to the logging attribute of the base table space. When the logging attribute of the base table space is altered back to LOGGED, all logging attributes that are linked for the associated LOB table spaces are altered back to LOGGED, and all of these links are dissolved.

You can dissolve the link between these logging attributes by altering the logging attribute of the LOB table space to NOT LOGGED, even though it has already been implicitly given this logging attribute. After such an alter, the logging attribute of the LOB table space is unaffected when the logging attribute of the base table is altered back to LOGGED. A LOB table space with the NOT LOGGED logging attribute does not have this attribute changed in any way if the logging attribute of the associated base table space is altered from LOGGED to NOT LOGGED. When altered in this way, the logging attributes of the LOB table space and the base table space are not linked. If the base table space is altered back to LOGGED, the logging attribute of any LOB table spaces that are not linked to the logging attribute of the base table space remain unchanged.

Altering table spaces for DB2 catalog tables: For details on altering options on catalog tables, see “SQL statements allowed on the catalog” on page 1689.

Invalidation of plans and packages: All of the plans and packages that refer to that table space are invalidated when the SBCS CCSID attribute of a table space is changed. When the SBCS CCSID attribute of a table space is altered, all the plans and packages that refer to that table space are marked invalid.

Alternative syntax and synonyms: For compatibility with previous releases of DB2, the following keywords are supported:

- You can specify the LOCKPART clause, but it has no effect. DB2 treats all partitioned table spaces as if they were defined as LOCKPART YES. LOCKPART YES specifies the use of selective partition locking. When all the conditions for selective partition locking are met, DB2 locks only the partitions that are accessed. When the conditions for selective partition locking are not met, DB2 locks every partition of the table space.
- When altering the partitions of a partitioned table space, the ALTER keyword that precedes PARTITION keyword is optional and if ALTER keyword is omitted, then you can specify PART as a synonym for PARTITION.
- You can specify LOG YES as a synonym for LOGGED and LOG NO as a synonym for NOT LOGGED.

Examples

Example 1: Alter table space DSN8S91D in database DSN8D91A. BP2 is the buffer pool associated with the table space. PAGE is the level at which locking is to take place.

```
ALTER TABLESPACE DSN8D91A.DSN8S91D
  BUFFERPOOL BP2
  LOCKSIZE PAGE;
```

Example 2: Alter table space DSN8S91E in database DSN8D91A. The table space is partitioned. Indicate that the data sets of the table space are not to be closed when there are no current users of the table space. Also, change all of the partitions so that DB2 will use a formula to determine any secondary space allocations, and change partition 1 to use a PCTFREE value of 20.

```
ALTER TABLESPACE DSN8D91A.DSN8S91E
  CLOSE NO
  SECQTY -1
  ALTER PARTITION 1 PCTFREE 20;
```

Example 3: The following statement changes the maximum number of partitions in a partition-by-growth table space:

```
ALTER TABLESPACE TS01DB.TS01TS
  MAXPARTITIONS 30;
```

ALTER TRUSTED CONTEXT

The ALTER TRUSTED CONTEXT statement modifies the definition of a trusted context at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

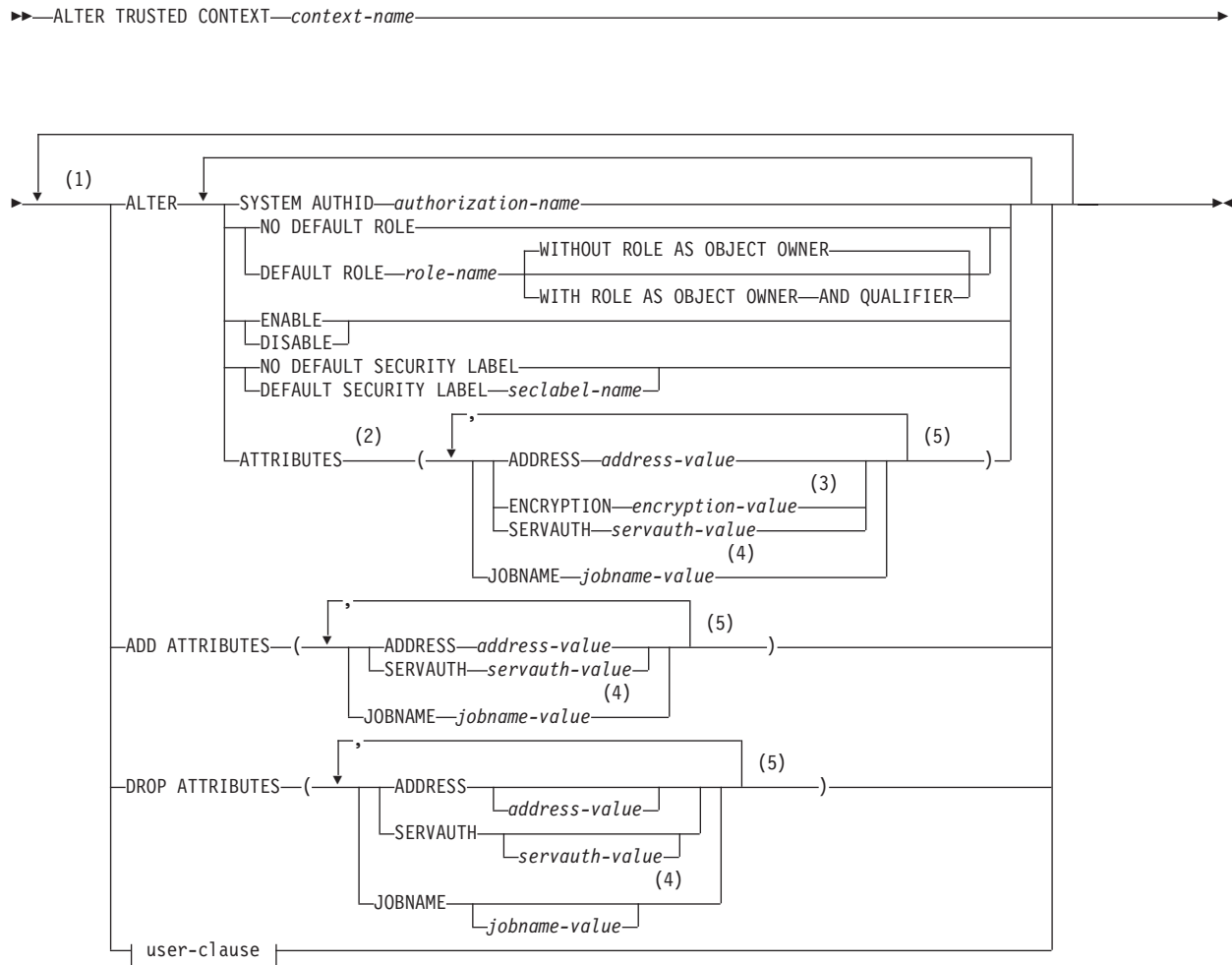
Authorization

The privilege set that is defined below must include SYSADM authority.

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the union of the set of privileges that are held by each authorization ID of the process. If the statement is run in a trusted context with a role, the privilege set is the union of the set of privileges that are held by the role that is associated with the primary authorization ID and the set of privileges that are held by each authorization ID of the process.

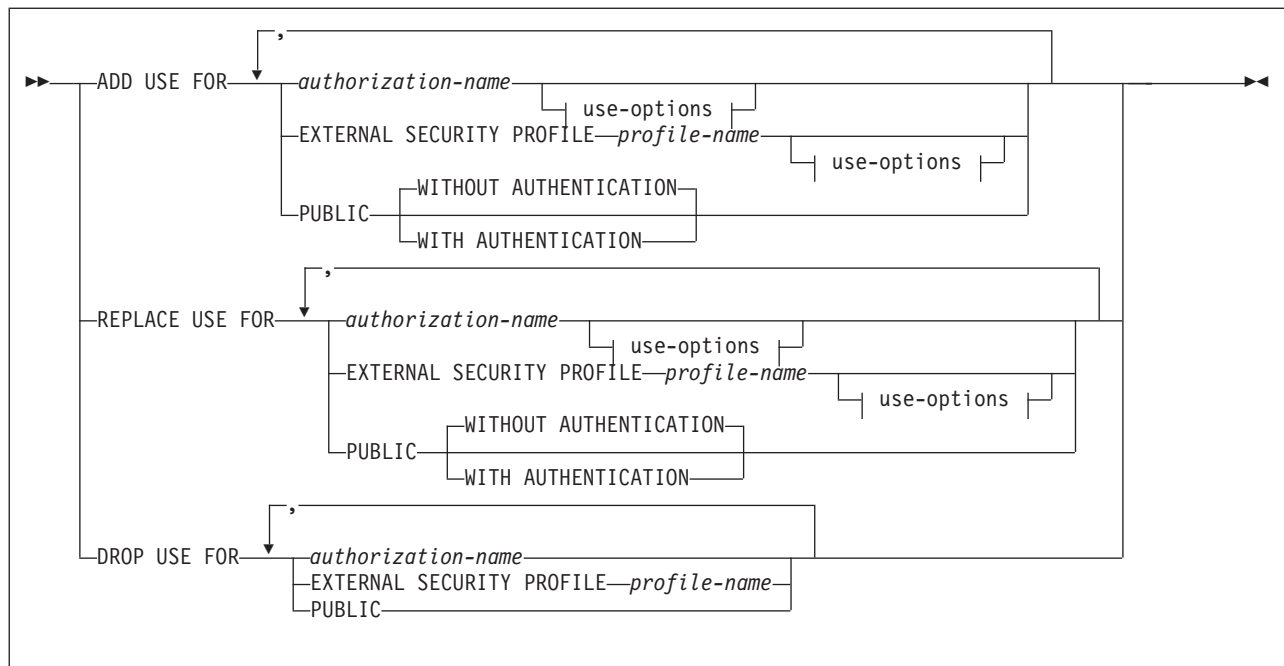
Syntax



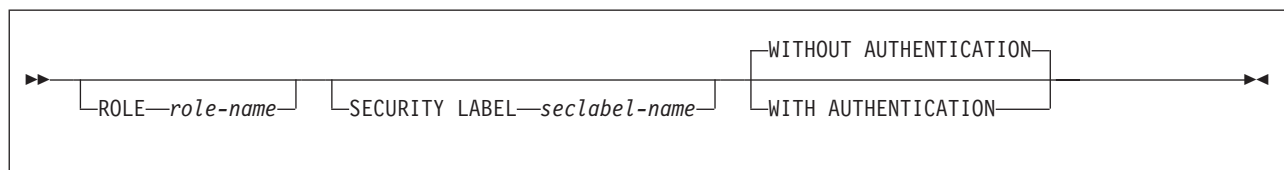
Notes:

- 1 These clauses can be specified in any order. Each clause must not be specified more than one time.
- 2 This clause and the clauses that follow can be specified in any order. Each clause must not be specified more than one time.
- 3 ENCRYPTION must not be specified more than one time.
- 4 JOBNAM must not be specified with ADDRESS, ENCRYPTION, or SERVAUTH.
- 5 Each pair of attribute name and corresponding value must be unique.

user-clause:



use-options:



Description

context-name

Identifies the trusted context to alter. *context-name* must refer to a trusted context that exists at the current server.

ALTER

Specifies that changes are to be made to the definition of an existing trusted context.

SYSTEM AUTHID *authorization-name*

Specifies that *authorization-name* is the system authorization ID for the trusted context. The system authorization ID is the primary authorization ID of the DB2 system that establishes the connection. For a remote connection, the authorization ID is derived from the system used ID that is provided by the external entity, such as a middleware server. For a local connection, the system authorization ID is derived depending on the sources, as specified in Table 92.

Table 92. System authorization ID for a local connection

Source of local connection	System authorization ID
Started task (RRSAF)	USER parameter on JOB statement or RACF USER.
TSO	TSO logon ID
BATCH	USER parameter on JOB statement

authorization-name must not be associated with an existing trusted context.

NO DEFAULT ROLE or DEFAULT ROLE *role-name*

Specifies whether a default role is associated with a trusted connection that is based on the specified trusted context. If a trusted connection for the specified context is active, the change goes into effect at the next connection reuse attempt or when a new connection is requested.

NO DEFAULT ROLE

Specifies that the trusted context does not have a default role. The authorization ID of the process is the owner of any object that is created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create that object.

DEFAULT ROLE *role-name*

Specifies that *role-name* is the role for the trusted context. *role-name* must identify a role that exists at the current server. This role is used with the user in a trusted connection that is based on the specified trusted context when the user does not have a user-specified role that is defined as part of the definition of this trusted context.

WITHOUT ROLE AS OBJECT OWNER or WITH ROLE AS OBJECT OWNER AND QUALIFIER

Specifies whether a role is used as the owner of objects that are created using a trusted connection that is based on the specified trusted context. If a trusted connection for the specified context is active, the change goes into effect at the next connection reuse attempt or when a new connection is requested.

WITHOUT ROLE AS OBJECT OWNER

Specifies that a role is not used as the owner of the objects that are created using a trusted connection that is based on the specified trusted context. The authorization ID of the process is the owner of any object that is created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create the object.

WITHOUT ROLE AS OBJECT OWNER is the default.

WITH ROLE AS OBJECT OWNER AND QUALIFIER

Specifies that the context assigned role is the owner of the objects that are created using a trusted connection that is based on this trusted context. That role must possess all of the privileges that are necessary to create the object. The context assigned role is the role that is defined for the user within this trusted context, if one is defined. Otherwise, the role is the default role that is associated with the trusted context. The role is also used as the grantor for any GRANT statements that are issued, and the revoker for any REVOKE statement that are issued using a trusted connection that is based on this trusted context.

AND QUALIFIER

Specifies that the *role-name* will be used as the default for the CURRENT SCHEMA special register. The *role-name* will also be included in the SQL PATH (in place of CURRENT SQLID).

When **WITH ROLE AS OBJECT OWNER AND QUALIFIER** is not specified, there is no change to the default of the CURRENT SCHEMA special register and SQL PATH.

DISABLE or ENABLE

Specifies whether the trusted context is in the enabled or disabled state.

DISABLE

Specifies that the trusted context is disabled. A trusted context that is disabled is not considered when a trusted connection is established.

ENABLE

Specifies that the trusted context is enabled.

NO DEFAULT SECURITY LABEL or DEFAULT SECURITY LABEL *seclabel-name*

Specifies whether a default security label is associated with a trusted connection that is based on this trusted context. If a trusted connection for the specified context is active, the change goes into effect at the next connection reuse attempt or when a new connection is requested.

NO DEFAULT SECURITY LABEL

Specifies that the trusted context does not have a default security label.

DEFAULT SECURITY LABEL *seclabel-name*

Specifies that *seclabel-name* is the default security label for the trusted context. *seclabel-name* is the security label that is used for multilevel security verification. *seclabel-name* must identify one of the RACF SECLABEL values that is defined for the SYSTEM AUTHID. This security label is used in a trusted connection that is based on the specified trusted context when the user does not have a specific security label defined as part of the definition of this trusted context. In this case, *seclabel-name* must also identify one of the RACF SECLABEL values that is defined for the user.

ALTER ATTRIBUTES or ADD ATTRIBUTES

Specifies a list of one or more connection trust attributes to change or add to the definition of a trusted context. The connection trust attributes are used to define the trusted context. If ALTER ATTRIBUTES is specified and the attribute is not currently part of the definition of the specified trusted context, an error is returned. Existing specifications for the specified attributes are changed to the new value if ALTER is specified. Attributes that are not specified retain the previously specified values.

ADDRESS *address-value*

Specifies the actual communication address that is used by the connection to communicate with the database manager. The protocol supported is only for TCP/IP. Previously specified ADDRESS values are removed when ALTER ATTRIBUTES is specified. The ADDRESS attribute can be specified multiple times, but each *address-value* must be unique.

When establishing a trusted connection, if multiple values are defined for the ADDRESS attribute for a trusted context, a candidate connection is considered to match this attribute if the address that is used by a connection matches any of the values that are defined for the ADDRESS attribute of the trusted context.

address-value specifies a string constant that contains the value that is associated with the ADDRESS trust attribute. *address-value* must be an IPv4 address, an IPv6 address, or a secure domain name with a length no greater than 254 bytes. No validation of *address-value* is done at the time the ALTER TRUSTED CONTEXT statement is processed. *address-value* must be left justified within the string constant.

- An IPv4 address is represented as a dotted decimal address. An example of an IPv4 address is 9.112.46.111.

- An IPv6 address is represented as a colon hexadecimal address. An example of an IPv6 address is 2001:0DB8:0000:0000:0008:0800:200C:417A. This address can also be express in a compressed form as 2001:DB8::8:800:200C:417A.
- A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is www.ibm.com. The gethostbyname socket call is used to resolve the domain name.

ENCRYPTION *encryption-value*

Specifies the minimum level of encryption of the data stream (network encryption) for the connection.

encryption-value specifies a string constant that contains the value that is associated with the ENCRYPTION trust attribute. *encryption-value* must be left justified within the string constant. ENCRYPTION must not be specified more than one time in the statement. *encryption-value* must be one of the following:

- NONE, which specifies that no specific level of encryption is required.
- LOW, which specifies that a minimum of light encryption is required. LOW corresponds to 64-bit DRDA encryption.
- HIGH, which specifies that strong encryption is required. HIGH corresponds to SSL encryption.

ENCRYPTION cannot be specified if ADD ATTRIBUTES is specified. See "CREATE TRUSTED CONTEXT" on page 1167 for more information about the ENCRYPTION attribute.

JOBNAME *jobname-value*

Specifies the z/OS job name or started task name (depending on the source of the address space) for local applications. Previously specified values for JOBNAME are removed when ALTER ATTRIBUTES is specified. The JOBNAME attribute can be specified multiple times, but each *jobname-value* must be unique.

jobname-value specifies a string constant that contains the value that is associated with the JOBNAME trust attribute. *jobname-value* is an EBCDIC 8 byte job name or started task name. *jobname-value* must be left justified within the string constant. The last character in the name can be a wildcard character (*) if the first character is an alphabetic character. If the job name ends with a wildcard, any job names that match the specified characters are considered for establishing the trusted connection.

The following table lists possible values for the job name depending on the source of the address space).

Table 93. Job name for local connection

Source of the address space	Job name
RRSAF	Job name or started task name
TSO	TSO logon ID
BATCH	Job name on JOB statement

SERVAUTH *servauth-value*

Specifies the name of a resource in the RACF SERVAUTH class. This resource is the network access security zone name that contains the IP address of the connection that is used to communicate with DB2.

Previously specified values for SERVAUTH are removed when ALTER

ATTRIBUTES is specified. The SERVAUTH attribute can be specified multiple times but each *servauth-value* must be unique.

servauth-value specifies a string constant that contains the value that is associated with the SERVAUTH trust attribute. *servauth-value* is an EBCDIC 64 byte RACF SERVAUTH CLASS resource name. *servauth-value* must be left justified in the string constant. No validation of *servauth-value* is done at the time the ALTER TRUSTED CONTEXT statement is processed.

DROP ATTRIBUTES

Specifies that one or more attributes are dropped from the definition of a trusted context. If the attribute is not currently specified as part of the definition of a trusted context, an error is returned. The specification of DROP ATTRIBUTES must not attempt to drop all of the existing attributes for a trusted context.

ADDRESS *address-value*

Specifies that the identified communication address is removed from the definition of the trusted context. *address-value* specifies a string constant that contains the value of an existing ADDRESS trust attribute.

JOBNAME *jobname-value*

Specifies that the identified job name is removed from the definition of the trusted context. *jobname-value* specifies a string constant that contains the value of an existing JOBNAME trust attribute.

SERVAUTH *servauth-value*

Specifies that the identified servauth that is removed from the definition of the trusted context. *servauth-value* specifies a string constant that contains the value of an existing SERVAUTH trust attribute.

ADD USE FOR

Specifies additional users who can use a trusted connection that is based on the specified trusted context.

authorization-name

Specifies that the trusted connection can be used by the specified *authorization-name*. This is the DB2 primary authorization ID. The *authorization-name* must not identify an authorization ID that is already defined to use the trusted context, and must not be specified more than one time in the ADD USE FOR clause.

ROLE *role-name*

Specifies that *role-name* is the role that is used when a trusted connection is used by the specified *authorization-name*. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the user overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*

Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by the specified *authorization-name*. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the user overrides any default security label that is associated with the trusted context.

EXTERNAL SECURITY PROFILE *profile-name*

Specifies that the trusted connection can be used by the DB2 primary authorization IDs that are permitted to use the specified *profile-name* in RACF. The *profile-name* must not already be defined to use the trusted

context and must not be specified more than one time in the **ADD USE FOR** clause. After you specify an external security profile, any user who is permitted access to the RACF profile can use the trusted context in addition to any users that are specified using the **ADD USE FOR** *authorization-name* clause.

ROLE *role-name*

Specifies that *role-name* is the role that is used when a trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the profile overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*

Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the profile overrides any default security label that is associated with the trusted context.

PUBLIC

Specifies that a trusted connection that is based on the specified trusted context can be used by any user. PUBLIC must not already be defined to use the trusted context and must not be specified more than one time in the **ADD USE FOR** clause.

All users that are using a trusted connection that is defined with PUBLIC use the privileges that are associated with the default role for the associated trusted context. If the default role is not defined for the trusted context, there is no role associated with the users that use a trusted connection that is based on the specified trusted context.

If the default security label for the trusted context is defined, all users that are using the trusted context must have the security label defined as one of the RACF SECLABEL values for the user. The default security label is used for multilevel security verification with all users that are using the trusted context.

The specifications for a user are determined in the following order of precedence:

- *authorization-name*
- EXTERNAL SECURITY PROFILE *profile-name*
- PUBLIC

For example, assume that a trusted context is defined with use for JOE WITH AUTHENTICATION, EXTERNAL SECURITY PROFILE SPROFILE WITHOUT AUTHENTICATION (with JOE and SAM permitted to use the RACF PROFILE SPROFILE), and PUBLIC WITH AUTHENTICATION. If the trusted connection is used by JOE, authentication is required. If the trusted connection is used by SAM, authentication is not required. However, if the trusted connection is used by SALLY, authentication is required.

REPLACE USE FOR

Specifies a change to the specified user or PUBLIC for who can use the trusted context.

| *authorization-name*

| Specifies the *authorization-name* that is changed for use of the trusted
| context. The trusted context must already be defined to allow use by
| *authorization-name*, and *authorization-name* must not be specified more than
| one time in the REPLACE USE FOR clause. The information that is
| associated with *authorization-name* is changed as indicated.

| **ROLE** *role-name*

| Specifies that *role-name* is the role that is used when a trusted
| connection is using the specified trusted context. The *role-name* must
| identify a role that exists at the current server. The role that is
| explicitly specified for the user overrides any default role that is
| associated with the trusted context.

| **SECURITY LABEL** *seclabel-name*

| Specifies that *seclabel-name* is the security label to use for multilevel
| security verification when the trusted connection is used by the
| specified *authorization-name*. The *seclabel-name* must be one of the RACF
| SECLABEL values that is defined for the user. The security label that is
| explicitly specified for the user overrides any default security label that
| is associated with the trusted context.

| **EXTERNAL SECURITY PROFILE** *profile-name*

| Specifies the *profile-name* to change attributes for use of the trusted
| connection. The trusted context must already be defined to allow the use of
| *profile-name*. *profile-name* must not be specified more than one time in the
| **REPLACE USE FOR** clause. The information that is associated with the
| profile name is changed as indicated.

| **ROLE** *role-name*

| Specifies that *role-name* is the role that is used when a trusted
| connection is used by any authorization ID that is permitted to use the
| specified *profile-name* in RACF. The role name must identify a role that
| exists at the current server. The role that is explicitly specified for the
| profile overrides any default role that is associated with the trusted
| context.

| **SECURITY LABEL** *seclabel-name*

| Specifies that *seclabel-name* is the security label to use for multilevel
| security verification when the trusted connection is used by any
| authorization ID that is permitted to use the specified *profile-name* in
| RACF. The *seclabel-name* must be one of the RACF SECLABEL values
| that is defined for the user. The security label that is explicitly specified
| for the user overrides any default security label that is associated with
| the trusted context.

| **PUBLIC**

| Specifies that the attributes for use of the trusted connection by PUBLIC
| are to be changed. PUBLIC must already be defined to use the trusted
| context, and PUBLIC must not be specified more than one time in the
| REPLACE USE FOR clause.

| All users that are using a trusted connection that is defined with PUBLIC
| use the privileges that are associated with the default role for the
| associated trusted context. If the default role is not defined for the trusted
| context, there is no role associated with the users that use a trusted
| connection that is based on the specified trusted context.

| If the default security label for the trusted context is defined, all users that
| are using the trusted context must have the security label defined as one of

the RACF SECLABEL values for the user. The default security label is used for multilevel security verification with all users that are using the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION

Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION

Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION

Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user.

DROP USE FOR

Specifies who can no longer use the trusted context. The users that are removed from the definition of the trusted context are the specified users (or PUBLIC) that are currently allowed to use the trusted context. If multiple users are specified to be dropped, and one or more of those users cannot be dropped, those users that can be dropped are dropped and a warning is returned. If none of the specified users can be removed from the definition of the trusted context, an error is returned.

authorization-name

Specifies the *authorization-name* that will no longer be able to use this trusted context.

EXTERNAL SECURITY PROFILE *profile-name*

Removes the ability for the specified *profile-name* to use the trusted context.

PUBLIC

Specifies that PUBLIC users will no longer be able to use this trusted context. The system authorization ID and individual authorization IDs that have been explicitly enabled can still use the trusted context.

Notes

Precedence for authorization-name and authentication requirements: If the *authorization-name* that is specified in the SYSTEM AUTHID clause is the same authorization name that is specified in the ADD or REPLACE USE FOR *authorization-name* clauses, the role or the security label that is specified for the *authorization-name* takes precedence over the default value and the value that is specified for the EXTERNAL SECURITY PROFILE *profile-name* (if one is specified). If the authorization name that is specified in the SYSTEM AUTHID clause is permitted to use one of the specified profile names and is not specified in ADD or REPLACE USE for *authorization-name*, the role or the security label that is specified for that *profile-name* takes precedence over the default value.

Authentication is required for SYSTEM AUTHID if the AUTHENTICATION clause is specified in the ADD or REPLACE USE FOR clauses, or if the subsystem parameter TCP/IP Already Verified is set to NO. For example, if *authorization-name* is the same as the authorization name that is specified in the SYSTEM AUTHID clause and the WITHOUT AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to NO, authentication is required for SYSTEM AUTHID when the remote trusted connection is established. If *authorization-name* is the SYSTEM AUTHID and the WITH AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to

YES, authentication is still required for SYSTEM AUTHID.

Order of precedence for users of a trusted connection: The specifications for a user are determined in the following order of precedence:

- *authorization-name*
- **EXTERNAL SECURITY PROFILE** *profile-name*
- **PUBLIC**

For example, assume that a trusted context is defined with use for JOE WITH AUTHENTICATION, EXTERNAL SECURITY PROFILE SPROFILE WITHOUT AUTHENTICATION, and PUBLIC WITH AUTHENTICATION. Users JOE and SAM are permitted to use the RACF PROFILE SPROFILE. If the trusted connection is used by JOE, authentication is required. If the trusted connection is used by SAM, authentication is not required. However, if user SALLY uses the trusted connection, authentication is required.

User-clause SYSTEM AUTHID considerations: If the *authorization-name* that is specified in the SYSTEM AUTHID clause is the same as the *authorization-name* that is specified in the user-clause *authorization-name*, the role or the security label that is specified for *authorization-name* takes precedence over the default value. The value that is specified for the *profile-name*, is permitted to use the profile. If the authorization name that is specified in the SYSTEM AUTHID clause is permitted to use one of the profile names and is not defined in *authorization-name*, the role or the security label that is specified for that *profile-name* takes precedence over the default value.

If authentication is required for SYSTEM AUTHID, either by specification of the AUTHENTICATION clause in the *user-clause* or by setting the value of the TCP/IP Already Verified subsystem parameter to NO, the authentication requirement takes precedence when establishing a remote trusted connection. For example, if *authorization-name* is the same as the authorization name that is specified for SYSTEM AUTHID and the WITHOUT AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to NO, an authentication token is required for SYSTEM AUTHID when the remote trusted connection is established. If *authorization-name* is the SYSTEM AUTHID and the WITH AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to YES, an authentication token is still required for SYSTEM AUTHID.

Order of operations: The order in which the clauses of the ALTER TRUSTED CONTEXT statement are applied are as follows:

- DROP ATTRIBUTES
- DROP USE FOR
- ALTER
- ADD ATTRIBUTES
- ADD USE FOR
- REPLACE USE FOR

Effect of changes on existing trusted connections: If trusted connections exist for the trusted context that is changed, the connections continue to use the unchanged definition of the trusted context until the connection is terminated or an attempt at reuse is made. If the trusted context is disabled while there are active trusted connections that are based on this trusted context, the connections continue to be

used until terminated or an attempt at reuse is made. If the trust attributes are changed, trusted connections that exist at the time that the trusted context is changed will continue to be used.

When changes to a trusted context take place: The changes to the definition of a trusted context take effect after the ALTER TRUSTED CONTEXT statement is committed. If the ALTER TRUSTED CONTEXT statement results in an error or is rolled back, the trusted context is not changed.

Role privileges: If no role is associated with the user or the trusted context, only the privileges that are associated with the user are applicable. This is the same as not using a trusted context.

Examples

Example 1: The following statement updates the default role of the trusted context CTX1:

```
ALTER TRUSTED CONTEXT CTX1
    ALTER DEFAULT ROLE CTXROLE2;
```

Example 2: The following statement changes the CTX3 trusted context to allow use for BILL, and it also puts the trusted context into the disabled state:

```
ALTER TRUSTED CONTEXT CTX3
    DISABLE
    ADD USE FOR BILL;
```

Example 3: The following statement changes the CTX4 trusted context to allow the previously defined user JOE to use the trusted context without authentication. The statement also adds use for PUBLIC with authentication and TOM with a role of SPLROLE:

```
ALTER TRUSTED CONTEXT CTX4
    REPLACE USE FOR JOE WITHOUT AUTHENTICATION
    ADD USE FOR PUBLIC WITH AUTHENTICATION,
    TOM ROLE SPLROLE;
```

Example 4: The following statement changes the REMOTECTX to use a different IPv4 address than it was originally defined to use. It also changes the encryption settings from NONE to LOW. After the ALTER statement is processed, the connection will be considered trusted only when it is established from 9.12.155.200 with low encryption. The connection will no longer be considered trusted if it is established from the previously defined addresses:

```
ALTER TRUSTED CONTEXT REMOTECTX
    ALTER ATTRIBUTES (ADDRESS '9.12.155.200',
        ENCRYPTION 'LOW');
```

ALTER VIEW

The ALTER VIEW statement regenerates a view using an existing view definition at the current server. ALTER VIEW is primarily used during DB2 migration or when DB2 maintenance is applied. To change a view definition (for example, to add additional columns), you must drop the view and create a new view using the CREATE VIEW statement.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- Ownership of the view
- SYSADM authority
- SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Syntax

►►—ALTER VIEW—*view-name*—REGENERATE—◀◀

Description

view-name

Identifies the view to be regenerated. The name must identify a view that exists at the current server.

REGENERATE

Specifies that the view is to be regenerated. The view definition in the catalog is used, and existing authorizations and dependent views are retained. The catalog is updated with the regenerated view definition. If the view cannot be successfully regenerated, an error is returned.

Examples

Check the catalog to find any views that were marked with view regeneration errors during catalog migration:

```
SELECT CREATOR,NAME FROM SYSIBM.SYSTABLES
WHERE TYPE = 'V' AND STATUS = 'R' AND TABLESTATUS = 'V';
```

Assume that the query returned MYVIEW as the name of a view with a regeneration error. Issue an ALTER VIEW statement to regenerate the view:

```
ALTER VIEW MYVIEW REGENERATE;
```

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a stored procedure.

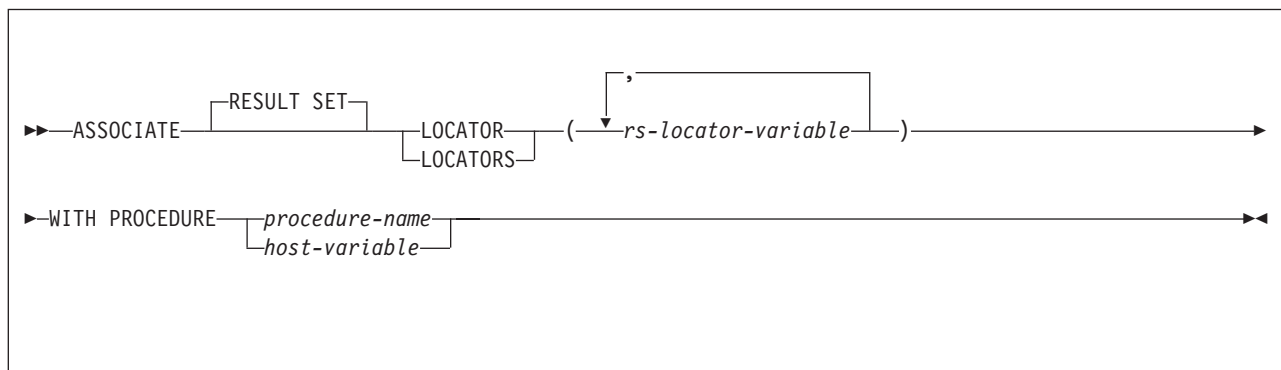
Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot be issued interactively.

Authorization

None required.

Syntax



Description

rs-locator-variable

Identifies a result set locator variable that has been declared according to the rules for declaring result set locator variables.

WITH PROCEDURE *procedure-name* **or** *host-variable*

Identifies the stored procedure that returned one or more result sets. When the ASSOCIATE LOCATORS statement is executed, the procedure name must identify a stored procedure that the requester has already invoked using the SQL CALL statement. The procedure name can be specified as a one-part, two-part, or three-part name. The procedure name in the ASSOCIATE LOCATORS statement must be specified the same way that it was specified on the CALL statement. For example, if a two-part procedure name was specified on the CALL statement, you must specify a two-part procedure name in the ASSOCIATE LOCATORS statement.

If a host variable is used to specify the name:

- It must be a character string variable with a length attribute that is not greater than 255.
- It must not be followed by an indicator variable.
- The value of the host variable is a specification that depends on the server. Regardless of the server, the specification must:
 - Be left justified within the host variable
 - Not contain embedded blanks
 - Be padded on the right with blanks if its length is less than that of the host variable

Notes

Assignment of locator values: If the ASSOCIATE LOCATORS statement specifies multiple locator variables, locator values are assigned to the locator variables in the order that the associated cursors are opened regardless of whether they are still open or not at runtime. Locator values are assigned to the locator variables in the same order that they would be placed in the SQLVAR entries in the SQLDA as a result of a DESCRIBE PROCEDURE statement.

Locator values are not provided for cursors that are closed when control is returned to the invoking application. If a cursor was closed and later opened again before returning to the invoking application, the most recently executed OPEN CURSOR statement for the cursor is used to determine the order in which the locator values are returned for the procedure result sets. For example, assume procedure P1 opens three cursors A, B, C, closes cursor B and then issues another OPEN CURSOR statement for cursor B before returning to the invoking application. The locator values assigned for the following ASSOCIATE LOCATORS statement will be in the order A, C, B:

```
ASSOCIATE RESULT SET LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P1;
-- assigns locators for result set cursors A, C, and B
```

More than one locator can be associated with a result set. You can issue multiple ASSOCIATE LOCATORS statements for the same stored procedure with different result set locator variables to associate multiple locators with each result set.

- If the number of result set locator variables specified in the ASSOCIATE LOCATORS statement is less than the number of result sets returned by the stored procedure, all locator variables specified in the statement are assigned a value, and a warning is issued. For example, assume procedure P1 exists and returns four result sets. Each of the following ASSOCIATE LOCATORS statement returns information on the first result set along with a warning that not enough locators were provided to obtain information about all the result sets.

```
CALL P1;
ASSOCIATE RESULT SET LOCATORS (:loc1) WITH PROCEDURE P1;
-- :loc1 is assigned a value for first result set, and a warning is returned
ASSOCIATE RESULT SET LOCATORS (:loc2) WITH PROCEDURE P1;
-- :loc2 is assigned a value for first result set, and a warning is returned
ASSOCIATE RESULT SET LOCATORS (:loc3) WITH PROCEDURE P1;
-- :loc3 is assigned a value for first result set, and a warning is returned
ASSOCIATE RESULT SET LOCATORS (:loc4) WITH PROCEDURE P1;
-- :loc4 is assigned a value for first result set, and a warning is returned
```
- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is greater than the number of locators returned by the stored procedure, the extra locator variables are assigned a value of 0.

Accessing result sets from multiple CALL statements: An application can access to result sets created by multiple CALL statements. The result sets can be created by different procedure or by the same procedure invoked multiple times.

- *Invoking different procedures:* Invoking different procedures with the same name can be done either explicitly by specifying the different collections or implicitly with the use of the PACKAGE PATH. For example, to identify the different collections explicitly, specify qualified names on the CALL statement. Although both procedures are named P2, they are different procedures. After the second CALL statement, result sets from both procedures are accessible to the application.


```
CALL X.P2;  
CALL Y.P2;
```

The collections for the two different procedures can also be determined implicitly from the PACKAGE PATH when unqualified procedure names are specified as part of the CALL statement. For example, assume that procedure P4 exists in collections X and Z. An application contains two CALL statements to invoke procedure P4. The references to procedure P4 in the CALL statements are unqualified. So, the PACKAGE PATH special register is used to resolve the procedure name. Procedure X.P4 is invoked for the first CALL statement and procedure Z.P4 is invoked by the second CALL statement. Following the second CALL statement, result sets from both procedures are accessible to the application.

```
SET CURRENT PACKAGE PATH = X, Y, Z;  
CALL P4;  
SET CURRENT PACKAGE PATH = PATH Z, Y, X;  
CALL P4;
```

- *Invoking the same procedure multiple times:* If the server and requester are both the same version of DB2, you can call a stored procedure multiple times within an application and at the same nesting level. Each call to the same stored procedure causes a unique instance of the stored procedure to run. If the stored procedure returns result sets, each instance of the stored procedure opens its own set of result set cursors. For more information on this situation, see Multiple calls to the same stored procedure.

When a procedure is invoked multiple times in an application and there is a need to process the result sets from the different instances at the same time, be sure to use the ASSOCIATE LOCATORS statement after each CALL statement to capture the locator values returned from each invocation of the procedure. For example, assume that procedure P exists in collection Z and that an application contains two CALL statements to invoke procedure P. The PACKAGE PATH is used to determine the collection for the procedure in the first CALL statement, and the collection is explicitly specified in the second CALL statement. Result sets from both procedures can be accessible to the application following both CALL statements if the locators for the result sets produced by the first CALL statement are captured with an ASSOCIATE LOCATOR statement before invoking the procedure the second time.

```
SET CURRENT PACKAGE PATH = X, Y, Z;  
CALL P3;  
ASSOCIATE LOCATORS ...  
CALL Z.P3;  
ASSOCIATE LOCATORS ...  
-- process the result sets using the locators
```

Using host variables: If the ASSOCIATE LOCATORS statement contains host variables, the following conditions apply:

- If the statement is executed statically, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.
- If the statement is executed dynamically, the contents of the host variables are assumed to be in the encoding scheme that is specified in the APPLICATION ENCODING bind option.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Use result set locator variables LOC1 and LOC2 to get the result set locator values for the two result sets returned by stored procedure P1. Assume that the stored procedure is called with a one-part name from current server SITE2.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL P1;
EXEC SQL ASSOCIATE RESULT SET LOCATORS (:LOC1, :LOC2)
      WITH PROCEDURE P1;
```

Example 2: Repeat the scenario in Example 1, but use a two-part name to specify an explicit schema name for the stored procedure to ensure that stored procedure P1 in schema MYSCHEMA is used.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL MYSCHEMA.P1;
EXEC SQL ASSOCIATE RESULT SET LOCATORS (:LOC1, :LOC2)
      WITH PROCEDURE MYSCHEMA.P1;
```

Example 3: Use result set locator variables LOC1 and LOC2 to get the result set locator values for the two result sets that are returned by the stored procedure named by host variable HV1. Assume that host variable HV1 contains the value SITE2.MYSCHEMA.P1 and the stored procedure is called with a three-part name.

```
EXEC SQL CALL SITE2.MYSCHEMA.P1;
EXEC SQL ASSOCIATE LOCATORS (:LOC1, :LOC2)
      WITH PROCEDURE :HV1;
```

The preceding example would be invalid if host variable HV1 had contained the value MYSCHEMA.P1, a two-part name. For the example to be valid with that two-part name in host variable HV1, the current server must be the same as the location name that is specified on the CALL statement as the following statements demonstrate. This is the only condition under which the names do not have to be specified the same way and a three-part name on the CALL statement can be used with a two-part name on the ASSOCIATE LOCATORS statement.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL SITE2.MYSCHEMA.P1;
EXEC SQL ASSOCIATE LOCATORS (:LOC1, :LOC2)
      WITH PROCEDURE :HV1;
```

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement marks the beginning of an SQL declare section. An SQL declare section contains declarations of host variables that are eligible to be used as host variables in SQL statements in a program.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax

►►—BEGIN DECLARE SECTION—◄◄

Description

The BEGIN DECLARE SECTION statement can be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. It is used to indicate the beginning of a host variable declaration section. A host variable section ends with an END DECLARE SECTION statement, described in “END DECLARE SECTION” on page 1273.

The following rules are enforced by the precompiler only if the host language is C or the STDSQL(YES) precompiler option is specified:

- A variable referred to in an SQL statement must be declared within a host variable declaration section of the source program in all host languages, other than Java and REXX. Furthermore, the declaration of each variable must appear before the first reference to the variable. Host variables are declared without the use of these statements in Java, and they are not declared at all in REXX.
- BEGIN DECLARE SECTION and END DECLARE SECTION statements must be paired and must not be nested.
- Host variable declaration sections can contain only host variable declarations, SQL INCLUDE statements that include host variable declarations, or DECLARE VARIABLE statements.

Notes

Host variable declaration sections are only required if the STDSQL(YES) option is specified or the host language is C. However, declare sections can be specified for any host language so that the source program can conform to IBM SQL. If declare sections are used, but not required, variables declared outside a declare section must not have the same name as variables declared within a declare section.

Example

```
EXEC SQL BEGIN DECLARE SECTION;  
  
    -- host variable declarations  
  
EXEC SQL END DECLARE SECTION;
```

CALL

The CALL statement invokes a stored procedure.

Invocation

This statement can be embedded in an application program. This statement can be executed interactively using the command line processor. Refer to *DB2 Application Programming and SQL Guide* for information about using the command line processor with the CALL statement. This statement can also be dynamically prepared, but only from an ODBC or CLI driver that supports dynamic CALL statements. IBM's ODBC and CLI drivers provide this capability.

Authorization

Invoking a stored procedure requires the EXECUTE privilege on the following:

- The stored procedure
You do not need the EXECUTE privilege on a stored procedure that was created prior to Version 6 of DB2 for z/OS.
- For external procedures (including external SQL procedures), additional authority is needed for the stored procedure package and most packages that run in the stored procedure.
The authorization that is required for which packages is explained in detail in *Authorization to execute packages under the stored procedure*.

Authorization to execute the stored procedure

The authorization ID or role that must have the EXECUTE privilege on the stored procedure depends on the form of the CALL statement:

- For static SQL programs that use the syntax *CALL procedure*, the owner of the plan or package that contains the CALL statement must have one of the following:
 - The EXECUTE privilege on the stored procedure
 - Ownership of the stored procedure
 - SYSADM authority
- For static SQL programs that use the syntax *CALL :host-variable*, the authorization ID or role of the plan or package that contains the CALL statement must have one of the following:
 - The EXECUTE privilege on the stored procedure
 - Ownership of the stored procedure
 - SYSADM authority

The DYNAMICRULES behavior for the plan or package that contains the CALL statement determines both the authorization ID or role and the privilege set that is held by that authorization ID or role:

Run behavior

The privilege set is the union of the set of privileges that are held by the SQL authorization ID and each authorization ID or role of the process.

Bind behavior

The privilege set is the privileges that are held by the primary authorization ID of the owner of the package or plan.

Define behavior

The privilege set is the privileges that are held by the authorization ID or role of the owner (definer) of the stored procedure or user-defined function that issued the CALL statement.

Invoke behavior

The privilege set is the privileges that are held by the authorization ID or role of the invoker of the stored procedure or user-defined function that issued the CALL statement. However, if the invoker is the primary authorization ID of the process or the CURRENT SQLID value, the privilege set is the union of the set of privileges that are held by each authorization ID or role.

For a list of the DYNAMICRULES values that specify run, bind, define, or invoke behavior, see Table 6 on page 64.

Authorization to execute packages under the stored procedure (including nested stored procedures)

The authorization that is required to run the stored procedure package and any packages that are used under the stored procedure (including nested stored procedures) apply to any form of the CALL statement as follows:

- **Stored procedure package:** One of the authorization IDs or roles that are defined in Set of authorization IDs must have at least one of the following privileges or authorities on the stored procedure package:
 - The EXECUTE privilege
 - Ownership of the package
 - PACKADM authority for the package's collection
 - SYSADM authority

A PKLIST entry is not required for the stored procedure package.

- **User-defined function packages and trigger packages:** If a stored procedure or any application under the stored procedure invokes a user-defined function, DB2 requires only the owner (the definer), and not the invoker of the user-defined function, to have EXECUTE authority on the user-defined function package. However, the authorization ID or role of the SQL statement that invokes the user-defined function must have EXECUTE authority on the function.

Similarly, if a trigger is used under a stored procedure, DB2 does not require EXECUTE authority on the trigger package; however, the authorization ID or role of the SQL statement that activates the trigger must have EXECUTE authority on the trigger.

For more information about the EXECUTE authority for user-defined functions, triggers, and user-defined function packages, see *DB2 Administration Guide*.

PKLIST entries are not required for any user-defined function packages or trigger packages that are used under the stored procedure.

- **Packages other than user-defined function, trigger, and stored procedure packages:** One of the authorization IDs or roles that is defined below under Set of authorization IDs must have at least one of the following privileges or authorities on any packages other than user-defined function and trigger packages that are used under the stored procedure:
 - The EXECUTE privilege
 - Ownership of the package
 - PACKADM authority for the package's collection

- SYSADM authority

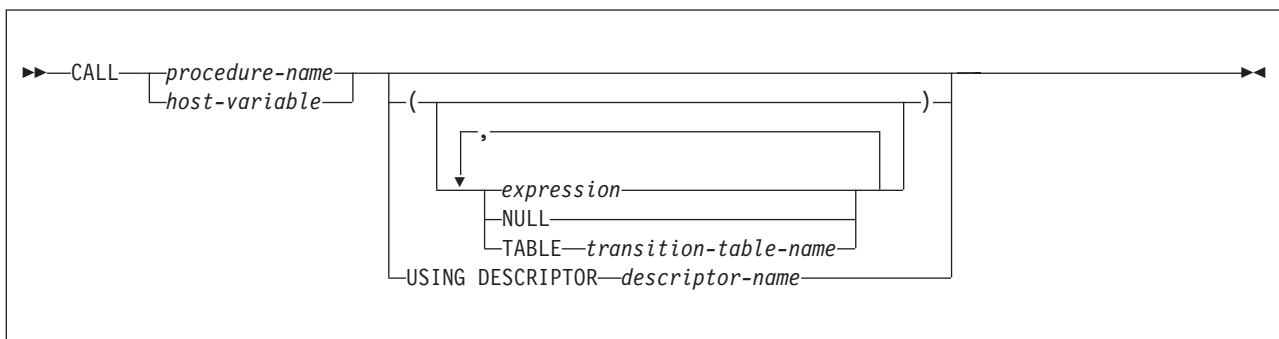
PKLIST entries are required for any of these packages that are used under the stored procedure.

For improved performance and simplicity, consider granting the EXECUTE ON PACKAGE privilege for the stored procedure package, and for any packages that run under the stored procedure, to the owner of the stored procedure.

Set of authorization IDs: DB2 checks the following authorization IDs, in the order in which they are listed, for the required authorization to execute the stored procedure package and any packages that run under the stored procedure other than user-defined function and trigger packages as described previously. Authorization checking ends after the first authorization ID that has EXECUTE ON PACKAGE privileges for the target package is found:

- The owner (the definer) of the stored procedure.
- The owner of the plan that contains the CALL statement that invokes the stored procedure if the calling application (a package or a DBRM that is bound directly to the plan) is local or if the calling application is distributed and DB2 for z/OS is both the requester and the server.
- The owner of the package that contains the CALL statement that invokes the stored procedure if the calling application is distributed and DB2 for z/OS is the server but not the requester, or if the calling application uses Recoverable Resources Management Services attachment facility (RRSAF) and has no plan.
- The authorization ID as determined by the value of the DYNAMICRULES bind option for the plan or package that contains the CALL statement if the CALL statement is in the form of CALL :host-variable.
 - If the calling application is bound with the DYNAMICRULES(RUN) option, DB2 checks either the authorization ID of the process at run time and its secondary authorization IDs or the single authorization ID that is determined by the other DYNAMICRULES bind option values.
 - If the calling application is bound with a value other than DYNAMICRULES(RUN), DB2 checks only a single authorization ID, even if that ID fails the EXECUTE ON PACKAGE authorization check.
 - If the calling application is a package and is bound with DYNAMICRULES(BIND), DB2 checks the authorization ID of the package owner. DB2 does not check the authorization ID of the plan owner.

Syntax



Description

procedure-name **or** *host-variable*

Identifies the procedure to call by the specified *procedure-name* or the procedure name contained in the *host-variable*. The identified procedure must exist at the current server.

If *procedure-name* specifies any of the three special characters that are alphabetic extenders for national languages, \$#@, specify the procedure name with a *host-variable*.

If a host variable is used:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 254.
- It must not be followed by an indicator variable.
- The value of the host variable is a specification that depends on the server. Regardless of the server, the specification must:
 - Be left justified within the host variable
 - Not contain embedded blanks
 - Be padded on the right with blanks if its length is less than that of the host variable

In addition, the specification can:

- Contain upper and lowercase characters. Lowercase characters are not folded to uppercase.
- Use a delimited identifier for any part of a three-part procedure name.

If the server is DB2 for z/OS, the specification must be a procedure name as defined above.

When the CALL statement is executed, the procedure name or specification must identify a stored procedure that exists at the server.

When the package that contains the CALL statement is bound, the stored procedure that is invoked must be created if VALIDATE(BIND) is specified. Although the stored procedure does not need to be created at bind time if VALIDATE(RUN) is specified, it must be created when the CALL statement is executed.

expression, **NULL**, **or** **TABLE** *transition-table-name*

Identifies a list of values to be passed as parameters to the stored procedure. If USING DESCRIPTOR is specified, each host variable described by the identified SQLDA is a parameter, or part of an expression that is a parameter, of the CALL statement. If host structures are not specified in the CALL statement, the *n*th parameter of the CALL statement corresponds to the *n*th parameter in the stored procedure, and the number of parameters in each must be the same. Otherwise, each reference to a host structure is replaced by a reference to each of the variables contained in that host structure, and the resulting number of parameters must be the same as the number of parameters defined for the stored procedure.

However, a character FOR BIT DATA parameter cannot be passed as input for a parameter that is not defined as character FOR BIT DATA. Likewise, a character argument that is not FOR BIT DATA cannot be passed as input for a parameter that is defined as character FOR BIT DATA.

Each parameter of a stored procedure is described at the server. In addition to attributes such as data type and length, the description of each parameter indicates how the stored procedure uses it:

- IN means as an input value
- OUT means as an output value
- INOUT means both as an input and an output value

When the CALL statement is executed, the value of each of its parameters is assigned to the corresponding parameter of the stored procedure. In cases where the parameters of the CALL statement are not an exact match to the data types of the parameters of the stored procedure, each parameter specified in the CALL statement is converted to the data type of the corresponding parameter of the stored procedure at execution. The conversion occurs according to the same rules as assignment to columns. For details on the rules used to assign parameters, see “Assignment and comparison” on page 102.

Conversion can occur when precision, scale, length, encoding scheme, or CCSID differ between the parameter specified in the CALL statement and the data type of the corresponding parameter of the stored procedure. Conversion might occur for a character string parameter specified in the CALL statement when the corresponding parameter of the stored procedure has a different encoding scheme or CCSID. For example, an error occurs when the CALL statement passes mixed data that actually contains DBCS characters as input to a parameter of the stored procedure that is declared with an SBCS subtype. Likewise, an error occurs when the stored procedure returns mixed data that actually contains DBCS characters in the parameter of the CALL statement that has an SBCS subtype.

Control is passed to the stored procedure according to the calling conventions of the host language. When execution of the stored procedure is complete, the value of each parameter of the stored procedure is assigned to the corresponding parameter of the CALL statement defined as OUT or INOUT.

expression

The parameter is the result of the specified expression, which is evaluated before the stored procedure is invoked.

If *expression* is a single host variable, the corresponding parameter of the procedure can be defined as IN, INOUT, or OUT. Otherwise, the corresponding parameter of the procedure must be defined as IN. In addition, the host variable can identify a structure. Any host variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables. A reference to a host structure is replaced by a reference to each of the variables contained in the host structure.

If the result of the expression can be the null value, either the description of the procedure must allow for null parameters or the corresponding parameter of the stored procedure must be defined as OUT.

The following additional rules apply depending on how the corresponding parameter was defined in the CREATE PROCEDURE statement for the procedure:

- IN *expression* can contain references to multiple host variables. In addition to the rules stated in “Expressions” on page 180 for *expression*, *expression* cannot include a column name, a scalar subselect, a file reference variable, an aggregate function, or a user-defined function that is sourced on an aggregate function.
- INOUT or OUT *expression* can only be a single host variable. *expression* cannot include a file reference variable.

NULL

The parameter is a null value. The corresponding parameter of the procedure must be defined as IN and the description of the procedure must allow for null parameters.

TABLE *transition-table-name*

The parameter is a transition table, and it is passed to the procedure as a table locator. You can use the CALL statement with the TABLE clause only within the definition of the triggered action of a trigger. The name of a transition table must be specified in the CALL statement if the corresponding parameter of the procedure was defined in the TABLE LIKE clause of the CREATE PROCEDURE statement. For information about creating a trigger, see “CREATE TRIGGER” on page 1151 and *DB2 Application Programming and SQL Guide*.

There is no effect on the transition table on the return from the procedure regardless of whether the parameter was defined as IN, INOUT, or OUT.

USING DESCRIPTOR *descriptor-name*

Identifies an SQLDA that contains a valid description of the host variables that are to be passed as parameters to the stored procedure. If the stored procedure has no parameters, an SQLDA is ignored.

Before the CALL statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA. This number must not be less than SQLD. This field is not part of the REXX SQLDA and therefore does not need to be set for REXX programs.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA. This number must be not be less than SQLN*44+16. This field is not part of the REXX SQLDA and therefore does not need to be set for REXX programs.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement. This number must be the same as the number of parameters of the stored procedure.
- SQLVAR occurrences to indicate the attributes of the variables.

There are additional considerations for setting the fields of the SQLDA when a variable that is passed as a parameter to the stored procedure has a LOB data type or is a LOB locator. For more information, see “SQL descriptor area (SQLDA)” on page 1656.

The SQL CALL statement ignores distinct type information in the SQLDA. Only the base SQL type information is used to process the input and output parameters described by the SQLDA.

See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C.

Notes

Procedure signatures: A procedure is identified by its schema, a procedure name, and its number of parameters. This is called a procedure signature, which must be unique within the database. DB2 for z/OS does not support overloaded procedure names (procedures with the same schema and procedure name, but with different numbers of parameters).

SQL path: A procedure can be invoked by referring to a qualified name (schema and procedure name), followed by an optional list of arguments that are enclosed in parentheses. A procedure can also be invoked without the schema name, which results in a choice of possible procedures in different schemas that have the same procedure name and same number of parameters. In this case, the SQL path is used to assist in procedure resolution. The SQL path is a list of schemas that is searched to identify a procedure with the same name and number of parameters as the procedure in the CALL statement. For CALL statements that explicitly specify a procedure name, the SQL path is specified by using the platform-specific bind option. For CALL *host-variable* statements, the SQL path is the value of the CURRENT PATH special register when the procedure is invoked.

Procedure resolution: Given a procedure invocation, the database manager must decide which of the possible procedures that has the same name to call.

A procedure name is a qualified or unqualified name. Each part of the name must be composed of SBCS characters:

- A fully qualified procedure name is a three-part name. The first part is an SQL identifier that contains the location name that identifies the DBMS at which the procedure is stored. The second part is an SQL identifier that contains the schema name of the stored procedure. The last part is an SQL identifier that contains the name of the stored procedure. A period must separate each of the parts. Any or all of the parts can be a delimited identifier.
- A two-part procedure name has one implicit qualifier. The implicit qualifier is the location name of the current server. The two parts identify the schema name and the name of the stored procedure. A period must separate the two parts.
- An unqualified procedure name is a one-part name with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second implicit qualifier depends on the server. If the server is DB2 for z/OS, the implicit qualifier is the schema name. DB2 uses the SQL path to determine the value of the schema name.
 - If the procedure name is specified as a string constant on the CALL statement (CALL *procedure-name*), the SQL path is the value of the PATH bind option that is associated with the calling package or plan.
 - If a host variable is specified for the procedure name on the CALL statement (CALL *host-variable*), the SQL path is the value of the CURRENT PATH special register.

DB2 searches the schema names in the SQL path from left to right until a stored procedure with the specified schema name is found in the DB2 catalog. When a matching *schema.procedure-name* is found, the search stops only if the following conditions are true:

- The user is authorized to call the stored procedure.
- The number of parameters in the definition of the stored procedure matches the number of parameters specified on the CALL statement.

If the list of schemas in the SQL path is exhausted before the procedure name is resolved, an error is returned.

When the procedure is resolved depends on how the procedure name is specified. For a CALL statement that specifies the procedure name using a host variable, procedure resolution occurs at runtime. For a CALL statement that contains the name of the procedure as an identifier, procedure resolution occurs when the CALL statement is bound.

Procedure resolution is done by the database manager using the following steps:

1. Find all procedures from the catalog where all of the following conditions are true:
 - For invocations where the schema name is specified (qualified references), the schema name and the procedure name match the invocation name.
For invocations where the schema name is not specified (unqualified references), the procedure name matches the invocation name, and the procedure has a schema name that matches one of the schemas in the SQL path.
 - The number of defined parameters matches the number of arguments that are specified in the invocation.
 - The invoker has the EXECUTE privilege on the procedure.
2. Of the candidate procedures that remain from step 1, choose the procedure whose schema is first in the SQL path. If no candidate procedures remain after step 1, an error is returned.
3. For CALL statements that use a host variable to specify the procedure name, the CURRENT ROUTINE VERSION special register can affect which version of the native SQL procedure is invoked. If the CURRENT ROUTINE VERSION special register is set, check if there is a version of the procedure with that version name. If not, choose the currently active version of the procedure.
For CALL statements that do not use a host variable to specify the procedure name, choose the currently active version of the procedure.

Version resolution: Normally, the currently active version of a native SQL procedure will be used on a CALL statement. However, if the CALL statement is a recursive call inside the body of the same stored procedure, and the original CALL statement uses a version that is different from the currently active version, the active version will not be used. The version from the original CALL statement will be used for any recursive CALL statements until the entire stored procedure finishes executing. This preserves the semantics of the version that is used by the original CALL statement. This includes the case where the recursive call is indirect. For example, assume that procedure SP1 call procedure SP2, which in turn recursively calls SP1. The second invocation of procedure SP1 will use the version of the procedure that is active at the time of the original CALL statement that invoked procedure SP1.

Since the currently active version can be used at the next CALL statement, it is possible that two or more versions of the same procedure can run at the same time. There could be different versions of an SQL procedure loaded by a given thread. For example, a CALL SP1 statement in an application will cause the currently active version, SP1_V1, to load and execute. After this CALL statement has completed, an ALTER PROCEDURE ALTER ACTIVE VERSION could execute and change the active version of the procedure SP1 to version SP1_V2. Subsequent CALL SP1 statements from the same thread will load the currently active version of the procedure, SP1_V2, and execute it.

Parameter assignments: When the CALL statement is executed, the value of each of its parameters is assigned with storage assignment rules to the corresponding parameter of the procedure. Control is passed to the procedure according to the calling conventions of the host language. When execution of the procedure is complete, the value of each parameter of the procedure is assigned with storage assignment rules to the corresponding parameter of the CALL statement defined as OUT or INOUT. If an error is returned by the procedure, OUT arguments are

undefined and INOUT arguments are unchanged. For details on the assignment rules, see “Assignment and comparison” on page 102.

Cursors and prepared statements in procedures: All cursors opened in the called procedure that are not result set cursors are closed and all statements prepared in the called procedure are destroyed when the procedure ends.

Result sets from procedures: Any cursors specified using the WITH RETURN clause that the procedure leaves open when it returns identifies a result set. In a procedure written in Java, all cursors are implicitly defined WITH RETURN.

Results sets are returned only when the procedure is called from CLI, JDBC, or SQLJ. If the procedure was invoked from CLI or Java, and more than one cursor is left open, the result sets can only be processed in the order in which the cursors were opened. Only unread rows are available to be fetched. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the procedure at the time the procedure is terminated, then rows 151 through 500 will be returned to the procedure.

Errors from procedures: A procedure can return errors or warnings using an SQLSTATE-like SQL statement. Applications should be aware of the possible SQLSTATES that can be expected when a procedure is invoked. The possible SQLSTATES depend on how the procedure is coded. Procedures might also return SQLSTATES such as those that begin with '38' or '39' if DB2 encounters problems executing the procedure. Applications should therefore be prepared to handle any error SQLSTATE that can result from issuing a CALL statement.

Improving performance: The capability of calling stored procedures is provided to improve the performance of DRDA distributed access (DB2 private protocol access is not supported). The capability is also useful for local operations. The server can be the local DB2. In which case, packages are still required.

All values of all parameters are passed from the requester to the server. To improve the performance of this operation, host variables that correspond to OUT parameters and have lengths of more than a few bytes should be set to null before the CALL statement is executed.

Using the CALL statement in a trigger: When a trigger issues a CALL statement to invoke a stored procedure, the parameters that are specified in the CALL statement cannot be host variables and the USING DESCRIPTOR clause cannot be specified.

Nesting CALL statements: A program that is executing as a stored procedure, a user-defined function, or a trigger can issue a CALL statement. When a stored procedure, user-defined function, or trigger calls a stored procedure, user-defined function, or trigger, the call is considered to be nested. Stored procedures, user-defined functions, and triggers can be nested up to 16 levels deep on a single system. Nesting can occur within a single DB2 subsystem or when a stored procedure or user-defined function is invoked at a remote server.

If a stored procedure returns any query result sets, the result sets are returned to the caller of the stored procedure. If the SQL CALL statement is nested, the result sets are visible only to the program that is at the previous nesting level. For example, Figure 18 on page 883 illustrates a scenario in which a client program calls stored procedure PROCA, which in turn calls stored procedure PROCB. Only PROCA can access any result sets that PROCB returns; the client program has no access to the query result sets. The number of query result sets that PROCB returns

does not count toward the maximum number of query results that PROCA can return.

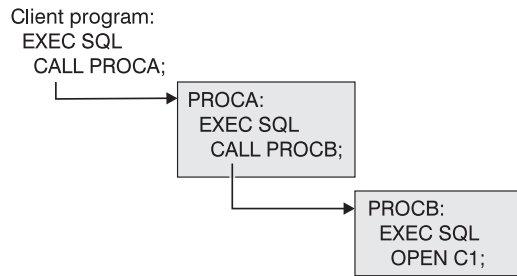


Figure 18. Nested CALL statements

Some stored procedures cannot be nested. A stored procedure, user-defined function, or trigger cannot call a stored procedure that is defined with the COMMIT ON RETURN attribute.

Multiple calls to the same stored procedure: If the server and requester are both Version 8 or later of DB2 for z/OS (running in new-function mode), you can call a stored procedure multiple times within an application and at the same nesting level. Each call to the same stored procedure causes a unique instance of the stored procedure to run. If the stored procedure returns result sets, each instance of the stored procedure opens its own set of result set cursors.

The application might receive a "resource unavailable message" if the CALL statement causes the values of the maximum number of active stored procedures or maximum number open cursors to be exceeded. The value of field MAX STORED PROCEDURES (on installation panel DSNTIPX) defines the maximum number of active stored procedures that are allowed per thread. The value of field MAX OPEN CURSORS (on installation panel DSNTIPX) defines the maximum number of open cursors (both result set cursors and regular cursors) that are allowed per thread.

If you make multiple calls to the same stored procedure within an application, be aware of the following considerations:

- A DESCRIBE PROCEDURE statement describes the last instance of the stored procedure.
- The ASSOCIATE LOCATORS statement works on the last instance of the stored procedure.
- The ALLOCATE CURSOR statement must specify a unique cursor name for a result set returned from an instance of the stored procedure. Otherwise, you will lose the data from the result sets that are returned from prior instances or calls to the stored procedure.

You should issue an ASSOCIATE LOCATORS statement (or DESCRIBE PROCEDURE statement) after each call to the stored procedure to get a unique locator value for each result set.

Using host variables: If the CALL statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Examples

Example 1: A PL/I application has been precompiled on DB2 ALPHA and a package was created at DB2 BETA with the BIND subcommand. A CREATE PROCEDURE statement was issued at BETA to define the procedure SUMARIZE, which allows nulls and has two parameters. The first parameter is defined as IN and the second parameter is defined as OUT. Some of the statements that the application that runs at DB2 ALPHA might use to call stored procedure SUMARIZE include:

```
EXEC SQL CONNECT TO BETA;  
V1 = 528671;  
IV = -1;  
EXEC SQL CALL SUMARIZE(:V1,:V2 INDICATOR :IV);
```

Example 2: Assume that stored procedure MYPROC exists and produces several result sets. An application might include statements like the following to access the result sets produced by MYPROC:

```
-- Invoke stored procedure MYPROC that returns several result sets  
EXEC SQL CALL MYPROC (...);  
-- Copy the locator values for the result sets into result set locator variables  
EXEC SQL ASSOCIATE RESULT SET LOCATORS (:RS1, :RS2, :RS3) WITH PROCEDURE MYPROC;  
-- Allocate cursors for the result set cursors  
EXEC SQL ALLOCATE CSR1 CURSOR FOR RESULT SET :RS1;  
EXEC SQL ALLOCATE CSR2 CURSOR FOR RESULT SET :RS2;  
EXEC SQL ALLOCATE CSR3 CURSOR FOR RESULT SET :RS3;  
-- Process data returned with the result set cursors  
DO WHILE (SQLCODE = 0);  
EXEC SQL FETCH CSR1 INTO .....  
END;  
EXEC SQL CLOSE CSR1;  
-- do similar processing with other result sets  
...
```

CLOSE

The CLOSE statement closes a cursor. If a temporary copy of a result table was created when the cursor was opened, that table is destroyed.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See “DECLARE CURSOR” on page 1191 for the authorization required to use a cursor.

Syntax

►►—CLOSE—*cursor-name*—◄◄

Description

cursor-name

Identifies the cursor to be closed. The cursor name must identify a declared cursor as explained in “DECLARE CURSOR” on page 1191. When the CLOSE statement is executed, the cursor must be in the open state.

Notes

Implicit cursor close: At the end of a unit of work, all open cursors declared without the WITH HOLD option that belong to an application process are implicitly closed.

Close cursors for performance: Explicitly closing cursors as soon as possible can improve performance.

Procedure considerations: Special rules apply to cursors within procedures that have not been closed before returning to the calling program. For more information, see “CALL” on page 874.

Allocated cursors: The cursor could have been allocated. See “ALLOCATE CURSOR” on page 696.

Example

A cursor is used to fetch one row at a time into the application program variables DNUM, DNAME, and MNUM. Finally, the cursor is closed. If the cursor is reopened, it is again located at the beginning of the rows to be fetched.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8910.DEPT
  WHERE ADMRDEPT = 'A00'
END-EXEC.
```

```
EXEC SQL OPEN C1 END-EXEC.

EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.

IF SQLCODE = 100
    PERFORM DATA-NOT-FOUND
ELSE
    PERFORM GET-REST-OF-DEPT
    UNTIL SQLCODE IS NOT EQUAL TO ZERO.

EXEC SQL CLOSE C1 END-EXEC.

GET-REST-OF-DEPT.
    EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.
```

COMMENT

The COMMENT statement adds or replaces comments in the descriptions of various objects in the DB2 catalog at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

For a comment on a table, view, alias, index, or column, the privilege set that is defined below must include at least one of the following:

- Ownership of the table, view, alias, or index
- DBADM authority for its database (tables and indexes only)
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

For a comment on a distinct type, stored procedure, trigger, or user-defined function, the privilege set that is defined below must include at least one of the following:

- Ownership of the distinct type, stored procedure, trigger, or user-defined function
- The ALTERIN privilege on the schema (for the addition of comments)
- SYSADM or SYSCTRL authority

For a comment on a package, the privilege set that is defined below must include at least one of the following:

- Ownership of the plan or package
- The BINDAGENT privilege granted from the package owner
- PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority

For a comment on a plan, the privilege set that is defined below must include at least one of the following:

- Ownership of the plan or package
- The BINDAGENT privilege granted from the plan owner
- SYSADM or SYSCTRL authority

For a comment on a role or a trusted context, the privilege set that is defined below must include at least one of the following:

- Ownership of the object
- SYSADM or SYSCTRL authority

For a comment on a sequence, the privilege set that is defined below must include at least one of the following:

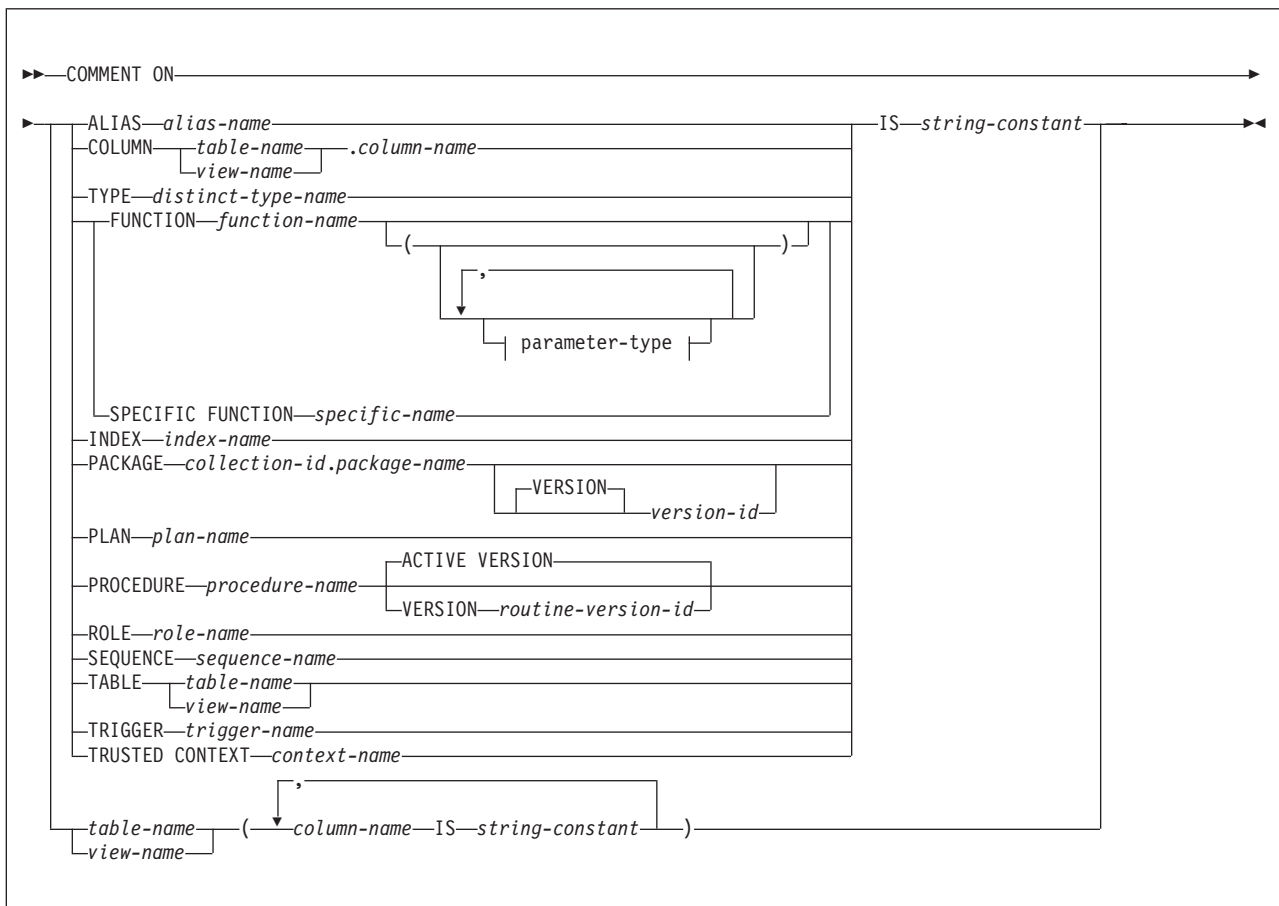
- Ownership of the sequence
- The ALTER privilege for the sequence

- The ALTERIN privilege on the schema
- SYSADM or SYSCTRL authority

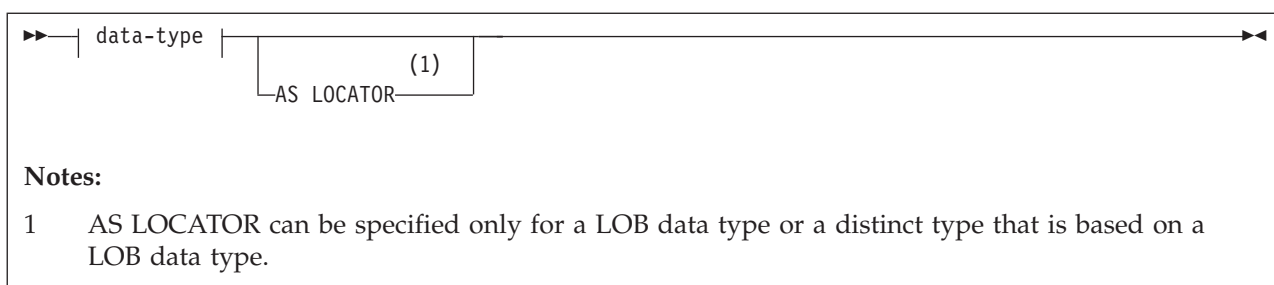
The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)

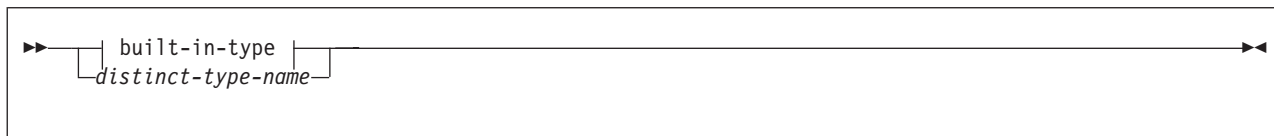
Syntax



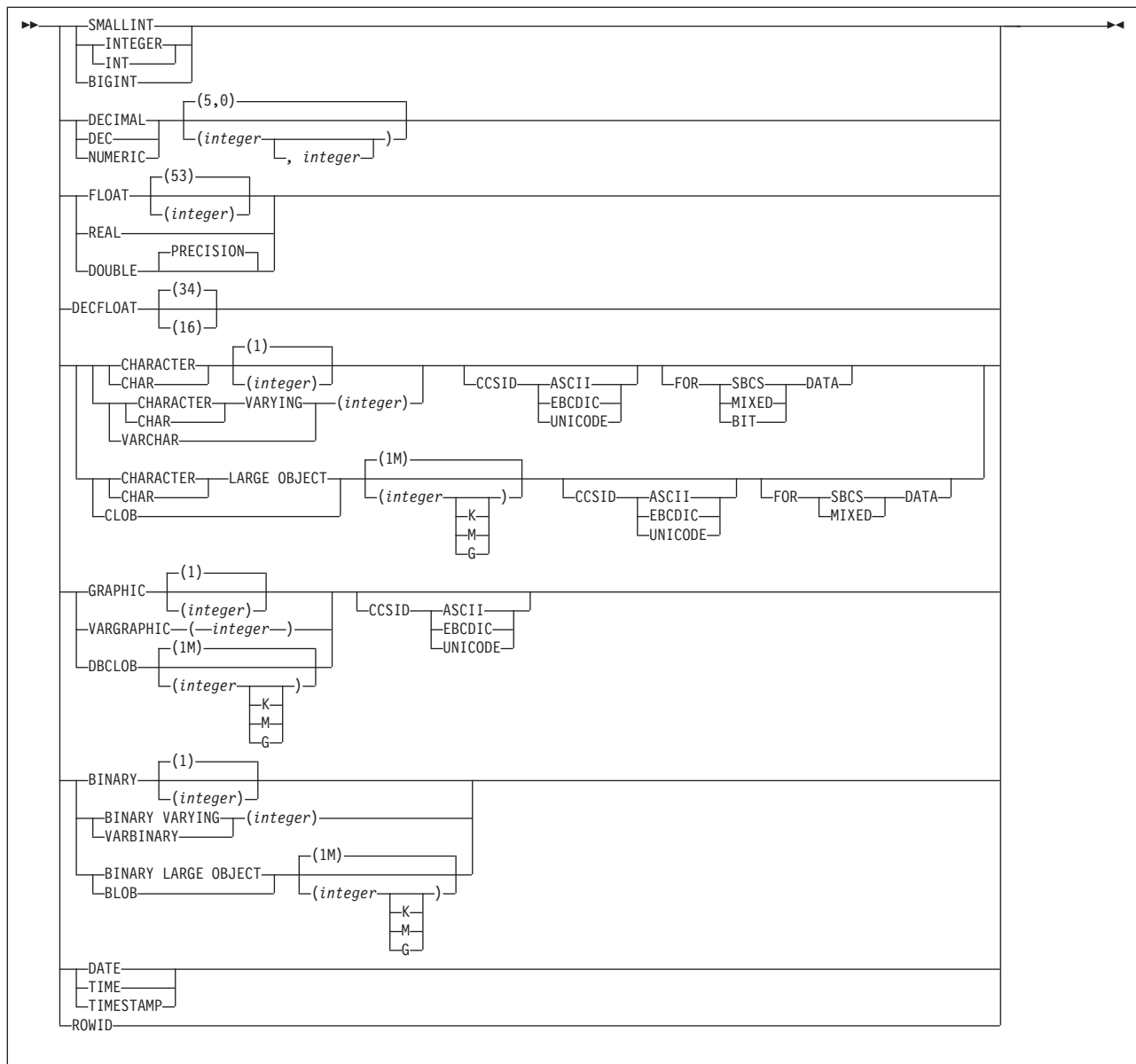
parameter-type



data-type



built-in-type



Description

ALIAS *alias-name*

Identifies the alias to which the comment applies. *alias-name* must identify an alias that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSTABLES catalog table for the row that describes the alias.

COLUMN *table-name.column-name* **or** *view-name.column-name*

Identifies the column to which the comment applies. The name must identify a column of a table or view that exists at the current server. The name must not identify a column of a declared temporary table. The comment is placed into the REMARKS column of the SYSIBM.SYSCOLUMNS catalog table, for the row that describes the column.

Do not use TABLE or COLUMN to comment on more than one column in a table or view. Give the table or view name and then, in parentheses, a list in the form:

column-name IS string-constant,
column-name IS string-constant,...

The column names must not be qualified, each name must identify a column of the specified table or view, and that table or view must exist at the current server.

FUNCTION or SPECIFIC FUNCTION

Identifies the function to which the comment applies. The function must exist at the current server, and it must be a function that was defined with the CREATE FUNCTION statement or a cast function that was generated by a CREATE TYPE statement. The comment is placed in the REMARKS column of the SYSIBM.SYSROUTINES catalog table for the row that describes the function.

The function can be identified by its name, function signature, or specific name. If the function was defined with a table parameter (the LIKE TABLE was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), you must identify the function with its function name, if it is unique, or with its specific name.

FUNCTION *function-name*

Identifies the function by its function name. There must be exactly one function with *function-name* in the schema. The function can have any number of input parameters. If the schema does not contain a function with *function-name*, or if the schema contains more than one function with this name, an error is returned.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types and the logical concatenation of the data types is used to identify the specific function instance to which the comment applies. Synonyms for data types are considered a match. The rules for function resolution (and the SQL path) are not used.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name()* is specified, the function that is identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parenthesis indicates that the data base manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the FOR *subtype* DATA clause or the CCSID clause is optional. Omission of either clause indicates that DB2 ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX *index-name*

Identifies the index to which the comment applies. *index-name* must identify an index that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSINDEXES catalog table for the row that describes the index.

PACKAGE *collection-id.package-name*

Identifies the package to which the comment applies. You must qualify the package name with a collection ID. *collection-id.package-name* must identify a package that exists at the current server. The name plus the implicitly or explicitly specified *version-id* must identify a package that exists at the current server. Omission of the *version-id* is an implicit specification of the null version.

The name must not identify a trigger package or a package that is associated with an SQL routine. Specify this clause to comment on a package that was created as the result of a BIND COPY command used to deploy a version of a native SQL procedure.

VERSION *version-id*

version-id is the version identifier that was assigned to the package's DBRM when the DBRM was created. If *version-id* is not specified, a null version is used as the version identifier.

Delimit the version identifier when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

For more on version identifiers, see the information on preparing an application program for execution in *DB2 Application Programming and SQL Guide*.

PLAN *plan-name*

Identifies the plan to which the comment applies. *plan-name* must identify a plan that exists at the current server.

PROCEDURE *procedure-name*

Identifies the procedure to which the comment applies. *procedure-name* must identify a procedure that exists at the current server.

ACTIVE VERSION

Specifies that the comment applies to the currently active version of the routine that is specified by *procedure-name*.

ACTIVE VERSION is the default.

VERSION *routine-version-id*

Specifies that the comment applies only to the version of the routine that is identified by *routine-version-id*. *routine-version-id* must identify a version of the specified routine that already exists at the current server. If *routine-version-id* is not specified, a null string is used as the version identifier.

ROLE *role-name*

Identifies the role to which the comment applies. *role-name* must identify a role that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSROLES catalog table for the row that describes the role.

SEQUENCE *sequence-name*

Identifies the sequence to which the comment applies.

sequence-name must identify a sequence that exists at the current server. *sequence-name* must not be the name of an internal sequence object that is used by DB2. The comment is placed in the REMARKS column of the SYSIBM.SYSSEQUENCES catalog table for the row that describes the sequence.

TABLE *table-name or view-name*

Identifies the table or view to which the comment applies. *table-name* or *view-name* must identify a table, auxiliary table, or view that exists at the current server. *table-name* must not identify a declared temporary table. The comment is placed in the REMARKS column of the SYSIBM.SYSTABLES catalog table for the row that describes the table or view.

TRIGGER *trigger-name*

Identifies the trigger to which the comment applies. *trigger-name* must identify a trigger that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSTRIGGERS catalog table for the row that describes the trigger.

TRUSTED CONTEXT *context-name*

Identifies the trusted context to which the comment applies. *context-name* must identify a trusted context that exists at the current server. The comment is

placed in the REMARKS column of the SYSIBM.SYSCONTEXT catalog table for the row that describes the trusted context.

TYPE *distinct-type-name*

Identifies the distinct type to which the comment applies. *distinct-type-name* must identify a distinct type that exists at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSDATATYPES catalog table for the row that describes the distinct type.

IS *string-constant*

Introduces the comment that you want to make. *string-constant* can be any SQL character string constant of up to 762 bytes.

Notes

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports DATA TYPE or DISTINCT TYPE as a synonym for TYPE.

Examples

Example 1: Enter a comment on table DSN8910.EMP.

```
COMMENT ON TABLE DSN8910.EMP
IS 'REFLECTS 1ST QTR 81 REORG';
```

Example 2: Enter a comment on view DSN8910.VDEPT.

```
COMMENT ON TABLE DSN8910.VDEPT
IS 'VIEW OF TABLE DSN8910.DEPT';
```

Example 3: Enter a comment on the DEPTNO column of table DSN8910.DEPT.

```
COMMENT ON COLUMN DSN8910.DEPT.DEPTNO
IS 'DEPARTMENT ID - UNIQUE';
```

Example 4: Enter comments on the two columns in table DSN8910.DEPT.

```
COMMENT ON DSN8910.DEPT
(MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
ADMNDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT');
```

Example 5: Assume that you are SMITH and that you created the distinct type DOCUMENT in schema SMITH. Enter comments on DOCUMENT.

```
COMMENT ON TYPE DOCUMENT
IS 'CONTAINS DATE, TABLE OF CONTENTS, BODY, INDEX, and GLOSSARY';
```

Example 6: Assume that you are SMITH and you know that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Enter comments on ATOMIC_WEIGHT.

```
COMMENT ON FUNCTION CHEM.ATOMIC_WEIGHT
IS 'TAKES ATOMIC NUMBER AND GIVES ATOMIC WEIGHT';
```

Example 7: Assume that you are SMITH and that you created the function CENTER in schema SMITH. Enter comments on CENTER, using the signature to uniquely identify the function instance.

```
COMMENT ON FUNCTION CENTER (INTEGER, FLOAT)
IS 'USES THE CHEBYCHEV METHOD';
```


Example 8: Assume that you are SMITH and that you created another function named CENTER in schema JOHNSON. You gave the function the specific name FOCUS97. Enter comments on CENTER, using the specific name to identify the function instance.

```
COMMENT ON SPECIFIC FUNCTION JOHNSON.FOCUS97
IS 'USES THE SQUARING TECHNIQUE';
```

Example 9: Assume that you are SMITH and that procedure OSMOSIS is in schema BIOLOGY. Enter comments on OSMOSIS. Your comments will apply to the currently active version of the procedure OSMOSIS.

```
COMMENT ON PROCEDURE BIOLOGY.OSMOSIS
IS 'CALCULATIONS THAT MODEL OSMOSIS';
```

Example 11: Assume that you are SMITH and that trigger BONUS is in your schema. Enter comments on BONUS.

```
COMMENT ON TRIGGER BONUS
IS 'LIMITS BONUSES TO 10% OF SALARY';
```

Example 12: Provide a comment for package MYPKG, which is in collection COLLIDA.

```
COMMENT ON COLLIDA.MYPKG
IS 'THIS IS MY PACKAGE';
```

Example 14: Provide a comment on role ROLE1:

```
COMMENT ON ROLE ROLE1
IS 'Role defined for trusted context, ctx1';
```

Example 15: Provide a comment on trusted context CTX1:

```
COMMENT ON TRUSTED CONTEXT CTX1
IS 'WEBSPPHERE SERVER';
```

COMMIT

The COMMIT statement ends the unit of recovery in which it is executed and a new unit of recovery is started for the process. The statement commits all changes made by SQL schema statements and SQL data change statements during the unit of work.

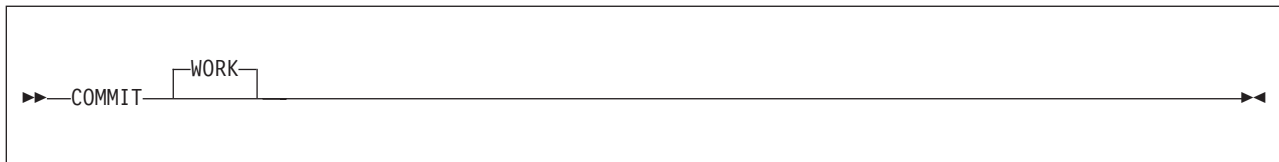
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. It cannot be used in the IMS or CICS environment.

Authorization

None required.

Syntax



Description

The COMMIT statement ends the unit of recovery in which it is executed and a new unit of recovery is started for the process. The statement commits all changes made by SQL schema statements and SQL data change statements during the unit of work. For more information see Chapter 5, “Statements,” on page 683.

Notes

Recommended coding practices: Code an explicit COMMIT or ROLLBACK statement at the end of an application process. Either an implicit commit or rollback operation will be performed at the end of an application process depending on the application environment. Thus, a portable application should explicitly execute a COMMIT or ROLLBACK statement before execution ends in those environments where explicit COMMIT or ROLLBACK is permitted.

Effect of COMMIT: All savepoints that are set within the unit of recovery are released, and all changes are committed for the following statements that are executed during the unit of recovery:

- ALTER
- COMMENT
- CREATE
- DELETE
- DROP
- EXPLAIN
- GRANT
- INSERT
- LABEL

- MERGE
- RENAME
- REVOKE
- UPDATE
- SELECT INTO with an SQL data change statement
- subselect with an SQL data change statement

SQL connections are ended when any of the following conditions apply:

- The connection is in the release pending status
- The connection is not in the release pending status but it is a remote connection and:
 - The DISCONNECT(AUTOMATIC) bind option is in effect, or
 - The DISCONNECT(CONDITIONAL) bind option is in effect and an open WITH HOLD cursor is not associated with the connection.

For existing connections, all LOB locators are disassociated, except for those locators for which a HOLD LOCATOR statement has been issued without a corresponding FREE LOCATOR statement. All open cursors that were declared without the WITH HOLD option are closed. All open cursors that were declared with the WITH HOLD option are preserved, along with any SELECT statements that were prepared for those cursors. All other prepared statements are destroyed unless dynamic caching is enabled for your system. In that case, all prepared SELECT and data change statements that are bound with DYNAMICKEEP(YES) are kept past the commit.

Prepared statements cannot be kept past a commit if any of the following conditions is true:

- SQL RELEASE has been issued for that site.
- Bind option DISCONNECT(AUTOMATIC) was used.
- Bind option DISCONNECT(CONDITIONAL) was used and there are no hold cursors for that site.

All implicitly acquired locks are released, except for the following locks:

- Locks that are required for the cursors that were not closed
- Table and table space locks when the RELEASE parameter on the bind command was not RELEASE(COMMIT)
- LOB locks and LOB table space locks that are required for held LOB locators

For an explanation of the duration of explicitly acquired locks, see *DB2 Performance Monitoring and Tuning Guide*.

All rows of every created temporary table of the application process are deleted with the exception that the rows of a created temporary table are not deleted if any program in the application process has an open WITH HOLD cursor that is dependent on that table. In addition, if RELEASE(COMMIT) is in effect, the logical work files for the created temporary tables whose rows are deleted are also deleted.

All rows of every declared temporary table of the application process are deleted with these exceptions:

- The rows of a declared temporary table that is defined with the ON COMMIT PRESERVE ROWS attribute are not deleted.

- The rows of a declared temporary table that is defined with the ON COMMIT DELETE ROWS attribute are not deleted if any program in the application process has an open WITH HOLD cursor that is dependent on that table.

Implicit commit operations: In all DB2 environments, the normal termination of a process is an implicit commit operation.

Restrictions on the use of COMMIT: The COMMIT statement cannot be used in the IMS or CICS environment. To cause a commit operation in these environments, SQL programs must use the call prescribed by their transaction manager. The effect of these commit operations on DB2 data is the same as that of the SQL COMMIT statement.

The COMMIT statement cannot be used in a stored procedure if the procedure is in the calling chain of a user-defined function or a trigger or DB2 is not the commit coordinator.

Effect of commit on special registers: Issuing a COMMIT statement may cause special registers to be re-initialized. Whether one of these special registers is affected by a commit depends on whether the special register has been explicitly set within the application process. For example, assume that the PATH special register has not been explicitly set with a SET PATH statement in the application process. After a commit, the value of PATH is re-initialized. For information on the initialization of PATH, which can take the current value of CURRENT SQLID into consideration, see “CURRENT PATH” on page 142.

Example

Commit all DB2 database changes made since the unit of recovery was started.

```
COMMIT WORK;
```

CONNECT

The CONNECT statement connects an application process to a database server. This server becomes the *current server* for the process. The CONNECT statement of DB2 for z/OS is equivalent to CONNECT (Type 2) in IBM DB2 SQL Reference for Cross-Platform Development.

Refer to “Distributed data” on page 31 for complete information about connections, the current server, commit processing, and distributed and remote units of work.

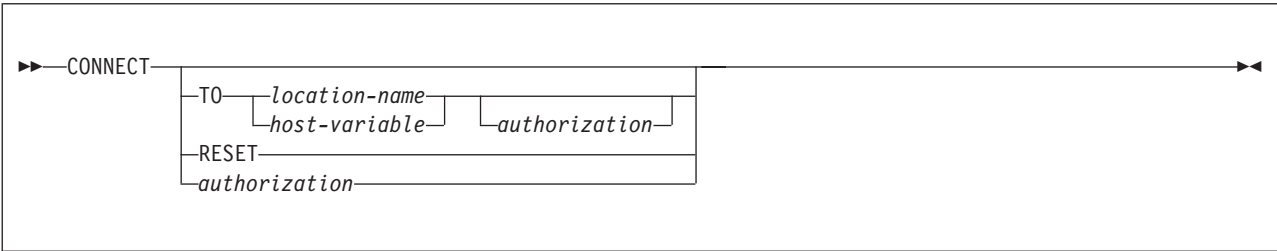
Invocation

This statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

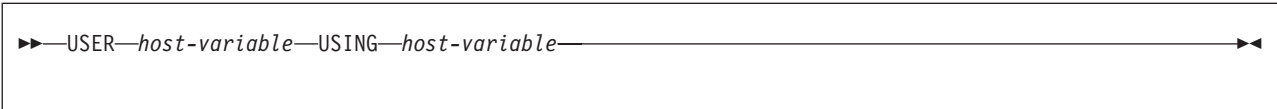
Authorization

The primary authorization ID of the process or the authorization ID that is specified in this statement must be authorized to connect to the specified server. The server performs the authorization check when the statement is executed, and determines the specific authorization that is required. See *DB2 Administration Guide* for further information.

Syntax



authorization:



Description

TO *location-name* **or** *host-variable*

Identifies the server by the specified location name or by the location name that is contained in the host variable. If a host variable is specified:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 16. (A C NUL-terminated character string can be up to 17 bytes long.)
- It must not be followed by an indicator variable.
- The location name must be left-justified within the host variable and must conform to the rules for forming an ordinary identifier.
- If the length of the location name is less than the length of the host variable, it must be padded on the right with blanks.

- It must not contain lowercase characters.
- If used with an SQL procedure language application, host variable must be a qualified SQL-variable name or a qualified SQL-parameter name.

When the CONNECT statement is executed:

- The location name must identify a server known to the local DB2 subsystem. Hence, the location name must be the location name of the local DB2 subsystem or it must appear in the LOCATION column of the SYSIBM.LOCATIONS table.
- The application process must not have an existing connection to the specified server, if the SQLRULES(STD) bind option is in effect.
- The application process must be in a connectable state, if the transaction is participating in a remote unit of work.

RESET

CONNECT RESET is equivalent to CONNECT TO *x* where *x* is the location name of the local DB2 subsystem.

- If the SQLRULES(DB2) bind option is in effect, CONNECT RESET establishes the local DB2 subsystem as the current SQL connection.
- If the SQLRULES(STD) bind option is in effect, CONNECT RESET establishes the local DB2 subsystem as the current SQL connection only if the connection does not exist.

authorization

Specifies an authorization ID and a password that is used to verify that the authorization ID is authorized to connect to the server. Authorization cannot be specified when the connection type is IMS or CICS for a connection to the local DB2 subsystem. An attempt to do so causes an SQL error.

USER *host-variable*

Identifies the authorization name to use when connecting to the server. The value of *host-variable* must satisfy the following rules:

- The value must be a CHAR or VARCHAR variable with a length attribute that is not greater than 128.
- The value must be left-justified within the host variable and must conform to the rules for forming an authorization name.
- The value must not be followed by an indicator variable.
- The value must be padded on the right with blanks if the length of the authorization name is less than the length of the host variable.

For a connection to the local DB2 subsystem, a user ID that is longer than 8 characters causes an SQL error.

USING *host-variable*

Identifies the password of the authorization name to use when connecting to the server. The value of *host-variable* must satisfy the following rules:

- The value must be a CHAR or VARCHAR variable with a length attribute that is not greater than 128.
- The value must be left-justified.
- The value must not include an indicator variable.
- The value must be padded on the right with blanks if the length of the password is less than the length of the host variable.
- The value must not contain lowercase characters.

For a connection to a DB2 subsystem, a password that is longer than 8 characters causes an SQL error.

CONNECT USER/USING is equivalent to CONNECT TO *x* USER/USING where *x* is the location name of the local DB2 subsystem (which has the semantic of CONNECT RESET).

CONNECT with no operand

This form of the CONNECT statement returns information about the current server in the SQLERRP field of the SQLCA. SQLERRP returns blanks if the application process is in the unconnected state.

Executing a CONNECT with no operand has no effect on connection states.

In a remote unit of work, this form of CONNECT does not require the application process to be in a connectable state.

Notes

Successful connection: With the exception of a CONNECT with no operand statement, if execution of the CONNECT statement is successful:

- One of the following scenarios takes place in a **distributed unit of work**:
 - If the location name does not identify a server to which the application process is already connected, an SQL connection to the server is created and placed in the current and held state. The previously current SQL connection, if any, is placed in the dormant state.
 - If the location name identifies a server to which the application process is already connected, the associated SQL connection is dormant, and the SQLRULES(DB2) option is in effect, the SQL connection is placed in the current state. The previously current SQL connection, if any, is placed in the dormant state.
 - If the location name identifies a server to which the application process is already connected, the associated SQL connection is current, and the SQLRULES(DB2) option is in effect, the states of all SQL connections of the application process are unchanged.
- The following actions occur in a **remote unit of work**:
 - The application process is connected to the specified server.
 - An existing SQL connection of the application process is ended. As a result, all cursors of that SQL connection are closed, all prepared statements of that connection are destroyed, and so on.
- The location name is placed in the CURRENT SERVER special register.
- When CONNECT is used to connect back to the local DB2 subsystem, the CURRENT SQLID special register is reinitialized if the USER/USING clause is specified.
- Information about the server is placed in the SQLERRP field of the SQLCA. If the server is a DB2 product, the information has the form *pppvrrm*, where:
 - *ppp* is:
 - ARI for DB2 Server for VSE & VM
 - DSN for DB2 for z/OS
 - QSQ for DB2 for i
 - SQL for DB2 for LUW
 - *vv* is a two-digit version identifier such as '09'.
 - *rr* is a two-digit release identifier such as '01'.
 - *m* is a one-digit maintenance level. Values 0, 1, 2, 3, and 4 are reserved for maintenance levels in compatibility and enabling-new-function mode. Values 5, 6, 7, 8, and 9 are for maintenance levels in new-function mode.

For example, if the server is Version 9 of DB2 for z/OS in new-function mode with the first level of maintenance, the value of SQLERRP is 'DSN09015'.

- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.

Tip: Use the GET DIAGNOSTICS statement to get detailed diagnostic information about the last SQL statement that was executed.

Unsuccessful connection: With the exception of a CONNECT with no operand statement, if execution of the CONNECT statement is unsuccessful:

- In a **distributed unit of work**, the connection state of the application process and the states of its SQL connections are unchanged unless the failure was because an authorization check failed. If this is the case, the connection is placed in the connectable and unconnected state.
- In a **remote unit of work**, the SQLERRP field of the SQLCA is set to the name of the DB2 requester module that detected the error.

If execution of the CONNECT statement is unsuccessful because the application process is not in the connectable state, the connection state of the application process is unchanged. If execution of the CONNECT statement is unsuccessful for any other reason, CURRENT SERVER is set to blanks and the application process is placed in the connectable and unconnected state.

Authorization: If the server is a DB2 subsystem, a user is authenticated in the following way:

- DB2 invokes RACF via the RACROUTE macro with REQUEST=VERIFY to verify the password.
- If the password is verified, DB2 then invokes RACF again via the RACROUTE macro with REQUEST=AUTH, to check whether the authorization ID is allowed to use DB2 resources defined to RACF.
- DB2 then invokes the connection exit routine if one has been defined.
- The connection then has a primary authorization ID, possibly one or more secondary IDs, and an SQL ID.

If the server is a remote DB2 subsystem, the requester generates authentication tokens and sends them to the remote site in the following way:

- The SECURITY_OUT column in SYSIBM.LUNAMES for SNA or the SECURITY_OUT column in SYSIBM.IPNames for TCP/IP must have one of the following values:
 - 'A' (already verified)
 - 'D' (userid and security-sensitive data encryption; TCP/IP only)
 - 'E' (userid, password, and security-sensitive data encryption; TCP/IP only)
 - 'P' (password)

When the value is 'A', the user ID and password specified on the CONNECT is still sent.

When the value is 'D', 'E', or 'P', the requester encrypts the user ID and password specified on the CONNECT for TCP/IP. However, if the Integrated Cryptographic Service Facility (ICSF) is not configured at the requester or if the server does not support encryption, one of the following actions occurs:

- If the value of SECURITY_OUT in SYSIBM.IPNames is 'D' or 'E', SQLCODE -904 is returned if ICSF is not configured at the requester, and SQLCODE -30082 is returned if the server does not support encryption.

- If the value of SECURITY_OUT in SYSIBM.IPNAMES is 'P', the requester does not encrypt the user ID and password and flows the tokens in clear text.
- For SNA, the ENCRYPTPSWDS column in SYSIBM.SYSLUNAMES must be not contain 'Y'.
- The authorization ID and password are verified at the server.
- In all cases, outbound translation—as specified in SYSIBM.USERNAMES—is not done.

Distributed unit of work: In general, the following are true:

- A CONNECT statement with the TO clause and the USER/USING clause can be executed only if no current or dormant connection to the named server exists. However, if the named server is the local DB2 subsystem and the CONNECT statement is the first SQL statement that is executed after the DB2 thread is created, the CONNECT statement executes successfully.
- A CONNECT statement without the TO clause but with the USER/USING clause can be executed only if no current or dormant connection to the local DB2 subsystem exists. However, if the CONNECT statement is the first SQL statement that is executed after the DB2 thread is created, the CONNECT statement executes successfully.

Remote unit of work: If the authorization check fails, the connection is placed in the connectable and unconnected state.

Precompiler options: Regardless of whether a program is precompiled with the CONNECT(1) or CONNECT(2) option, DB2 for z/OS negotiates with the remote server during the connection process to determine how to perform commits. If the remote server does not support the two-phase commit protocol, DB2 downgrades to perform one-phase commits.

Programs containing CONNECT statements that are precompiled with different CONNECT precompiler options cannot execute as part of the same application process. An error occurs when an attempt is made to execute the invalid CONNECT statement.

Host variables: If a CONNECT statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Error processing: A CONNECT statement can return and indicate a successful execution even when no physical connection yet exists. DB2 delays the physical connection process, when possible, to economize on the number of messages it sends to a server. Therefore, errors in CONNECT statement processing can be reported following the next executable SQL statement, not immediately following the CONNECT statement.

Examples

Example 1: Connect an application to a DBMS. The location name is in the character-string variable *LOCNAME*, the authorization identifier is in the character-string variable *AUTHID*, and the password is in the character-string variable *PASSWORD*.

```
EXEC SQL CONNECT TO :LOCNAME USER :AUTHID USING :PASSWORD;
```

Example 2: Obtain information about the current server.

```
EXEC SQL CONNECT;
```

Example 3: Execute SQL statements in a distributed unit of work. The first CONNECT statement creates a connection to the EASTDB server. The second CONNECT statement creates a connection to the WESTDB server, and places the SQL connection to EASTDB in the dormant state.

```
EXEC SQL CONNECT TO EASTDB;  
-- execute statements referencing objects at EASTDB  
EXEC SQL CONNECT TO WESTDB;  
-- execute statements referencing objects at WESTDB
```

Example 4: Connect the application to a DBMS whose location identifier is in the character-string variable LOC using the authorization identifier in the character-string variable AUTHID and the password in the character-string variable PASSWORD. Perform work for the user, and then release the connection and connect again using a different user ID and password.

```
EXEC SQL CONNECT TO :LOC USER :AUTHID USING :PASSWORD;  
-- execute SQL statements accessing data on the server  
RELEASE :LOC;  
EXEC SQL COMMIT;  
-- set AUTHID and PASSWORD to new values  
EXEC SQL CONNECT TO :LOC USER :AUTHID USING :PASSWORD;  
-- execute SQL statements accessing data on the server
```

Example 5: Change servers in a remote unit of work. Assume that the application connected to a remote DB2 server, opened a cursor, and fetched rows from the cursor's result table. Subsequently, to connect to the local DB2 subsystem, the application executes the following statements:

```
EXEC SQL COMMIT WORK;  
EXEC SQL CONNECT RESET;
```

The COMMIT is required because opening the cursor caused the application to enter the unconnectable and connected state.

If the cursor was declared with the WITH HOLD clause and was not closed with a CLOSE statement, it would still be open even after execution of the COMMIT statement. However, it would be closed with the execution of the CONNECT statement.

CREATE ALIAS

The CREATE ALIAS statement defines an alias for a table or view. The definition is recorded in the DB2 catalog at the current server. The table or view does not have to be described in that catalog.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEALIAS privilege
- SYSADM or SYSCTRL authority
- DBADM or DBCTRL authority on the database that contains the table, if the alias is for a table and the value of field DBADM CREATE AUTH on installation panel DSNTIPP is YES

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner of the plan or package is a role, this role must hold the privileges for the privilege set. If the specified alias name includes a qualifier that is not the same as this authorization ID, the privilege set must include one of the following authorities:

- SYSADM or SYSCTRL authority
- DBADM or DBCTRL authority on the database that contains the table, if the alias is for a table and the value of field DBADM CREATE AUTH on installation panel DSNTIPP is YES

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, or SYSCTRL authority.

If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. If the process is not running in a trusted context that is defined with the ROLE AS OBJECT OWNER clause and the specified alias name includes a qualifier that is not the same as this authorization ID:

- The privilege set must include SYSADM or SYSCTRL authority.

- The privilege set must include DBADM or DBCTRL authority on the database that contains the table, if the alias is for a table and the value of field DBADM CREATE AUTH on installation panel DSNTIPP is YES.
- The qualifier must be the same as one of the authorization IDs of the process and the privileges that are held by that authorization ID must include the CREATEALIAS privilege. This is an exception to the rule that the privilege set is the privileges that are held by the SQL authorization ID of the process.

►► CREATE ALIAS *alias-name* FOR

<i>table-name</i>
<i>view-name</i>

 ►►

alias-name

FOR *table-name* **or** *view-name*

Notes

An alias can be defined for a table, view, or alias that is not at the current server. When so defined, the existence of the referenced object is not verified at the time the alias is created. But the object must exist when a statement that contains the alias is executed. And if that object is also an alias, it must refer to a table or view at the server where that alias is defined.

When an application uses three-part name aliases for remote objects and DRDA access, the application program must be bound at each location that is specified in the three-part names. Also, you need to define the alias at the remote site as well as at the local site.

Example

Create an alias for a catalog table at a DB2 with location name DB2USCALABOA5281.

```
CREATE ALIAS LATABLES FOR DB2USCALABOA5281.SYSIBM.SYSTABLES;
```

CREATE AUXILIARY TABLE

The CREATE AUXILIARY TABLE statement creates an auxiliary table at the current server for storing LOB data.

Invocation

This statement can be embedded in an application program or issued interactively if the value of special register CURRENT RULES is 'DB2' and the table space is explicitly created when the statement is executed. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Do not use this statement if the value of special register CURRENT RULES is 'STD' or if the table space is implicitly created. When the values of the CURRENT RULES special register is 'STD' and a base table is created with LOB columns or altered such that LOB columns are added, DB2 automatically creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column. DB2 also automatically creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column if the table space is implicitly created. DB2 chooses the names and characteristics of these objects. For more information about the names and the characteristics, see Creating a table with LOB columns.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATETAB privilege for the database implicitly or explicitly specified by the IN clause
- DBADM, DBCTRL, or DBMAINT authority for the database
- SYSADM or SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specifies, a role is the owner. Otherwise, an authorization ID is the owner. If the specified table name includes a qualifier that is not the same as this authorization ID, the privilege set must include SYSADM or SYSCTRL authority, DBADM authority for the database, or DBCTRL authority for the database.

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, or SYSCTRL authority.

If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

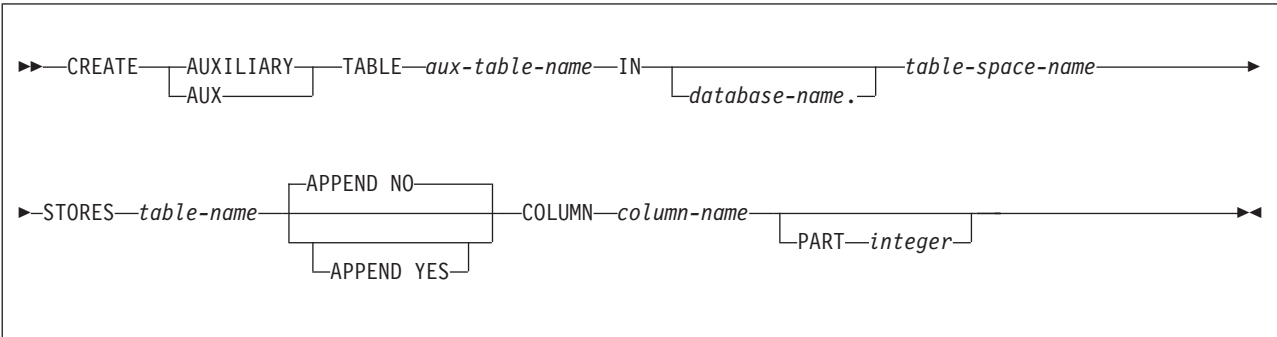
- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privilege set is the set of privileges that are held by the role that is associated with the primary authorization ID of the process. If the process is in a trusted

context, any authorization ID can be the qualifier. However, if the process is not in a trusted context and if the specified table name includes a qualifier that is not the same as the SQL authorization ID of the process, the following rules apply:

- If the privilege set includes SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier can be any valid schema name.
- If the privilege set lacks SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier is valid only if it is the same as one of the authorization IDs of the process and the privilege set that are held by that authorization ID includes all²⁵ privileges needed to create the table.

Syntax



Description

AUXILIARY or AUX

Specifies a table that is used to store the LOB data for a LOB column (or a column with a distinct type that is based on a LOB data type).

aux-table-name

Names the auxiliary table. The name, including the implicit or explicit qualifiers, must not identify a table, view, alias, or synonym that exists at the current server. If the name is qualified, the name can be a two-part or three-part name. If a three-part name is used, the first part must match the value of field DB2 LOCATION NAME on installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.)

IN database-name.table-space-name or IN table-space-name

Identifies the table space in which the auxiliary table is created. The name must identify an empty LOB table space that currently exists at the current server. The LOB table space must be in the same database as the associated base table.

If you specify a database and a table space, the table space must belong to the specified database. If you specify only a table space, it must belong to the database that contains the specified table space. If you specify only a table space, this table space must belong to DSNDB04. This type of table space is only created when SET CURRENT RULES='DB2' is specified.

STORES table-name COLUMN column-name

Identifies the base table and the column of that table that is to be stored in the

25. Exception: The CREATETAB privilege is checked on the SQL authorization ID of the process.

auxiliary table. If the base table is nonpartitioned, an auxiliary table must not already exist for the specified column. If the base table is partitioned, an auxiliary table must not already exist for the specified column and specified partition.

The encoding scheme for the LOB data stored in the auxiliary table is the same as the encoding scheme for the base table. It is either ASCII, EBCDIC, or UNICODE depending on the value of the CCSID clause when the base table was created.

APPEND NO or APPEND YES

Specifies whether append processing is used for the table. The APPEND clause must not be specified for a table in a work file table space.

If the base table is in a range-partitioned table space, the APPEND option on the LOB table might be different for each partition (depending if the LOB table space and associated objects for each partition are created explicitly or implicitly). If the base table is in a partition by growth table space, the APPEND attributes of LOB table will be inherited by each partition.

APPEND NO

Specifies that append processing is not used for the table. For insert and LOAD operations, DB2 will attempt to place data rows in a well clustered manner with respect to the value in the row's cluster key columns.

APPEND NO is the default

APPEND YES

Specifies that data rows are placed into the table without regard to clustering during the insert and LOAD operations.

PART integer

Specifies the partition of the base table for which the auxiliary table is to store the specified column. You can specify PART only if the base table is defined in a partitioned table space, and no other auxiliary table exists for the same LOB column of the base table.

Notes

Owner privileges: There are no specific privileges on an auxiliary table. For more information about ownership of an object, see "Authorization, privileges, and object ownership" on page 60.

Determining the number of auxiliary tables to create: If the base table is nonpartitioned, you might need to create one LOB table space and one auxiliary table for each LOB column in the base table. If the base table is partitioned, for each LOB column, you might need to create one LOB table space and one auxiliary table for each partition. For example if your base table has three partitions and two LOB columns, you might need to create three LOB table spaces and three auxiliary tables for each LOB column. In other words, you might need a total of six LOB table spaces and six auxiliary tables.

Auxiliary tables in LOB table spaces that are logged: When you create an auxiliary table in a LOB table space that is LOGGED, and the associated base table space is NOT LOGGED, the logging attribute of the LOB table space is implicitly changed to NOT LOGGED and the logging attributes of the base table space and the LOB table space are linked.

| *Append processing and unused free space in a table:* An update or delete of LOB
| data creates some free space in the LOB table that can be used by the next insert. If
| the table uses append processing, any free space that is not at the end of the table
| space will not be reused during the insert operation. Any unused free space in the
| table can be reclaimed by running the REORG utility with either the SHRELEV
| REFERENCE or SHRLEVEL CHANGE keywords. The REORG utility is not
| influenced by the APPEND option.

Example

Assume that a column named EMP_PHOTO with a data type of BLOB(110K) has been added to sample employee table DSN8910.EMP for each employee's photo. Create auxiliary table EMP_PHOTO_ATAB to store the BLOB data for the BLOB column in LOB table space DSN8D91A.PHOTOLTS.

```
CREATE AUX TABLE EMP_PHOTO_ATAB  
  IN DSN8D91A.PHOTOLTS  
  STORES DSN8910.EMP  
  COLUMN EMP_PHOTO;
```

CREATE DATABASE

The CREATE DATABASE statement defines a DB2 database at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

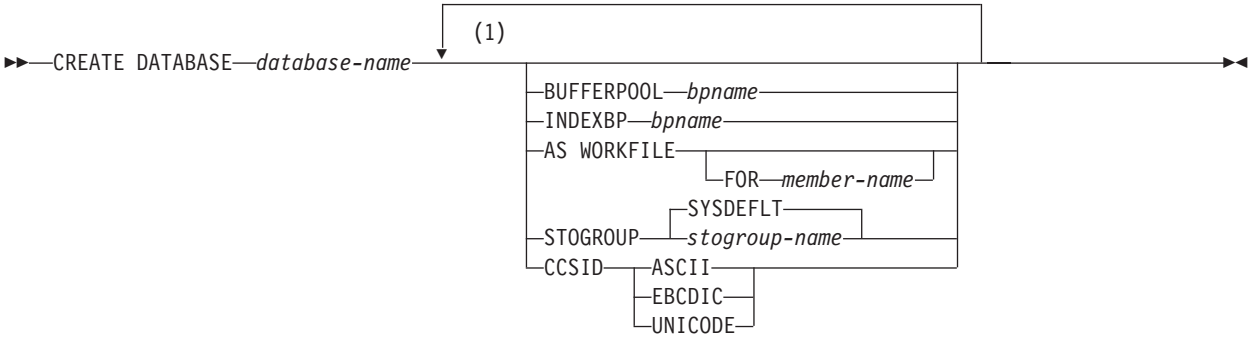
- The CREATEDBA privilege
- The CREATEDBC privilege
- SYSADM or SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privilege set is the set of privileges that are held by the role that is associated with the primary authorization ID of the process.

See “Notes” on page 914 for the authorization effect of a successful CREATE DATABASE statement.

Syntax



Notes:

- 1 The same clause must not be specified more than one time.

Description

database-name

Names the database. The name must not start with DSNDB and must not identify a database that exists at the current server. *database-name* must not be in the form of eight characters that start with DSN followed by exactly five

digits. If the database is to be a work file database in a data sharing environment, DSNDB07 is an acceptable work file database name. However, only one member of a data sharing group can use DSNDB07 as the name of its work file database.

BUFFERPOOL *bpname*

Specifies the default buffer pool name to be used for table spaces created within the database. If the database is a work file database, 8KB and 16KB buffer pools cannot be specified. See “Naming conventions” on page 51 for more details about *bpname*.

If you omit the BUFFERPOOL clause, the default 4-KB buffer pool for user data that is specified on installation panel DSNTIP1 is used.

INDEXBP *bpname*

Specifies the default buffer pool name to be used for the indexes created within the database. The name can identify a 4KB, 8KB, 16KB, or 32KB buffer pool. See “Naming conventions” on page 51 for more details about *bpname*.

If you omit the INDEXBP clause, the buffer pool specified for user indexes on installation panel DSNTIP1 is used. The default value for the user indexes field on that panel is BP0.

AS WORKFILE

Specifies the database is a work file database. AS WORKFILE can be specified only in a data sharing environment. Only one work file database can be created for each DB2 subsystem. The work file database is used for work files, created global temporary table, declared temporary tables, and sensitive static scrollable cursors.

PUBLIC implicitly receives the CREATETAB privilege (without GRANT authority) to define a declared temporary table in the work file database. This implicit privilege is not recorded in the DB2 catalog and cannot be revoked.

The CCSID clause is not supported for a work file database. If you specify AS WORKFILE, do not use the CCSID clause.

FOR *member-name*

Specifies the member for which this database is to be created. Specify FOR member-name only in a data sharing environment.

If FOR *member-name* is not specified, the member is the DB2 subsystem on which the CREATE DATABASE statement is executed.

STOGROUP *stogroup-name*

Specifies the storage group to be used, as required, as a default storage group to support DASD space requirements for table spaces and indexes within the database. The default is SYSDEFLT.

CCSID *encoding-scheme*

Specifies the default encoding scheme for data stored in the database. The default applies to table spaces created in the database. All tables stored within a table space must use the same encoding scheme.

ASCII Specifies that the data must be encoded using the ASCII CCSIDs of the server.

EBCDIC

Specifies that the data must be encoded using the EBCDIC CCSIDs of the server.

UNICODE

Specifies that the data must be encoded using the UNICODE CCSIDs of the server.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or UNICODE data is used.

The option defaults to the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Do not use the CCSID clause if you specify the AS WORKFILE clause.

Notes

If the statement is embedded in an application program, the owner of the plan or package is the owner of the database. If the statement is dynamically prepared, the SQL authorization ID of the process is the owner of the database.

If the owner of the database has the CREATEDBA, SYSADM, or SYSCTRL authority, the owner acquires DBADM authority for the database. DBADM authority for a database includes table privileges on all tables in that database. Thus, if a user with SYSCTRL authority creates a database, that user has table privileges on all tables in that database. This is an exception to the rule that SYSCTRL authority does not include table privileges.

If the owner of the database has the CREATEDBC privilege, but not the CREATEDBA privilege, the owner acquires DBCTRL authority for the database. In this case, no authorization ID has DBADM authority for the database until it is granted by an authorization ID with SYSADM authority.

Examples

Example 1: Create database DSN8D91P. Specify DSN8G910 as the default storage group to be used for the table spaces and indexes in the database. Specify 8KB buffer pool BP8K1 as the default buffer pool to be used for table spaces in the database, and BP2 as the default buffer pool to be used for indexes in the database.

```
CREATE DATABASE DSN8D91P
  STOGROUP DSN8G910
  BUFFERPOOL BP8K1
  INDEXBP BP2;
```

Example 2: Create database DSN8TEMP. Use the defaults for the default storage group and default buffer pool names. Specify ASCII as the default encoding scheme for data stored in the database.

```
CREATE DATABASE DSN8TEMP
  CCSID ASCII;
```

CREATE FUNCTION

The CREATE FUNCTION statement registers a user-defined function with a database server. You can register four different types of functions with this statement, each of which is described separately.

External scalar

The function is written in a programming language and returns a scalar value. The external executable is registered with a database server along with various attributes of the function. See “CREATE FUNCTION (external scalar)” on page 916.

External table

The function is written in a programming language and returns a complete table. The external executable is registered with a database server along with various attributes of the function. See “CREATE FUNCTION (external table)” on page 940.

Sourced

The function is implemented by invoking another function (either built-in, external, SQL, or sourced) that already exists at the server. The function inherits the attributes of the underlying source function. See “CREATE FUNCTION (sourced)” on page 958.

SQL scalar

The function is written exclusively in SQL statements and returns a scalar value. The body of an SQL scalar function is written in the SQL procedural language and is defined at the current server along with various attributes of the function. See “CREATE FUNCTION (SQL scalar)” on page 972.

CREATE FUNCTION (external scalar)

This CREATE FUNCTION statement registers a user-defined external scalar function with a database server. A scalar function returns a single value each time it is invoked.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

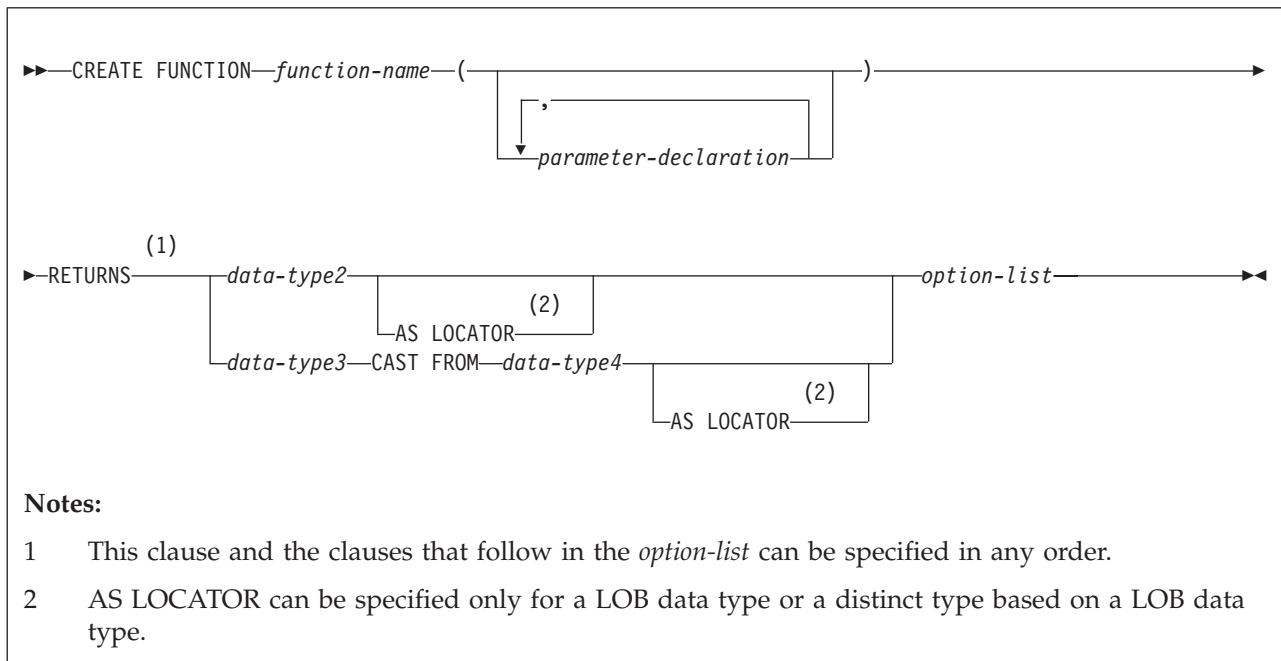
- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Additional privileges are required if the function uses a table as a parameter, refers to a distinct type, or is to run in a WLM (workload manager) environment. These privileges are:

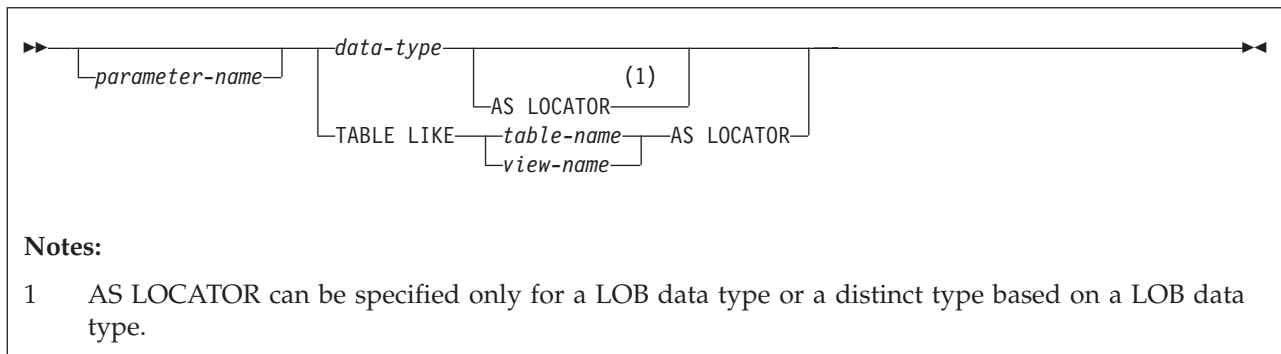
- The SELECT privilege on any table that is an input parameter to the function.
- The USAGE privilege on each distinct type that the function references.
- Authority to create programs in the specified WLM environment. This authorization is obtained from an external security product, such as RACF.

When LANGUAGE is JAVA and a *jar-name* is specified in the EXTERNAL NAME clause, the privilege set must include USAGE on the JAR file.

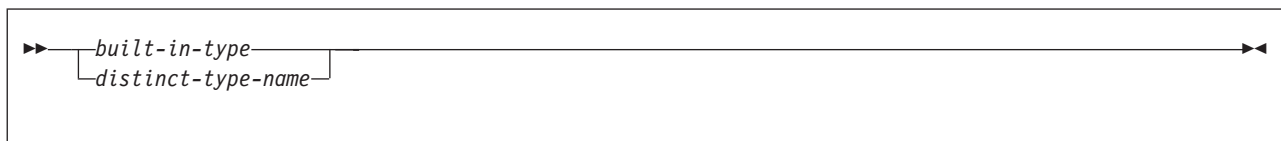
Syntax



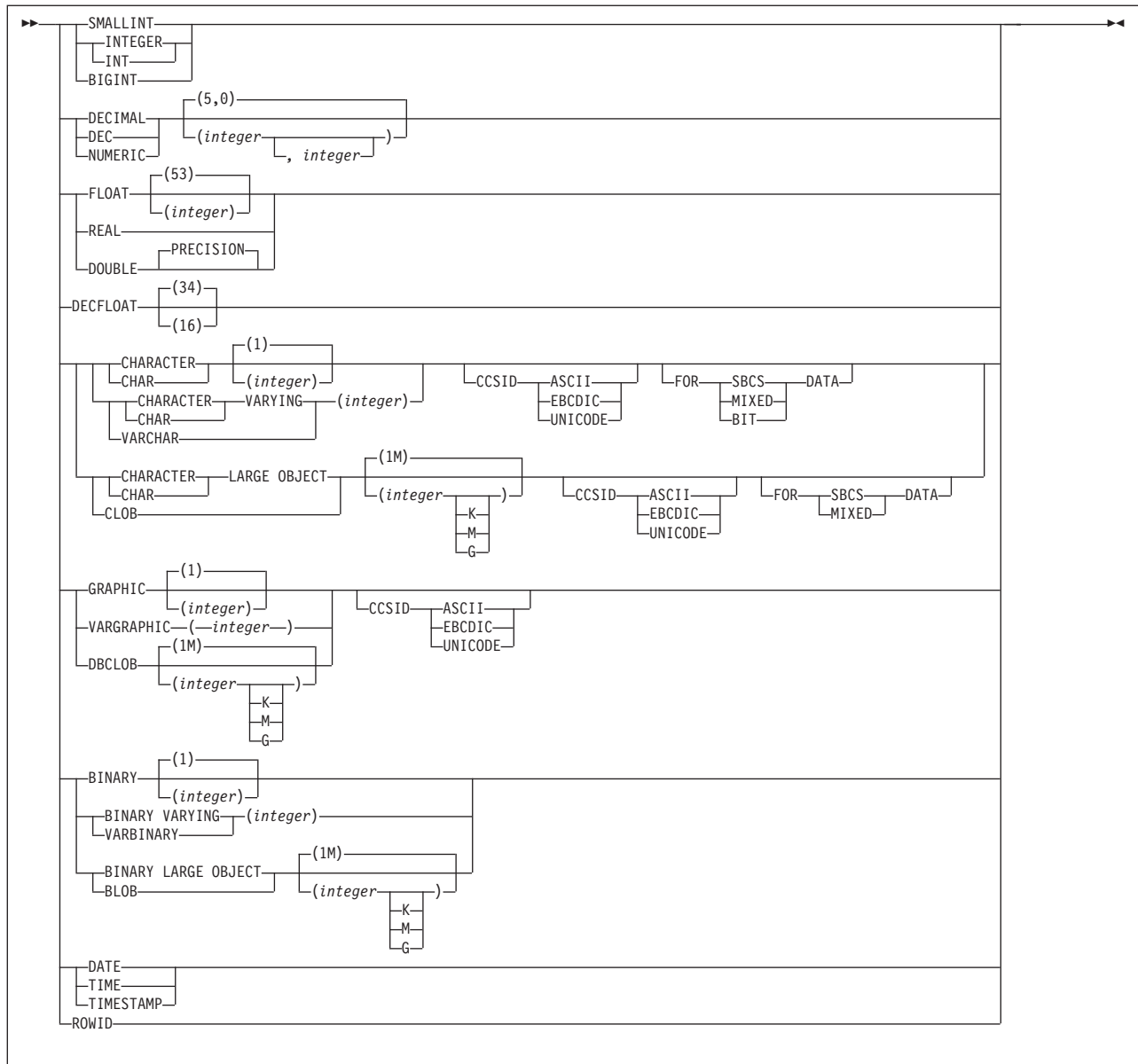
parameter-declaration:



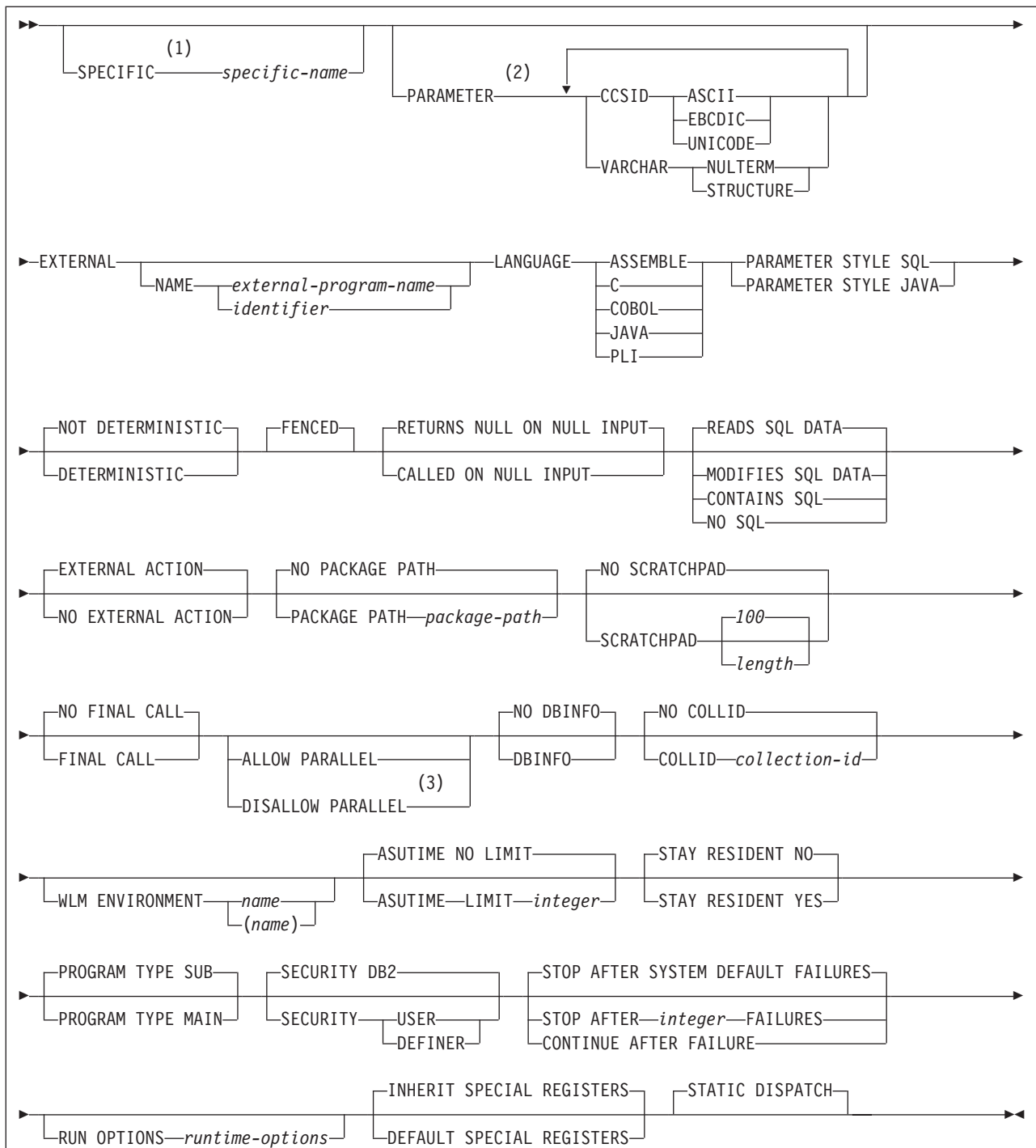
data-type:



built-in-type:



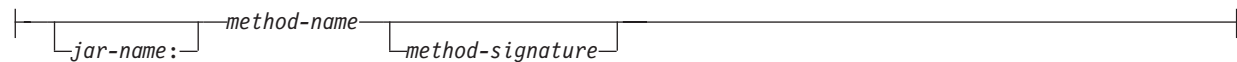
option-list:



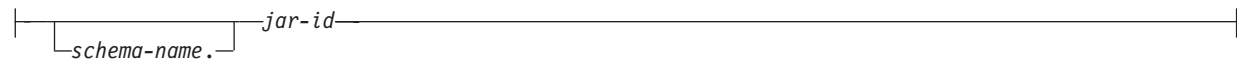
Notes:

- 1 The clauses in the option list can be specified in any order
- 2 The same clause must not be specified more than one time.
- 3 If NOT DETERMINISTIC, EXTERNAL ACTION, SCRATCHPAD, or FINAL CALL is specified, DISALLOW PARALLEL is the default.

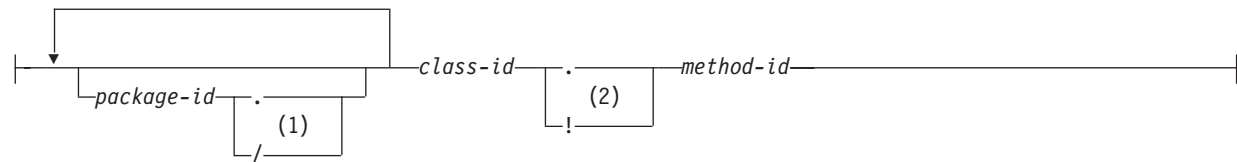
external-java-routine-name:



jar-name:



method-name:



method-signature:



Notes:

- 1 The slash (/) is supported for compatibility with previous release of DB2 for z/OS.
- 2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

Description

function-name

Names the user-defined function. The name is implicitly or explicitly qualified by a schema name.

The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, subtype or encoding scheme attributes of the data type) must not identify a user-defined function that exists at the current server. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

You can use the same name for more than one function if the function signature of each function is unique.

- The unqualified form of *function-name* must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

ALL	LIKE	UNIQUE
AND	MATCH	UNKNOWN
ANY	NOT	=
BETWEEN	NULL	≠
DISTINCT	ONLY	<

	EXCEPT	OR	<=
	EXISTS	OVERLAPS	<<
	FALSE	SIMILAR	>
	FOR	SOME	>=
	FROM	TABLE	>>
	IN	TRUE	<>
	IS	TYPE	

| The schema name can be 'SYSTOOLS' or 'SYSFUN' if the user who executes the
 | CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema
 | name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(parameter-declaration,...)

Identifies the number of input parameters of the function, and specifies the data type of each parameter. All of the parameters for a function are input parameters and are nullable. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

parameter-name

Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name.

data-type

Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the input parameter is a built-in data type.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

If you specify the name of the distinct type without a schema name, DB2 resolves the schema name by searching the schemas in the SQL path.

| The implicitly or explicitly specified encoding scheme of all of the
 | parameters with a character or graphic string data type must be the
 | same—either all ASCII, all EBCDIC, or all UNICODE.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked. An error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

- A distinct type parameter is passed as the source type of the distinct type.

AS LOCATOR

Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

TABLE LIKE *table-name* or *view-name* AS LOCATOR

Specifies that the parameter is a transition table. However, when the function is invoked, the actual values in the transition table are not passed to the function. A single value is passed instead. This single value is a locator to the table, which the function uses to access the columns of the transition table. A function with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACE. The number of columns and the attributes of those columns are determined at the time the CREATE FUNCTION statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the function.

table-name or *view-name* must identify a table or view that exists at the current server. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the function cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE FUNCTION statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.

- If the CREATE FUNCTION statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the function is invoked, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length, precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE FUNCTION statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE FUNCTION statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

RETURNS

Specifies the data type for the result of the function. Consider this clause in conjunction with the optional CAST FROM clause.

data-type2

Specifies the data type of the output. The output parameter is nullable.

The same considerations that apply to the data type and nullability of input parameter, as described under *data-type*, apply to the data type of the result of the function.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only if the output from the function has a LOB data type or a distinct type based on a LOB data type.

data-type3 **CAST FROM** *data-type4*

Specifies the data type of the output of the function (*data-type4*) and the data type in which that output is returned to the invoking statement (*data-type3*). The two data types can be different. For example, for the following definition, the function returns a DOUBLE value, which DB2 converts to a DECIMAL value and then passes to the statement that invoked the function:

```
CREATE FUNCTION SQRT(DECIMAL(15,0))
  RETURNS DECIMAL(15,0) CAST FROM DOUBLE
...
```

The value of *data-type4* must not be a distinct type and must be castable to *data-type3*. The value for *data-type3* can be any built-in data type other than a distinct type. (For information on casting data types, see “Casting between data types” on page 96.) The encoding scheme of the parameters, if they are string data types, must be the same.

If the PARAMETER VARCHAR clause is specified, *data-type3* and *data-type4* should be specified as VARCHAR.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the value. You can specify AS LOCATOR only if *data-type4* is a LOB data type or a distinct type based on a LOB data type.

SPECIFIC *specific-name*

Specifies a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

SQLxxxxxxxxxxxx

where 'xxxxxxxxxxxx' is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and must be used in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

PARAMETER CCSID or VARCHAR

Specifies the encoding scheme for character and graphic string parameters, and in the case of LANGUAGE C, specifies that representation of variable length string parameters.

CCSID

Indicates whether the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or RETURNS clause, or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for *all* string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

VARCHAR

Specifies that the representation of the values of varying length character string-parameters, including, if applicable, the output of the function, for functions which specify LANGUAGE C.

This option can only be specified if LANGUAGE C is also specified.

NULTERM

Specifies that variable length character string parameters are represented in a NUL-terminated string form.

STRUCTURE

Specifies that variable length character string parameters are represented in a VARCHAR structure form.

Using the PARAMETER VARCHAR clause, there is no way to specify the VARCHAR form of an individual parameter as there is with the PARAMETER CCSID clause. The PARAMETER VARCHAR clause only applies to parameters in the parameter list of a function and in the RETURNS clause. It does not apply to system-generated parameters of the routine such as message tokens and DBINFO.

In a data sharing environment, you should not specify the PARAMETER VARCHAR clause until all members of the data sharing group support the clause. If some group members support this clause and others do not, and PARAMETER VARCHAR is specified in an external routine, the routine will encounter different parameter forms depending on which group member invokes the routine.

EXTERNAL

Specifies that the CREATE FUNCTION statement is being used to define a new function that is based on code that is written in an external programming language.

DB2 loads the load module when the function is invoked. The load module is created when the program that contains the function body is compiled and link-edited. The load module does not need to exist when the CREATE FUNCTION statement is executed. However, it must exist and be accessible by the current server when the function is invoked.

You can specify the EXTERNAL clause in one of the following ways:

```
EXTERNAL  
EXTERNAL NAME PKJVSP1  
EXTERNAL NAME 'PKJVSP1'
```

If you specify an external program name, you must use the NAME keyword. For example, this syntax is not valid:

```
EXTERNAL PKJVSP1
```

NAME *external-program-name* **or** *identifier*

Identifies the user-written code that implements the user-defined function.

If LANGUAGE is JAVA, *external-program-name* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *external-program-names* are specified, the total length of all of them must not be greater than 1305 bytes and they must be separated by a space or a line break. Do not specify a JAR file for a JAVA function for which NO SQL is also specified.

An *external-java-routine-name* contains the following parts:

jar-name

Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified

with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER bind option when the package or plan was created or last rebound. If the QUALIFIER was not specified, the schema name is the owner of the package or plan.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the CREATE FUNCTION statement is processed.

If *jar-name* is not specified, the function is loaded from the class file directly instead of being loaded from a JAR file. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed function.

method-name

Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method ID's are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what these can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id

Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.UserFuncs." The Java virtual machine looks in the directory "/myPacks/UserFuncs/" for the classes.

class-id

Identifies the class identifier of the Java object.

method-id

Identifies the method identifier with the Java class to be invoked.

method-signature

Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the user-defined function involves any input or output parameters that can be NULL. When the function being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the CREATE FUNCTION statement.

For other values of LANGUAGE, the name can be a string constant that is no longer than 8 characters. It must conform to the naming conventions for load modules. Alphabetical extenders for national languages can be used as the first character and as subsequent characters in the load module name.

If you do not specify the NAME clause, 'NAME *function-name*' is implicit. In this case, *function-name* must not be longer than 8 characters.

LANGUAGE

Specifies the language interface convention to which the body of the function is written. All programs must be designed to run in IBM's Language Environment environment.

ASSEMBLE

The function is written in Assembler.

C The function is written in C or C++.

COBOL

The function is written in COBOL, including the object-oriented language extensions.

JAVA

The user-defined function is written in Java and is executed in the Java Virtual Machine. When LANGUAGE JAVA is specified, the EXTERNAL NAME clause must also be specified with a valid *external-java-routine-name* and PARAMETER STYLE must be specified with JAVA.

Do not specify LANGUAGE JAVA when SCRATCHPAD, FINAL CALL, DBINFO, PROGRAM TYPE MAIN, or RUN OPTIONS is in effect.

PLI

The function is written in PL/I.

PARAMETER STYLE

Specifies the conventions for passing parameters to and returning a value from the function.

SQL

Specifies the parameter passing convention that supports passing null values both as input and for output. The parameters that are passed between the invoking SQL statement and the function include:

- *n* parameters for the input parameters that are specified for the function
- A parameter for the result of the function
- *n* parameters for the indicator variables for the input parameters
- A parameter for the indicator variable for the result
- The SQLSTATE to be returned to DB2
- The qualified name of the function
- The specific name of the function
- The SQL diagnostic string to be returned to DB2
- The function can also pass from zero to three additional parameters:
 - The scratchpad, if SCRATCHPAD is specified
 - The call type, if FINAL CALL is specified
 - The DBINFO structure, if DBINFO is specified

JAVA

Indicates that the user-defined function uses a convention for passing parameters that conforms to the Java and SQLJ specifications.

PARAMETER STYLE JAVA can be specified only if LANGUAGE is specified as JAVA. JAVA must be specified for PARAMETER STYLE when LANGUAGE JAVA is specified.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC

The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT or SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers, or any function that contains SQL statements.

NOT DETERMINISTIC is the default.

Some functions that are not deterministic can receive incorrect results if the function is executed by parallel tasks. Specify the DISALLOW PARALLEL clause for these functions.

DETERMINISTIC

The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT or SQL data change statements that refer to this function.

DETERMINISTIC is not the default. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

FENCED

Specifies that the external function runs in an external address space to prevent the function from corrupting DB2 storage.

FENCED is the default.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON NULL INPUT

The function is not called if any of the input arguments is null. The result is the null value. RETURNS NULL ON INPUT is the default.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL

Specifies the classification of SQL statements that the function can execute. DB2 verifies that the SQL statements that the function issues are consistent with this specification. For the data access classification of each statement, see Table 138 on page 1605.

MODIFIES SQL DATA

Specifies that the function can execute any SQL statement except the statements that are not supported in functions. Do not specify MODIFIES SQL DATA when ALLOW PARALLEL is in effect.

READS SQL DATA

Specifies that the function can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data. The default is READS SQL DATA.

CONTAINS SQL

Specifies that the function can execute only SQL statements with an access indication of CONTAINS SQL or NO SQL. The function cannot execute statements that read or modify data.

NO SQL

Specifies that the function can execute only SQL statements with a data access classification of NO SQL. Do not specify NO SQL for a JAVA function that uses a JAR file.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

Because DB2 uses the RRS attachment for external functions, DB2 can participate in two-phase commit with any other resource manager that uses RRS. For resource managers that do not use RRS, there is no coordination of commit or rollback operations on non-DB2 resources.

EXTERNAL ACTION

Specifies that the function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

If you specify EXTERNAL ACTION, DB2:

- Materializes the views and table expressions in SELECT or data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify EXTERNAL ACTION if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

EXTERNAL ACTION is the default.

NO EXTERNAL ACTION

Specifies that the function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT and data change statements that refer to this function. NO EXTERNAL ACTION is

not the default. If applicable, specify NO EXTERNAL ACTION to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

NO PACKAGE PATH or PACKAGE PATH *package-path*

Specifies the package path to use when the function is run. This is the list of the possible package collections into which the DBRM this is associated with the function is bound.

NO PACKAGE PATH

Specifies that the list of package collections for the function is the same as the list of package collection IDs for the program that invokes the function. If the program that invokes the function does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*

Specifies a list of package collections, in the same format as the SET CURRENT PACKAGE PATH special register.

If the COLLID clause is specified with PACKAGE PATH, the COLLID clause is ignored when the function is invoked.

The *package-path* value that is provided when the function is created is checked when the function is invoked. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO SCRATCHPAD or SCRATCHPAD

Specifies whether DB2 is to provide a scratchpad for the function. It is strongly recommended that external functions be reentrant, and a scratchpad provides an area for the function to save information from one invocation to the next.

NO SCRATCHPAD

Specifies that a scratchpad is not allocated and passed to the function. NO SCRATCHPAD is the default.

SCRATCHPAD *length*

Specifies that when the function is invoked for the first time, DB2 allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- DB2 initializes the scratchpad to all binary zeros (X'00's).
- The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

```
SELECT A, UDFX(A) FROM TABLEB
WHERE UDFX(A) > 103 OR UDFX(A) < 19;
```

If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the

SQL statement. Specify the `DISALLOW PARALLEL` clause for functions that will not work correctly with parallelism.

- The scratchpad is persistent. DB2 preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. DB2 initializes the scratchpads when it begins to execute an SQL statement. DB2 does not reset scratchpads when a correlated subquery begins to execute.
- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify `FINAL CALL` to ensure that DB2 calls the function one more time so that the function can free those system resources.

Each time the function invoked, DB2 passes an additional argument to the function that contains the address of the scratchpad.

If you specify `SCRATCHPAD`, DB2:

- Does not move the function from one task control block (TCB) to another between `FETCH` operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared `WITH HOLD`.

Do not specify `SCRATCHPAD` when `LANGUAGE JAVA` is in effect.

NO FINAL CALL or FINAL CALL

Specifies whether a *final call* is made to the function. A final call enables the function to free any system resources that it has acquired. A final call is useful when the function has been defined with the `SCRATCHPAD` keyword and the function acquires system resource and anchors them in the scratchpad.

NO FINAL CALL

Specifies that a final call is not made to the function. The function does not receive an additional argument that specifies the type of call. `NO FINAL CALL` is the default.

FINAL CALL

Specifies that a final call is made to the function. To differentiate between final calls and other calls, the function receives an additional argument that specifies the type of call. The types of calls are:

First call

Specifies that the first call to the function for this reference to the function in this SQL statement. A first call is a normal call—SQL arguments are passed and the function is expected to return a result.

Normal call

Specifies that SQL arguments are passed and the function is expected to return a result.

Final call

Specifies that the last call to the function to enable the function to free resources. A final call is not a normal call. If an error occurs, DB2 attempts to make the final call unless the function abended. A final call occurs at these times:

- *End of statement*: When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.

- *End of a parallel task:* When the function is executed by parallel tasks.
- *End of transaction:* When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made when the cursor is closed or the application ends. If a commit occurs at the end of a parallel task, a final call is made regardless of whether a cursor defined as WITH HOLD is open.

If a commit, rollback, or abort operation causes the final call, the function cannot issue any SQL statements when it is invoked.

Some functions that use a final call can receive incorrect results if parallel tasks execute the function. For example, if a function sends a note for each final call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that have inappropriate actions when executed in parallel.

Do not specify FINAL CALL when LANGUAGE JAVA is in effect.

ALLOW or DISALLOW PARALLEL

For a single reference to the function, specifies whether parallelism can be used when the function is invoked. Although parallelism can be used for most scalar functions, some functions such as those that depend on a single copy of the scratchpad cannot be invoked with parallel tasks.

Consider these characteristics when determining which clause to use:

- If all invocations of the function are completely independent from one another, specify ALLOW PARALLEL.
- If each invocation of the function updates the scratchpad, providing values that are of interest to the next invocation, such as incrementing a counter, specify DISALLOW PARALLEL.
- If the scratchpad is used only so that some expensive initialization processing is performed a minimal number of times, specify ALLOW PARALLEL.
- If the function performs some external action that should apply to only one partition, specify DISALLOW PARALLEL.
- If the function is defined with MODIFIES SQL DATA, specify DISALLOW PARALLEL, not ALLOW PARALLEL.

ALLOW PARALLEL is the default unless NOT DETERMINISTIC, EXTERNAL ACTION, SCRATCHPAD, or FINAL CALL is specified, in which case, DISALLOW PARALLEL is the default.

ALLOW PARALLEL

Specifies that DB2 can consider parallelism for the function. Parallelism is not forced on the SQL statement that invokes the function or on any SQL statement in the function. Existing restrictions on parallelism apply.

DISALLOW PARALLEL

Specifies that DB2 does not consider parallelism for the function.

NO DBINFO or DBINFO

Specifies whether additional status information is passed to the function when it is invoked.

NO DBINFO

No additional information is passed. NO DBINFO is the default.

DBINFO

An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application runtime authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

Do not specify DBINFO when LANGUAGE JAVA is in effect.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is to be used when the function is executed. This is the package collection into which the DBRM that is associated with the function program is bound.

NO COLLID

The package collection for the function is the same as the package collection of the program that invokes the function. If a trigger invokes the function, the collection of the trigger package is used. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*

The name of the package collection that is to be used when the function is executed.

WLM ENVIRONMENT

Identifies the WLM (workload manager) application environment in which the function is to run. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the function runs in the WLM-established stored procedure address space that is specified at installation time. When LANGUAGE is JAVA, you must specify WLM ENVIRONMENT, and the WLM environment in which the function is to run must be Java-enabled.

name

The WLM environment in which the function must run. If another user-defined function or a stored procedure calls the function and that calling routine is running in an address space that is not associated with the WLM environment, DB2 routes the function request to a different address space.

(name,*)

When an SQL application program directly invokes the function, the WLM environment in which the function runs.

If another user-defined function or a stored procedure calls the function, the function runs in same environment that the calling routine uses. In this case, authorization to run the function in the WLM environment is not checked because the authorization of the calling routine suffices.

Users must have the appropriate authorization to execute functions in the specified WLM environment. For an example of a RACF command that provides this authorization, see *Running external functions in WLM environments*.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of the function can run. The value is unrelated to the ASUTIME column of the resource limit specification table. This option is ignored if LANGUAGE JAVA is specified.

When you are debugging a function, setting a limit can be helpful if the function gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the service units. NO LIMIT is the default.

LIMIT integer

The limit on the service units is a positive integer in the range of 1 to 2 147 483 647. If the function uses more service units than the specified value, DB2 cancels the function.

STAY RESIDENT

Specifies whether the load module for the function is to remain resident in memory when the function ends. This option is ignored if LANGUAGE JAVA is specified.

NO The load module is deleted from memory after the function ends. Use NO for non-reentrant functions. NO is the default.

YES

The load module remains resident in memory after the function ends. Use YES for reentrant functions.

PROGRAM TYPE

Specifies whether the function program runs as a main routine or a subroutine.

SUB

The function runs as a subroutine. With LANGUAGE JAVA, PROGRAM TYPE SUB is the only valid option. SUB is the default.

MAIN

The function runs as a main routine.

SECURITY

Specifies how the function interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2

The function does not require an external security environment. If the function accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the WLM-established stored procedure address space.

DB2 is the default.

USER

An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the primary authorization ID of the process that invoked the function.

DEFINER

An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the function.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE

Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES

Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX. This is the default.

STOP AFTER *nn* FAILURES

Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE

Specifies that this routine should not be placed in a stopped state after any failure.

RUN OPTIONS *runtime-options*

Specifies the Language Environment runtime options to be used for the function. You must specify *runtime-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any runtime options to Language Environment, and Language Environment uses its installation defaults. For a description of the Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

Do not specify RUN OPTIONS when LANGUAGE JAVA is in effect.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Specifies that the values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a user-defined function in Table 37 on page 151.

DEFAULT SPECIAL REGISTERS

Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a user-defined function in Table 37 on page 151.

STATIC DISPATCH

At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters. STATIC DISPATCH is the default.

Notes

Owner privileges: The owner is authorized to execute the function (EXECUTE privilege) and has the ability to grant these privileges to others. For more information, see “GRANT (function or procedure privileges)” on page 1340. For more information about ownership of the object, see “Authorization, privileges, and object ownership” on page 60.

Choosing data types for parameters: When you choose the data types of the input and output parameters for your function, consider the rules of promotion that can affect the values of the parameters. (See “Promotion of data types” on page 95). For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, using the following data types for parameters is recommended:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters: The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII or all EBCDIC.

Determining the uniqueness of functions in a schema: At the current server, the function signature of each function, which is the qualified function name combined with the number and data types of the input parameters, must be unique. If the function has more than 30 input parameters, only the data types of the first 30 are used to determine uniqueness. This means that two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a single schema must not contain multiple functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, DB2 does not consider any length, precision, scale, subtype or encoding scheme attributes in the comparison. DB2 considers the synonyms of data types (DECIMAL and NUMERIC, REAL and FLOAT, and DOUBLE and FLOAT) a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), DECIMAL(4,3), or DECFLOAT(16) and DECFLOAT(34).

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15)) ...  
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...  
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...  
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Character string representation considerations: The PARAMETER VARCHAR clause is specific to LANGUAGE C functions because of the native use of NUL-terminated strings in C. VARCHAR structure representation is useful when

character string data is known to contain embedded NUL-terminators. It is also useful when it cannot be guaranteed that character string data does not contain embedded NUL-terminators.

PARAMETER VARCHAR does not apply to fixed length character strings, VARCHAR FOR BIT DATA, CLOB, DBCLOB, or implicitly generated parameters. The clause does not apply to VARCHAR FOR BIT DATA because BIT DATA can contain X'00' characters, and its value representation starts with length information. It does not apply to LOB data because a LOB value representation starts with length information.

PARAMETER VARCHAR does not apply to optional parameters that are implicitly provided to an external function. For example, a CREATE FUNCTION statement for LANGUAGE C must also specify PARAMETER STYLE SQL, which returns an SQLSTATE NUL-terminated character string; that SQLSTATE will not be represented in VARCHAR structured form. Likewise, none of the parameters that represent the qualified name of the function, the specific name of the function, or the SQL diagnostic string that is returned to the database manager will be represented in VARCHAR structured form.

Considerations for accessing message tokens and DBINFO: DB2 returns system-generated parameters from a routine, such as message tokens and DBINFO. The message tokens and DBINFO are character string data. The CCSID for system-generated string parameters is determined from the CCSID that is in effect for string parameters that are defined for the routine. If the parameter list for the routine does not include any character or graphic string parameters, the CCSID for system-generated string parameters is determined from the PARAMETER CCSID option that is in effect for the routine. For example, with a Unicode database, you can specify PARAMETER CCSID EBCDIC to have the system-generated string parameters returned to the invoking application in EBCDIC.

Overriding a built-in function: Giving an external function the same name as a built-in function is not a recommended practice unless you are trying to change the functionality of the built-in function.

If you do intend to create an external function with the same name as a built-in function, be careful to maintain the uniqueness of its function signature. If your function has the same name and data types of the corresponding parameters of the built-in function but implements different logic, DB2 might choose the wrong function when the function is invoked with an unqualified function name. Thus, the application might fail, or perhaps even worse, run successfully but provide an inappropriate result.

Running external functions in WLM environments: You can use the WLM ENVIRONMENT clause to identify the address space in which a function or is to run. Using different WLM environments lets you isolate one group of programs from another. For example, you might choose to isolate programs based on security requirements and place all payroll applications in one WLM environment because those applications deal with data, such as employee salaries.

To prevent a user from defining functions in sensitive WLM environments, DB2 invokes the external security manager to determine whether the user has authorization to issue CREATE FUNCTION statements that refer to the specified WLM environment. The following example shows the RACF command that authorizes DB2 user DB2USER1 to register a function on DB2 subsystem DB2A that runs in the WLM environment named PAYROLL.

```
PERMIT DB2A.WLMENV.PAYROLL CLASS(DSNR) ID(DB2USER1) ACCESS(READ)
```

Scrollable cursors specified with user-defined functions: A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NOT NULL CALL as a synonym for RETURNS NULL ON NULL INPUT
- NULL CALL as a synonym for CALLED ON NULL INPUT
- PARAMETER STYLE DB2SQL as a synonym for PARAMETER STYLE SQL

Examples

Example 1: Assume that you want to write an external function program in C that implements the following logic:

```
output = 2 * input - 4
```

The function should return a null value if and only if one of the input arguments is null. The simplest way to avoid a function call and get a null result when an input value is null is to specify RETURNS NULL ON NULL INPUT on the CREATE FUNCTION statement or allow it to be the default. Write the statement needed to register the function, using the specific name MINENULL1.

```
CREATE FUNCTION NTEST1 (SMALLINT)
  RETURNS SMALLINT
  EXTERNAL NAME 'NTESTMOD'
  SPECIFIC MINENULL1
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  FENCED
  PARAMETER STYLE SQL
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION;
```

Example 2: Assume that user Smith wants to register an external function named CENTER in schema SMITH. The function program will be written in C and will be reentrant. Write the statement that Smith needs to register the function, letting DB2 generate a specific name for the function.

```
CREATE FUNCTION CENTER (INTEGER, FLOAT)
  RETURNS FLOAT
  EXTERNAL NAME 'MIDDLE'
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  FENCED
  PARAMETER STYLE SQL
  NO EXTERNAL ACTION
  STAY RESIDENT YES;
```

Example 3: Assume that user McBride (who has administrative authority) wants to register an external function named CENTER in the SMITH schema. McBride plans to give the function specific name FOCUS98. The function program uses a scratchpad to perform some one-time only initialization and save the results. The

function program returns a value with a FLOAT data type. Write the statement McBride needs to register the function and ensure that when the function is invoked, it returns a value with a data type of DECIMAL(8,4).

```
CREATE FUNCTION SMITH.CENTER (FLOAT, FLOAT, FLOAT)
  RETURNS DECIMAL(8,4) CAST FROM FLOAT
  EXTERNAL NAME 'CMOD'
  SPECIFIC FOCUS98
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  FENCED
  PARAMETER STYLE SQL
  NO EXTERNAL ACTION
  SCRATCHPAD
  NO FINAL CALL;
```

Example 4: The following example registers a Java user-defined function that returns the position of the first vowel in a string. The user-defined function is written in Java, is to be run fenced, and is the FINDVWL method of class JAVAUDFS.

```
CREATE FUNCTION FINDV (CLOB(100K))
  RETURNS INTEGER
  FENCED
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'JAVAUDFS.FINDVWL'
  NO EXTERNAL ACTION
  CALLED ON NULL INPUT
  DETERMINISTIC
  NO SQL;
```

CREATE FUNCTION (external table)

This CREATE FUNCTION statement registers a user-defined external table function with a database server. A user-defined external table function can be used in the FROM clause of a subselect. It returns a table to the subselect by returning one row at a time each time it is invoked.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Additional privileges are required if the function uses a table as a parameter, refers to a distinct type, or is to run in a WLM (workload manager) environment. These privileges are:

- The SELECT privilege on any table that is an input parameter to the function.
- The USAGE privilege on each distinct type that the function references.
- Authority to create programs in the specified WLM environment. This authorization is obtained from an external security product, such as RACF.

Syntax

```

>> CREATE FUNCTION function-name (
    [ parameter-declaration ]
)

```

```

>> RETURNS TABLE (
    (1) [ column-name data-type
        (2) AS LOCATOR ]
) option-list

```

Notes:

- 1 This clause and the clauses that follow in the *option-list* can be specified in any order.
- 2 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

parameter-declaration:

```

>> [ parameter-name ] parameter-type

```

parameter-type:

```

>> data-type
    [ AS LOCATOR (1) ]
    TABLE LIKE [ table-name AS LOCATOR
                 view-name ]

```

Notes:

- 1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

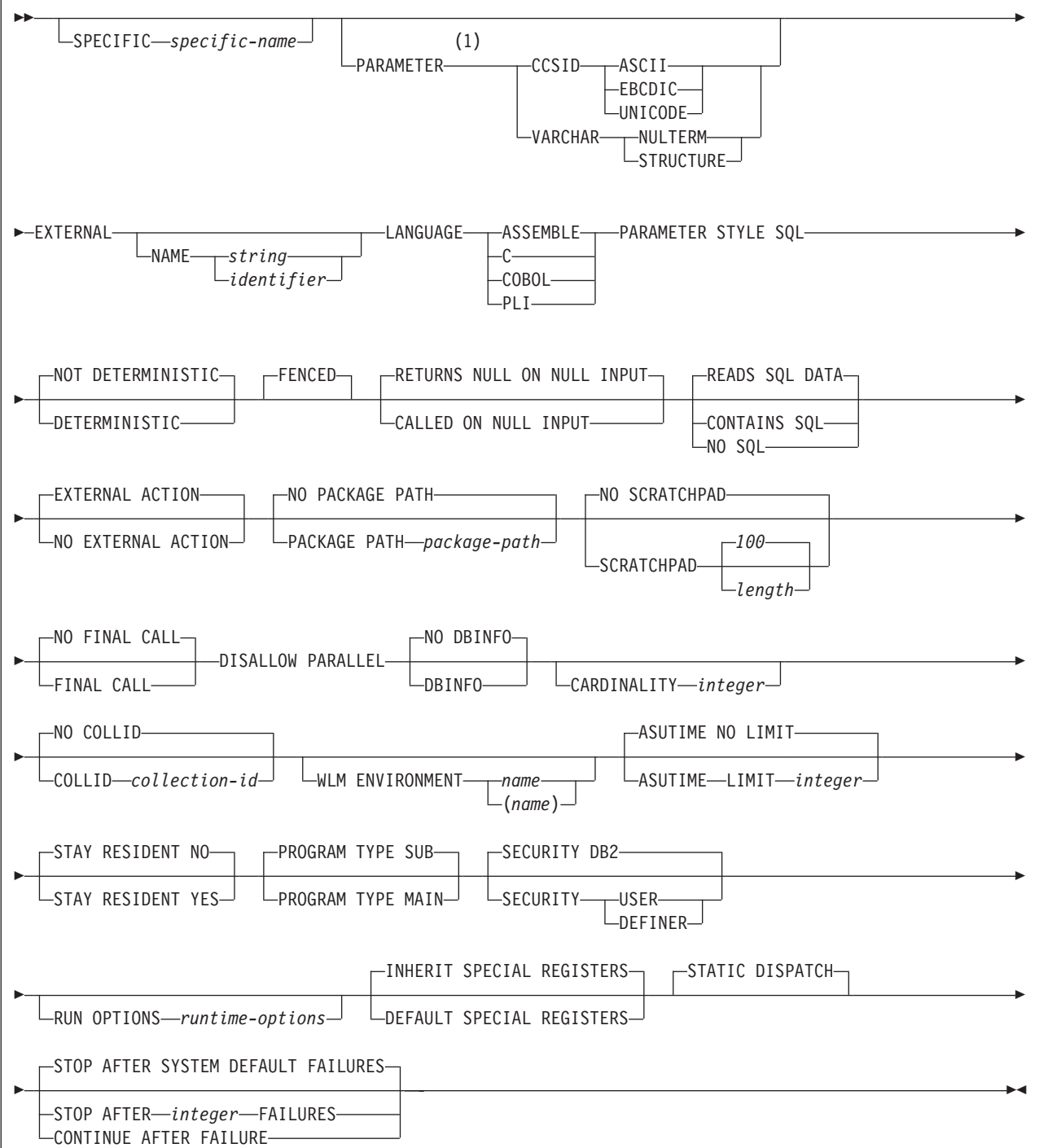
data-type:

```

>> [ built-in-type
    distinct-type-name ]

```

built-in-type:



Notes:

- 1 The same clause must not be specified more than one time.

Description

function-name
Names the user-defined function. The name is implicitly or explicitly qualified by a schema name.

The combination of name, schema name, the number of parameters, and the data type of each parameter²⁶ (without regard for any length, precision, scale, subtype or encoding scheme attributes of the data type) must not identify a user-defined function that exists at the current server.

You can use the same name for more than one function if the function signature of each function is unique.

- The unqualified form of *function-name* must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

ALL	LIKE	UNIQUE
AND	MATCH	UNKNOWN
ANY	NOT	=
BETWEEN	NULL	≠
DISTINCT	ONLY	<
EXCEPT	OR	<=
EXISTS	OVERLAPS	≠
FALSE	SIMILAR	>
FOR	SOME	>=
FROM	TABLE	→
IN	TRUE	<>
IS	TYPE	

The schema name can be 'SYSTOOLS' or 'SYSFUN' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(parameter-declaration,...)

Identifies the number of input parameters of the function, and specifies the data type of each parameter. All of the parameters for a function are input parameters and are nullable. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

parameter-name

Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name. The same name cannot be used for a parameter name and a column name.

data-type

Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the input parameter is a built-in data type.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length,

26. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

If you specify the name of the distinct type without a schema name, DB2 resolves the schema name by searching the schemas in the SQL path.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked. An error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

- A distinct type parameter is passed as the source type of the distinct type.

AS LOCATOR

Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type that is based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

TABLE LIKE *table-name* or *view-name* AS LOCATOR

Specifies that the parameter is a transition table. However, when the function is invoked, the actual values in the transition table are not passed to the function. A single value is passed instead. This single value is a locator to the table, which the function uses to access the columns of the transition table. A function with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACE. The number of columns and the attributes of those columns are determined at the time the CREATE FUNCTION statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the function.

table-name or *view-name* must identify a table or view that exists at the current server. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the function cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE FUNCTION statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.
- If the CREATE FUNCTION statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the function is invoked, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length, precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE FUNCTION statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE FUNCTION statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

RETURNS TABLE(*column-name data-type ...*)

Identifies that the output of the function is a table. The parentheses that follow the keyword enclose the list of names and data types of the columns of the table.

column-name

Specifies the name of the column. The name is an SQL identifier and must be unique within the RETURNS TABLE clause for the function.

data-type

Specifies the data type of the column. The column is nullable.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only for a LOB data type or a distinct type based on a LOB data type.

SPECIFIC *specific-name*

Specifies a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

SQLxxxxxxxxxxx

where 'xxxxxxxxxxx' is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

PARAMETER CCSID or VARCHAR

CCSID

Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or RETURNS TABLE clause, or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

VARCHAR

Specifies that the representation of the values of varying length character string-parameters, including, if applicable, the output of the function, for functions which specify LANGUAGE C.

This option can only be specified if LANGUAGE C is also specified.

NULTERM

Specifies that variable length character string parameters are represented in a NUL-terminated string form.

STRUCTURE

Specifies that variable length character string parameters are represented in a VARCHAR structure form.

Using the PARAMETER VARCHAR clause, there is no way to specify the VARCHAR form of an individual parameter as there is with PARAMETER CCSID. The PARAMETER VARCHAR clause only applies to parameters in the parameter list of a function and in the RETURNS TABLE clause. It does not apply to system-generated parameters of the routine such as message tokens and DBINFO.

In a data sharing environment, you should not specify the `PARAMETER VARCHAR` clause until all members of the data sharing group support the clause. If some group members support this clause and others do not, and `PARAMETER VARCHAR` is specified in an external routine, the routine will encounter different parameter forms depending on which group member invokes the routine.

EXTERNAL

Specifies that the function being registered is based on code that is written in an external programming language and adheres to the documented linkage conventions and interface of that language.

If you do not specify the `NAME` clause, `'NAME function-name'` is implicit. In this case, *function-name* must not be longer than 8 characters.

NAME *string or identifier*

Identifies the name of the load module that contains the user-written code that implements the logic of the function.

For other values of `LANGUAGE`, the name can be a string constant that is no longer than 8 characters. It must conform to the naming conventions for load modules. Alphabetical extenders for national languages can be used as the first character and as subsequent characters in the load module name.

DB2 loads the load module when the function is invoked. The load module is created when the program that contains the function body is compiled and link-edited. The load module does not need to exist when the `CREATE FUNCTION` statement is executed. However, it must exist and be accessible by the current server when the function is invoked.

You can specify the `EXTERNAL` clause in one of the following ways:

```
EXTERNAL  
EXTERNAL NAME PKJVSP1  
EXTERNAL NAME 'PKJVSP1'
```

If you specify an external program name, you must use the `NAME` keyword. For example, this syntax is not valid:

```
EXTERNAL PKJVSP1
```

LANGUAGE

Specifies the application programming language in which the function program is written. All programs must be designed to run in IBM's Language Environment environment.

ASSEMBLE

The function is written in Assembler.

C The function is written in C or C++. The `VARCHAR` clause can only be specified if `LANGUAGE C` is specified.

COBOL

The function is written in COBOL, including the object-oriented language extensions.

PLI

The function is written in PL/I.

PARAMETER STYLE SQL

Specifies the linkage convention that the function program uses to receive input parameters from and pass return values to the invoking SQL statement.

PARAMETER STYLE SQL specifies the parameter passing convention that supports passing null values both as input and for output. The parameters that are passed between the invoking SQL statement and the function include:

- n parameters for the input parameters that are specified for the function
- m parameters for the result columns of the function that are specified on the RETURNS TABLE clause
- n parameters for the indicator variables for the input parameters
- m parameters for the indicator variables of the result columns of the function that are specified on the RETURNS TABLE clause
- The SQLSTATE to be returned to DB2
- The qualified name of the function
- The specific name of the function
- The SQL diagnostic string to be returned to DB2
- The scratchpad, if SCRATCHPAD is specified
- The call type
- The DBINFO structure, if DBINFO is specified

For complete details about the structure of the parameter list that is passed, see *DB2 Application Programming and SQL Guide*.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC

The function might not return the same results each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT and SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers, or any function that contains SQL statements.

NOT DETERMINISTIC is the default.

DETERMINISTIC

The function always returns the same result each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function.

DETERMINISTIC is not the default. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

FENCED

Specifies that the function runs in an external address space to prevent the function from corrupting DB2 storage.

FENCED is the default.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON NULL INPUT

The function is not called if any of the input arguments is null. The result is an empty table, which is a table with no rows. RETURNS NULL ON INPUT is the default.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null argument values. The function can return an empty table, depending on its logic.

READS SQL DATA, CONTAINS SQL, or NO SQL

Specifies which SQL statements, if any, can be executed in the function or any routine that is called from this function. The default is READS SQL DATA. For the data access classification of each statement, see Table 138 on page 1605.

READS SQL DATA

Specifies that the function can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the function can execute only SQL statements with an access indication of CONTAINS SQL or NO SQL. The function cannot execute statements that read or modify data.

NO SQL

Specifies that the function can execute only SQL statements with a data access classification of NO SQL.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

Because DB2 uses the RRS attachment for functions, DB2 can participate in two-phase commit with any other resource manager that uses RRS. For resource managers that do not use RRS, there is no coordination of commit or rollback operations on non-DB2 resources.

EXTERNAL ACTION

The function can take an action that changes the state of an object that DB2 does not manage.

If you specify EXTERNAL ACTION, DB2:

- Materializes the views and table expressions in SELECT and SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify EXTERNAL ACTION if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

EXTERNAL ACTION is the default.

NO EXTERNAL ACTION

The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function. NO EXTERNAL ACTION is not the default. If applicable, specify NO EXTERNAL ACTION to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

NO PACKAGE PATH or PACKAGE PATH *package-path*

Specifies the packagepath to use when the function is run. This is the list of the possible package collections into which the DBRM this is associated with the function is bound.

NO PACKAGE PATH

Specifies that the list of package collections for the function is the same as the list of package collection IDs for the program that invokes the function. If the program that invokes the function does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*

Specifies a list of package collections, in the same format as the SET CURRENT PACKAGE PATH special register.

If the COLLID clause is specified with PACKAGE PATH, the COLLID clause is ignored when the function is invoked.

The *package-path* value that is provided when the function is created is checked when the function is invoked. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO SCRATCHPAD or SCRATCHPAD

Specifies whether DB2 provides a scratchpad for the function. It is strongly recommended that functions be reentrant, and a scratchpad provides an area for the function to save information from one invocation to the next.

NO SCRATCHPAD

Specifies that a scratchpad is not allocated and passed to the function. NO SCRATCHPAD is the default.

SCRATCHPAD *length*

Specifies that when the function is invoked for the first time, DB2 allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.
- DB2 initializes the scratchpad to all binary zeros (X'00's).
- The scope of a scratchpad is the SQL statement. Each reference to the function in an SQL statement has a scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, two scratchpads are allocated for the two references to UDFX in the following SQL statement:

```
SELECT *  
FROM TABLE (UDFX(A)), TABLE (UDFX(B));
```

- The scratchpad is persistent. DB2 preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. DB2

initializes the scratchpads when it begins to execute an SQL statement. DB2 does not reset scratchpads when a correlated subquery begins to execute.

- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify **FINAL CALL** to ensure that DB2 calls the function one more time so that the function can free those system resources.

Each time the function invoked, DB2 passes an additional argument to the function that contains the address of the scratchpad.

If you specify **SCRATCHPAD**, DB2:

- Does not move the function from one task control block (TCB) to another between **FETCH** operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared **WITH HOLD**.

NO FINAL CALL or FINAL CALL

Specifies whether a *first call* and a *final call* are made to the function.

NO FINAL CALL

A first call and final call are not made to the function. **NO FINAL CALL** is the default.

FINAL CALL

A first call and final call are made to the function in addition to one or more *open*, *fetch*, or *close calls*.

The types of calls are:

First call

A *first call* occurs only if the function was defined with **FINAL CALL**. Before a first call, the scratchpad is set to binary zeros. Argument values are passed to the function, and the function might acquire memory or perform other one-time only resource initialization. However, the function should not return any data to DB2, but it can set return values for the SQL-state and diagnostic-message arguments.

Open call

An *open call* occurs unless the function returns an error. The scratchpad is set to binary zeros only if the function was defined with **NO FINAL CALL**. Argument values are passed to the function, and the function might perform any one-time initialization actions that are required. However, the function should not return any data to DB2.

Fetch call

A *fetch call* occurs unless the function returns an error during the first call or open call. Argument values are passed to the function, and DB2 expects the function to return a row of data or the end-of-table condition. If a scratchpad is also passed to the function, it remains untouched from the previous call.

Close call

A *close call* occurs unless the function returns an error during the first call, open call, or fetch call. No SQL-argument or SQL-argument-ind values are passed to the function, and if the function attempts to

examine these values, unpredictable results might occur. If a scratchpad is also passed to the function, it remains untouched from the previous call.

The function should not return any data to DB2, but it can set return values for the SQL-state and diagnostic-message arguments. Also on close call, a function that is defined with NO FINAL CALL should release any system resources that it acquired. (A function that is defined with FINAL CALL should release any acquired resources on the final call.)

Final The *final call* balances the first call, and like the first call, occurs only if the function was defined with FINAL CALL. The function can set return values for the SQL-state and diagnostic-message arguments. The function should also release any system resources that it acquired. A final call occurs at these times:

- *End of statement*: When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of transaction*: When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

If a commit, rollback, or abort operation causes the final call, the function cannot issue any SQL statements when it is invoked.

DISALLOW PARALLEL

Specifies that DB2 does not consider parallelism for the function.

NO DBINFO or DBINFO

Specifies whether additional status information is passed to the function when it is invoked.

NO DBINFO

No additional information is passed. NO DBINFO is the default.

DBINFO

An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application runtime authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

CARDINALITY integer

Specifies an estimate of the expected number of rows that the function returns. The number is used for optimization purposes. The value of *integer* must range from 0 to 2147483647.

If you do not specify CARDINALITY, DB2 assumes a finite value. The finite value is the same value that DB2 assumes for tables for which the RUNSTATS utility has not gathered statistics.

If a function has an infinite cardinality—the function never returns the “end-of-table” condition and always returns a row, then a query that requires the “end-of-table” to work correctly will need to be interrupted. Thus, avoid using such functions in queries that involve GROUP BY and ORDER BY.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is to be used when the function is executed. This is the package collection into which the DBRM that is associated with the function program is bound.

NO COLLID

The package collection for the function is the same as the package collection of the program that invokes the function. If a trigger invokes the function, the collection of the trigger package is used. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*

The name of the package collection that is to be used when the external is executed.

WLM ENVIRONMENT

Identifies the WLM (workload manager) application environment in which the function is to run. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the function runs in the WLM-established stored procedure address space that is specified at installation time.

name

The WLM environment in which the function must run. If another user-defined function or a stored procedure calls the function and that calling routine is running in an address space that is not associated with the WLM environment, DB2 routes the function request to a different address space.

(*name*,*)

When an SQL application program directly invokes the function, the WLM environment in which the function runs.

If another user-defined function or a stored procedure calls the function, the function runs in same environment that the calling routine uses. In this case, authorization to run the function in the WLM environment is not checked because the authorization of the calling routine suffices.

Users must have the appropriate authorization to execute functions in the specified WLM environment. For an example of a RACF command that provides this authorization, see *Running external functions in WLM environments*.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of the function can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a function, setting a limit can be helpful if the function gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the service units. NO LIMIT is the default.

LIMIT *integer*

The limit on the service units is a positive integer in the range of 1 to 2 147 483 647. If the function uses more service units than the specified value, DB2 cancels the function.

STAY RESIDENT

Specifies whether the load module for the function is to remain resident in memory when the function ends.

NO The load module is deleted from memory after the function ends. Use NO for non-reentrant functions. NO is the default.

YES

The load module remains resident in memory after the function ends. Use YES for reentrant functions.

PROGRAM TYPE

Specifies whether the function program runs as a main routine or a subroutine.

SUB

The function runs as a subroutine. SUB is the default.

MAIN

The function runs as a main routine.

SECURITY

Specifies how the function interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2

The function does not require an external security environment. If the function accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the WLM-established stored procedure address space.

DB2 is the default.

USER

An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the primary authorization ID of the process that invoked the function.

DEFINER

An external security environment should be established for the function. If the function accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the function.

RUN OPTIONS *runtime-options*

Specifies the Language Environment runtime options to be used for the function. You must specify *runtime-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any runtime options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Specifies that the values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a user-defined function in Table 37 on page 151.

DEFAULT SPECIAL REGISTERS

Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a user-defined function in Table 37 on page 151.

STATIC DISPATCH

At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters. STATIC DISPATCH is the default.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE

Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES

Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX. This is the default.

STOP AFTER *nn* FAILURES

Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE

Specifies that this routine should not be placed in a stopped state after any failure.

Notes

See “Notes” on page 935 for information about:

- Owner privileges
- Choosing data types for parameters
- Specifying the encoding scheme for parameters
- Determining the uniqueness of functions in a schema
- Character string representation considerations
- Overriding a built-in function
- Scrollable cursors specified with user-defined functions
- Restrictions on nesting

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NOT NULL CALL as a synonym for RETURNS NULL ON NULL INPUT
- NULL CALL as a synonym for CALLED ON NULL INPUT
- PARAMETER STYLE DB2SQL as a synonym for PARAMETER STYLE SQL

Example

The following example registers a table function written to return a row consisting of a single document identifier column for each known document in a text management system. The first parameter matches a given subject area and the second parameter contains a given string.

Within the context of a single session, the table function always returns the same table; therefore, it is defined as DETERMINISTIC. In addition, the DISALLOW PARALLEL keyword is added because table functions cannot operate in parallel.

Although the size of the output for DOCMATCH is highly variable, CARDINALITY 20 is a representative value and is specified to help DB2.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
    RETURNS TABLE (DOC_ID CHAR(16))
    EXTERNAL NAME ABC
    LANGUAGE C
    PARAMETER STYLE SQL
    NO SQL
    DETERMINISTIC
    NO EXTERNAL ACTION
    FENCED
    SCRATCHPAD
    FINAL CALL
    DISALLOW PARALLEL
    CARDINALITY 20;
```

CREATE FUNCTION (sourced)

This CREATE FUNCTION statement registers a user-defined function that is based on an existing scalar or aggregate function with a database server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

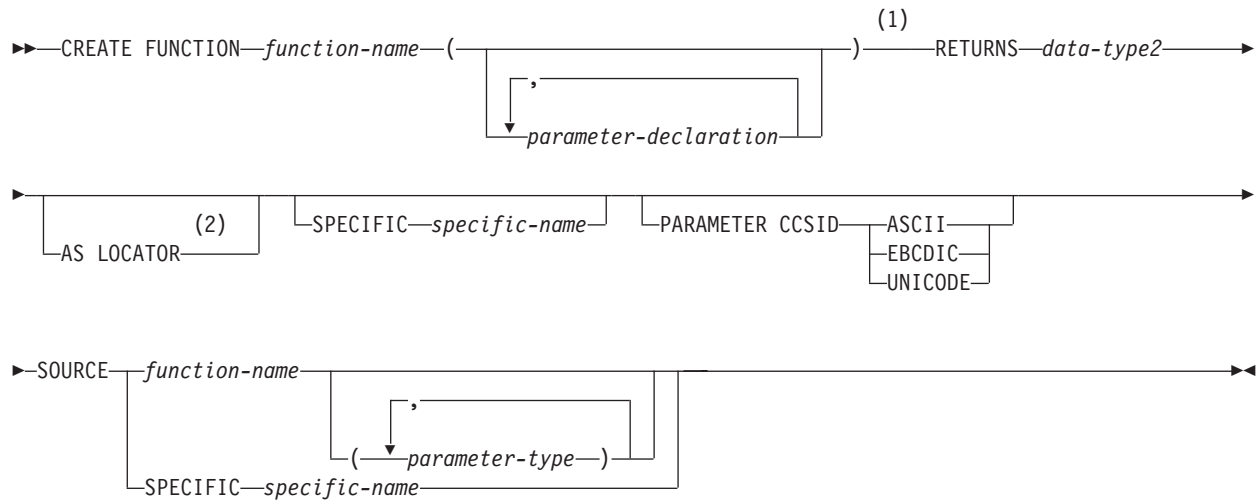
If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Additional privileges are required for the source function, and other privileges are also needed if the function uses a table as a parameter, or refers to a distinct type. These privileges are:

- The EXECUTE privilege for the function that the SOURCE clause references.
- The SELECT privilege on any table that is an input parameter to the function.
- The USAGE privilege on each distinct type that the function references.

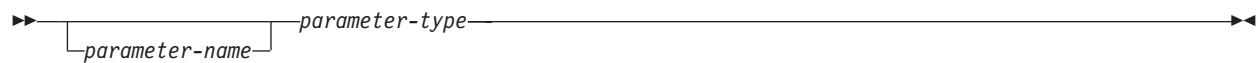
Syntax



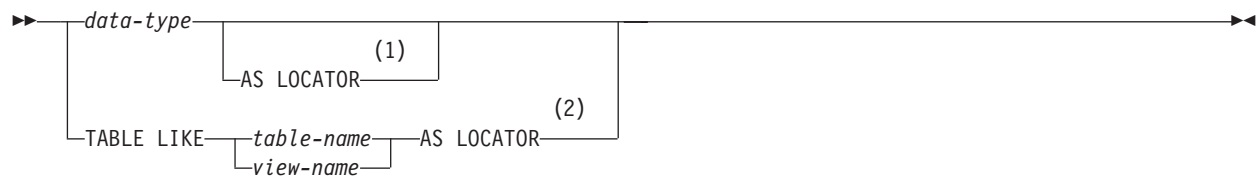
Notes:

- 1 RETURNS, SPECIFIC, and SOURCE can be specified in any order.
- 2 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

parameter-declaration:



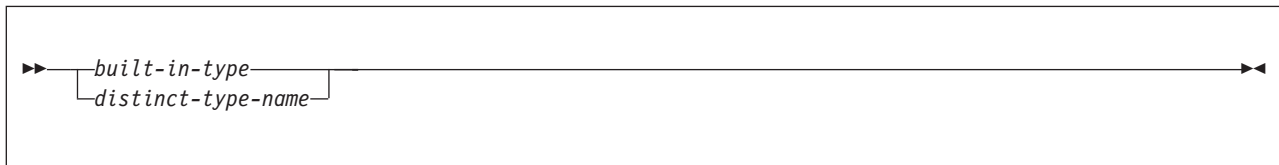
parameter-type:



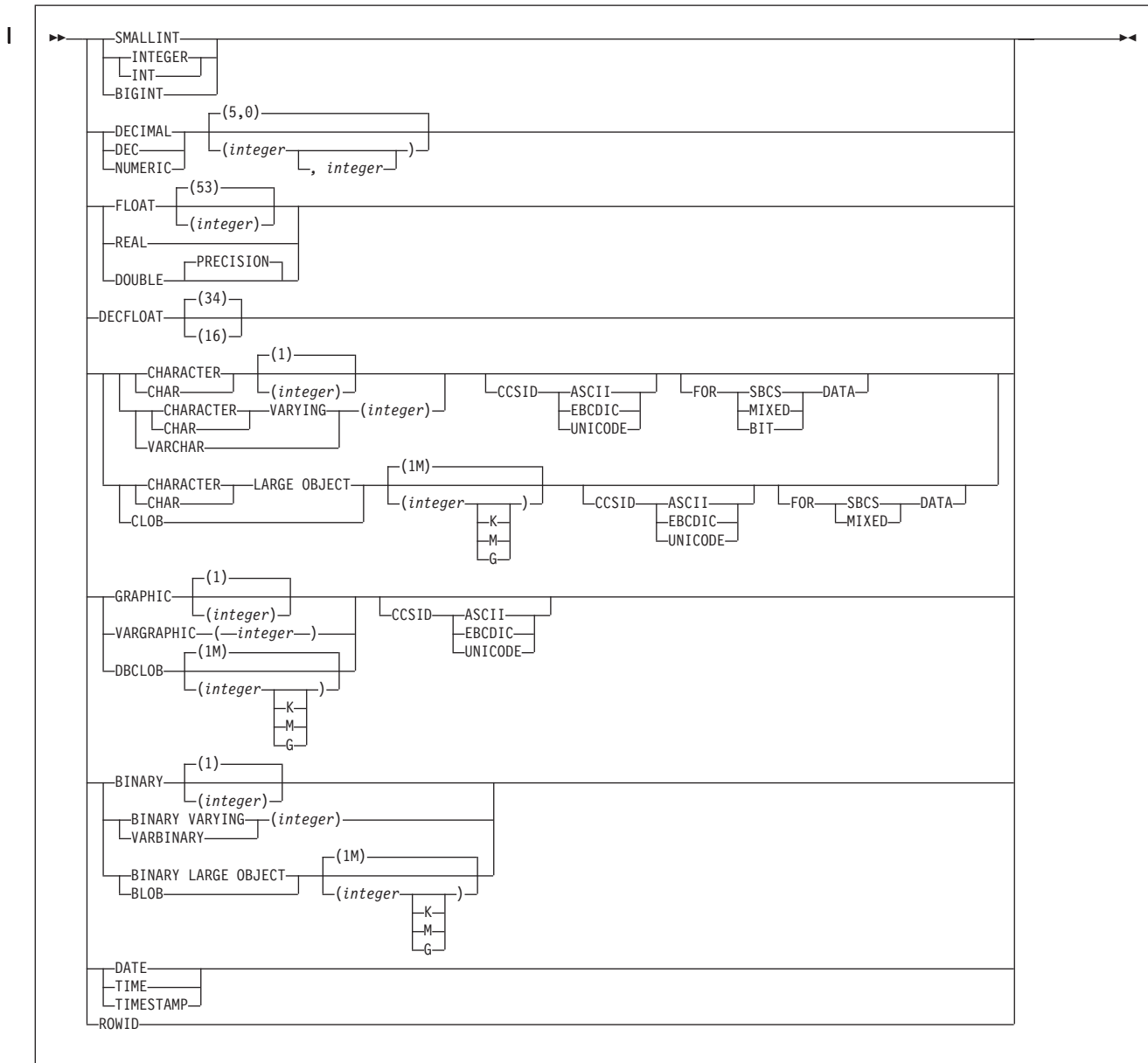
Notes:

- 1 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.
- 2 The TABLE LIKE name AS LOCATOR clause can only be specified for the parameter list of the function that is being defined.

data-type:



built-in-type:



Description

function-name

Names the user-defined function. The name is implicitly or explicitly qualified by a schema name.

The combination of name, schema name, the number of parameters, and the data type of each parameter²⁷ (without regard for any length, precision, scale, subtype or encoding scheme attributes of the data type) must not identify a user-defined function that exists at the current server.

If the function is sourced on an existing function to enable the use of the existing function with a distinct type, the name can be the same name as the existing function. In general, more than one function can have the same name if the function signature of each function is unique.

You can use the same name for more than one function if the function signature of each function is unique.

- The unqualified form of *function-name* must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

ALL	LIKE	UNIQUE
AND	MATCH	UNKNOWN
ANY	NOT	=
BETWEEN	NULL	≠
DISTINCT	ONLY	<
EXCEPT	OR	≤
EXISTS	OVERLAPS	≠
FALSE	SIMILAR	>
FOR	SOME	≥
FROM	TABLE	→
IN	TRUE	<>
IS	TYPE	

The schema name can be 'SYSTOOLS' or 'SYSFUN' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(*parameter-declaration*,...)

Specifies the number of input parameters of the function and the data type of each parameter. All of the parameters for a function are input parameters and are nullable. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

parameter-name

Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name.

data-type

Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the input parameter is a built-in data type.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding

27. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

- A distinct type parameter is passed as the source type of the distinct type.

You can specify any built-in data type or distinct type that matches or can be cast to the data type of the corresponding parameter of the source function (the function that is identified in the SOURCE clause). (For information on casting data types, see “Casting between data types” on page 96.) Length, precision, or scale attributes do not have to be specified for data types with these attributes. When specifying data types with these attributes, follow these rules:

- An empty set of parentheses can be used to indicate that the length, precision, or scale is the same as the source function.
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default values are used.

AS LOCATOR

Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

TABLE LIKE *table-name* or *view-name* AS LOCATOR

Specifies that the parameter is a transition table. However, when the function is invoked, the actual values in the transition table are not passed

to the function. A single value is passed instead. This single value is a locator to the table, which the function uses to access the columns of the transition table. A function with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACE. The number of columns and the attributes of those columns are determined at the time the CREATE FUNCTION statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the function.

table-name or *view-name* must identify a table or view that exists at the current server. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the function cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE FUNCTION statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.
- If the CREATE FUNCTION statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the function is invoked, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length, precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE FUNCTION statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE FUNCTION statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

RETURNS

Identifies the output of the function.

data-type2

Specifies the data type of the output. The output is nullable.

You can specify any built-in data type or distinct type that can be cast from the data type of the result of the source function. (For information on

casting data types, see “Casting between data types” on page 96.) For additional rules that apply to the data type that you can specify, see Rules for creating sourced functions.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only if the output from the function has a LOB data type or a distinct type based on a LOB data type.

SPECIFIC *specific-name*

Provides a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

SQLxxxxxxxxxxx

where 'xxxxxxxxxxx' is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

PARAMETER CCSID

Indicates whether the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or RETURNS clause, or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character and graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

SOURCE

Specifies that the new function is being defined as a sourced function. A *sourced function* is implemented by another function (the *source function*). The source function must be a scalar or aggregate function that exists at the current server, and it must be one of the following types of functions:

- A function that was defined with a CREATE FUNCTION statement
- A cast function that was generated by a CREATE TYPE statement
- A built-in function

If the source function is not a built-in function, the particular function can be identified by its name, function signature, or specific name.

If the source function is a built-in function, the SOURCE clause must include a function signature for the built-in function. The source function must not be any of the built-in functions shown in Table 94 (if a particular syntax is shown, only the indicated form cannot be specified).

Table 94. Built-in functions that cannot be the source function. When listed with specific conditions, the function cannot be a source function under those conditions. Otherwise, the function cannot be a source function regardless of its arguments.

Type of function	Restricted functions
Aggregate	COUNT(*) COUNT_BIG(*) XMLAGG

Table 94. Built-in functions that cannot be the source function (continued). When listed with specific conditions, the function cannot be a source function under those conditions. Otherwise, the function cannot be a source function regardless of its arguments.

Type of function	Restricted functions
Scalar function	<p>CHAR(<i>datetime-expression</i>, <i>second-argument</i>) where <i>second-argument</i> is ISO, USA, EUR, JIS, or LOCAL or if CHAR is specified with OCTETS, CODEUNITS16, or CODEUNITS32.</p> <p>CHARACTER_LENGTH</p> <p>CLOB if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>COALESCE</p> <p>DBCLOB if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>DECRYPT_BIT where second argument is DEFAULT</p> <p>DECRYPT_CHAR where second argument is DEFAULT</p> <p>DECRYPT_DB where second argument is DEFAULT</p> <p>EXTRACT</p> <p>GETVARIABLE where second argument is DEFAULT</p> <p>GRAPHIC if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>INSERT if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>LEFT if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>LOCAL</p> <p>LOCATE if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>MAX</p> <p>MIN</p> <p>NULLIF</p> <p>POSITION</p> <p>RID</p> <p>RIGHT if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>STRIP where multiple arguments are specified</p> <p>SUBSTRING</p> <p>VARCHAR if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>VARGRAPHIC if OCTETS, CODEUNITS16, or CODEUNITS32 is specified</p> <p>XMLCONCAT</p> <p>XMLELEMENT</p> <p>XMLFOREST</p> <p>XMLNAMESPACES</p>

If you base the sourced function directly or indirectly on an external scalar function, the sourced function inherits the attributes of the external scalar function. This can involve several layers of sourced functions. For example, assume that function A is sourced on function B, which in turn is sourced on function C. Function C is an external scalar function. Functions A and B inherit all of the attributes that are specified on the EXTERNAL clause of the CREATE FUNCTION statement for function C.

function-name
Identifies the function that is to be used as the source function. The source

function can be defined with any number of parameters. If more than one function is defined with the specified name in the specified or implicit schema, an error is returned.

If you specify an unqualified *function-name*, DB2 searches the schemas of the SQL path. DB2 selects the first schema that has only one function with this name on which the user has EXECUTE authority. An error is returned if a function is not found or a schema has more than one function with this name.

function-name (parameter-type,...)

Identifies the function that is to be used as the source function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. DB2 uses the number of data types and the logical concatenation of the data types to identify the specific function instance. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name()* is specified, the identified function must have zero parameters.

function-name

Identifies the function name of the source function. If you specify an unqualified name, DB2 searches the schemas of the SQL path. Otherwise, DB2 searches for the function in the specified schema.

parameter-type,...

Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

Empty parentheses are allowed for some data types that are specified in this context. For data types that have a length, precision, or scale attribute, use one of the following specifications:

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match. For example, DEC() is considered a match for a parameter of a function that is defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parentheses because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not need to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type

are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

If you omit the FOR *subtype* DATA clause or the CCSID clause for data types with a subtype or encoding scheme attribute, DB2 is to ignore the attribute when determining whether the data types match. An exception to ignoring the attribute is FOR BIT DATA. A character FOR BIT DATA parameter of the new function cannot correspond to a parameter of the source function that is not defined as character FOR BIT DATA. Likewise, a character parameter of the new function that is not FOR BIT DATA cannot correspond to a parameter of the source function that is defined as character FOR BIT DATA.

The number of input parameters in the function that is being created must be the same as the number of parameters in the source function. If the data type of each input parameter is not the same as or castable to the corresponding parameter of the source function, an error occurs. The data type of the final result of the source function must match or be castable to the result of the sourced function.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or distinct type that is based on a LOB.

SPECIFIC *specific-name*

Identifies the function to be used as the source function by its specific name.

If you specify an unqualified *specific-name*, DB2 searches the SQL path to locate the schema. DB2 selects the first schema that contains a function with this specific name for which the user has EXECUTE authority. DB2 returns an error if it cannot find a function with the specific name in one of the schemas in the SQL path.

If you specify a qualified *specific-name*, DB2 searches the named schema for the function. DB2 returns an error if it cannot find a function with the specific name.

Notes

Owner privileges: The owner is authorized to execute the function (EXECUTE privilege) in the following cases:

- If the underlying function is a user-defined function, and the owner is authorized with the grant option to execute the underlying function, the privilege on the new function includes the grant option. Otherwise, the owner can execute the new function but cannot grant others the privilege to do so.
- If the underlying function is a built-in function, the owner is authorized with the grant option to execute the underlying built-in function and the privilege on the new function includes the grant option.

For more information, see “GRANT (function or procedure privileges)” on page 1340. For more information about ownership of the object, see “Authorization, privileges, and object ownership” on page 60.

Choosing data types for parameters: When you choose the data types of the input and output parameters for your function, consider the rules of promotion that can

affect the values of the parameters. (See “Promotion of data types” on page 95). For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, we recommend using the following data types for parameters:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters: The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Determining the uniqueness of functions in a schema: At the current server, the function signature of each function, which is the qualified function name combined with the number and data types of the input parameters, must be unique. This means that two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a schema must not contain two functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, DB2 does not consider any length, precision, scale, subtype or encoding scheme attributes in the comparison. DB2 considers the synonyms of data types (DECIMAL and NUMERIC, REAL and FLOAT, and DOUBLE and FLOAT) a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), DECIMAL(4,3), or DECFLOAT(16) and DECFLOAT(34).

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15)) ...  
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...  
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...  
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Rules for creating sourced functions: For the discussion in this section, assume that the function that is being created is named NEWF and the source function is named SOURCEF. Consider the following rules when creating a sourced function:

- The unqualified names of the sourced function and source function can be different (NEWF and SOURCEF).
- The number of input parameters for NEWF and SOURCEF must be the same.

- When specifying the input parameters and output for NEWF, you can specify a value for the precision, scale, subtype, or encoding scheme for a data type with any of these attributes or use empty parentheses.

Empty parentheses, such as VARCHAR(), indicate that the value of the attribute is the same as the attribute for the corresponding parameter of SOURCEF, or that is determined by data type promotion. If you specify any values for the attributes, DB2 checks the values against the corresponding input parameters and returned output of SOURCEF as described next.

- When the CREATE FUNCTION statement is executed, DB2 checks the input parameters of NEWF against those of SOURCEF. The data type of each input parameter of NEWF function must be either the same as, or promotable to, the data type of the corresponding parameter of SOURCEF. (For information on the promotion of data types, see “Casting between data types” on page 96.)

This checking does not guarantee that an error will not occur when NEWF is invoked. For example, an argument that matches the data type and length or precision attributes of a NEWF parameter might not be promotable if the corresponding SOURCEF parameter has a shorter length or less precision. In general, do not define the parameters of a sourced function with length or precision attributes that are greater than the attributes of the corresponding parameters of the source function.

- When the CREATE FUNCTION statement is executed, DB2 checks the data type identified in the RETURNS clause of NEWF against the data type that SOURCEF returns. The data type that SOURCEF returns must be either the same as, or promotable to, the RETURNS data type of NEWF.

This checking does not guarantee that an error will not occur when NEWF is invoked. For example, the value of a result that matches the data type and length or precision attributes of those specified for SOURCEF's result might not be promotable if the RETURNS data type of NEWF has a shorter length or less precision. Consider the possible effects of defining the RETURNS data type of a sourced function with length or precision attributes that are less than the attributes defined for the data type returned by source function.

Scrollable cursors specified with user-defined functions: A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Examples

Example 1: Assume that you created a distinct type HATSIZE, which you based on the built-in data type INTEGER. You want to have an AVG function to compute the average hat size of different departments. Create a sourced function that is based on built-in function AVG.

```
CREATE FUNCTION AVE (HATSIZE) RETURNS HATSIZE
SOURCE SYSIBM.AVG (INTEGER);
```

When you created distinct type HATSIZE, two cast functions were generated, which allow HATSIZE to be cast to INTEGER for the argument and INTEGER to be cast to HATSIZE for the result of the function.

Example 2: After Smith registered the external scalar function CENTER in his schema, you decide that you want to use this function, but you want it to accept

two INTEGER arguments instead of one INTEGER argument and one FLOAT argument. Create a sourced function that is based on CENTER.

```
CREATE FUNCTION MYCENTER (INTEGER, INTEGER)
  RETURNS FLOAT
  SOURCE SMITH.CENTER (INTEGER, FLOAT);
```

CREATE FUNCTION (SQL scalar)

The CREATE FUNCTION (SQL Scalar) statement defines an SQL scalar function at the current server. The function returns a single value each time it is invoked.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

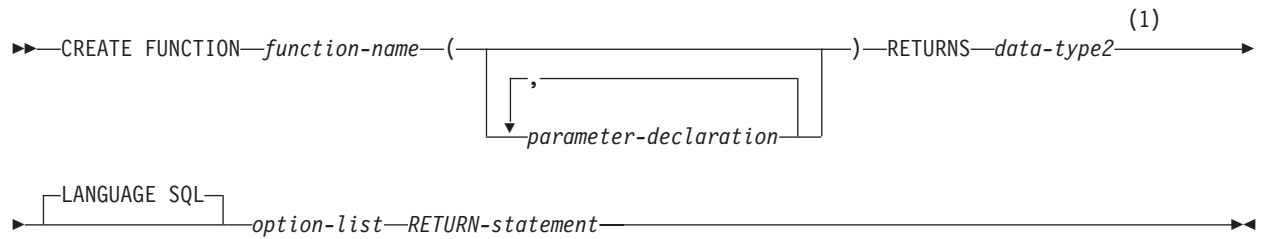
If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Additional privileges are required if the function uses a table as a parameter or refers to a distinct type. These privileges are:

- The SELECT privilege on any table that is an input parameter to the function.
- The USAGE privilege on each distinct type that the function references.

Syntax



Notes:

- 1 The `RETURNS` clause, the `RETURN-statement`, and the clauses in the `option-list` can be specified in any order. However, the same clause cannot be specified more than once.

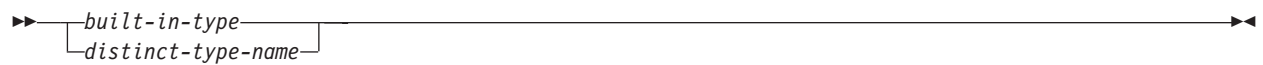
parameter-declaration:



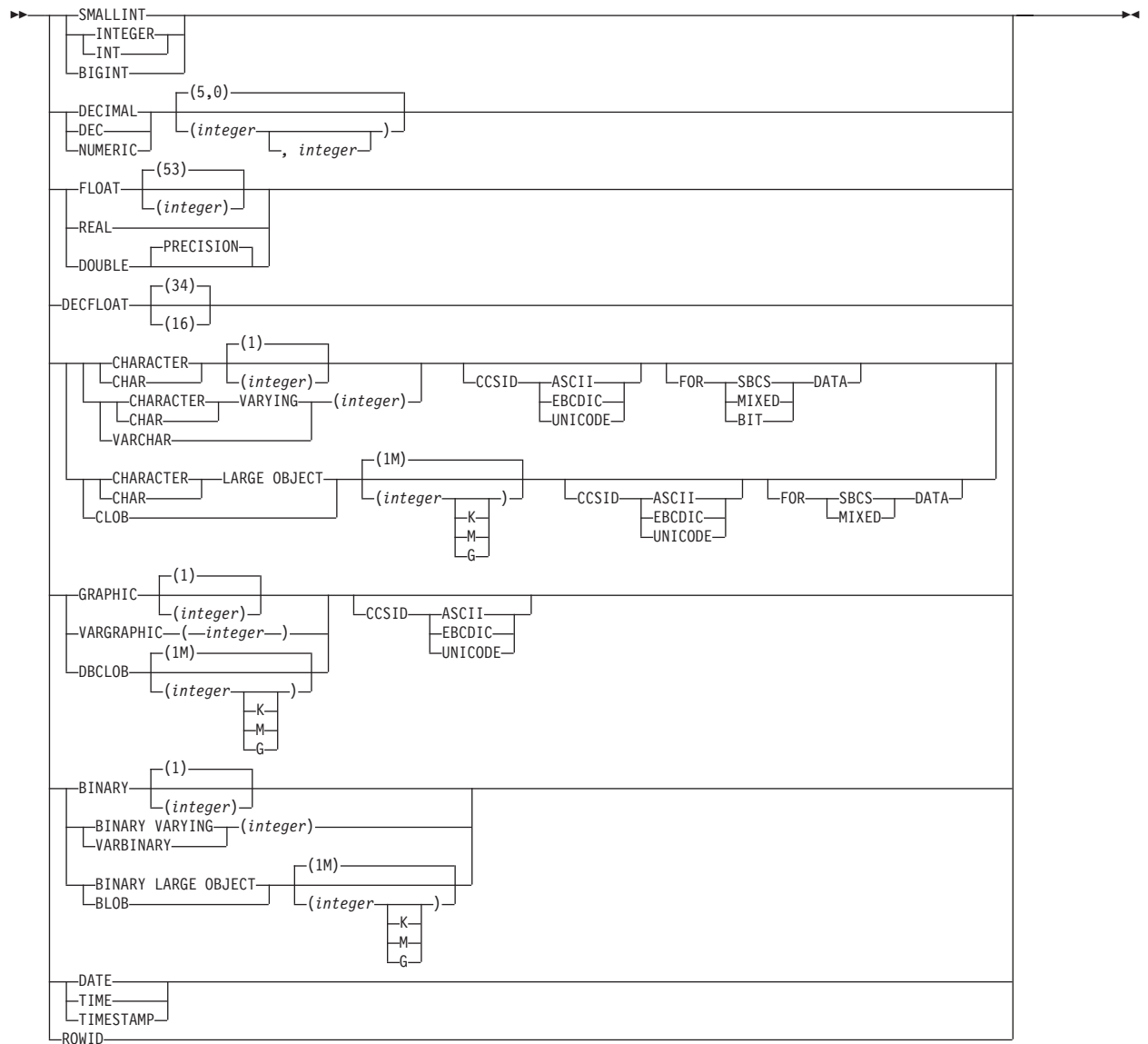
Notes:

- 1 Note that the `parameter-name` is required for SQL functions.

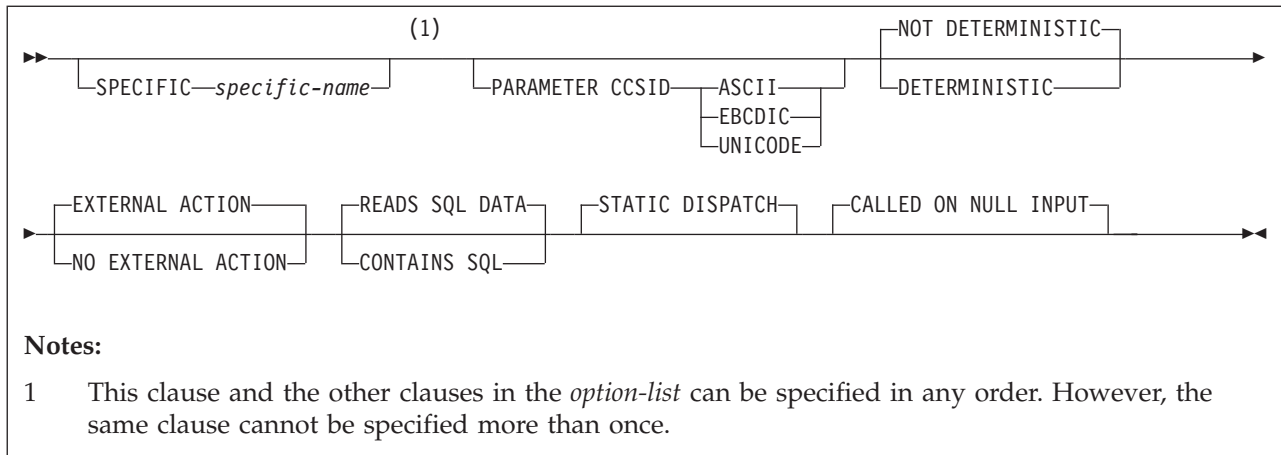
data-type:



built-in-type:



option-list:



Description

function-name

Names the user-defined function. The name is implicitly or explicitly qualified by a schema name. The combination of name, schema name, the number of parameters, and the data type of each parameter²⁸ (without regard for any length, precision, scale, subtype or encoding scheme attributes of the data type) must not identify a user-defined function that exists at the current server.

You can use the same name for more than one function if the function signature of each function is unique.

- The unqualified form of *function-name* is an SQL identifier.

The name must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

ALL	LIKE	UNIQUE
AND	MATCH	UNKNOWN
ANY	NOT	=
BETWEEN	NULL	≠
DISTINCT	ONLY	<
EXCEPT	OR	<=
EXISTS	OVERLAPS	≠<
FALSE	SIMILAR	>
FOR	SOME	>=
FROM	TABLE	→
IN	TRUE	<>
IS	TYPE	

The schema name can be 'SYSTOOLS' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(parameter-declaration,...)

Identifies the number of input parameters of the function, and specifies the name and data type of each parameter. All of the parameters for a function are input parameters. There must be one entry in the list for each parameter that the function expects to receive. A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

28. If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

parameter-name

Specifies the name of the input parameter. The name is an SQL identifier, and each name in the parameter list must not be the same as any other name.

data-type

Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the input parameter is a built-in data type.

For information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

If you specify the name of the distinct type without a schema name, DB2 resolves the schema name by searching the schemas in the SQL path.

The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type must be the same—either all ASCII, all EBCDIC, or all UNICODE.

Although parameters with a character data type have an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the function program can receive character data of any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the function is invoked. An error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

- A distinct type parameter is passed as the source type of the distinct type.

RETURNS

Identifies the output of the function.

data-type2

Specifies the data type of the output. The output is nullable.

The same considerations that apply to the data type of input parameter, as described under data-type, apply to the data type of the output of the function.

LANGUAGE SQL

Specifies that the function is written exclusively in SQL.

SPECIFIC *specific-name*

Specifies a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function that exists at the current server.

The unqualified form of *specific-name* is an SQL identifier. The qualified form is an SQL identifier (the schema name) followed by a period and an SQL identifier.

If you do not specify a schema name, it is the same as the explicit or implicit schema name of the function name (*function-name*). If you specify a schema name, it must be the same as the explicit or implicit schema name of the function name.

If you do not specify the SPECIFIC clause, the default specific name is the name of the function. However, if the function name does not provide a unique specific name or if the function name is a single asterisk, DB2 generates a specific name in the form of:

SQLxxxxxxxxxxx

where 'xxxxxxxxxxx' is a string of 12 characters that make the name unique.

The specific name is stored in the SPECIFIC column of the SYSROUTINES catalog table. The specific name can be used to uniquely identify the function in several SQL statements (such as ALTER FUNCTION, COMMENT, DROP, GRANT, and REVOKE) and must be used in DB2 commands (START FUNCTION, STOP FUNCTION, and DISPLAY FUNCTION). However, the function cannot be invoked by its specific name.

PARAMETER CCSID

Indicates whether the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or RETURNS clause, or in the field DEF ENCODING SCHEME on installation panel DSNTPF.

This clause provides a convenient way to specify the encoding scheme for character and graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the function returns the same results each time that the function is invoked with the same input arguments.

NOT DETERMINISTIC

The function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. DB2 uses this information to disable the merging of views and table expressions when processing SELECT and SQL data change statements that refer to this function. An example of a function that is not deterministic is one that generates random numbers.

NOT DETERMINISTIC must be specified explicitly or implicitly if the function program accesses a special register or invokes another function that is not deterministic. NOT DETERMINISTIC is the default.

DETERMINISTIC

The function always returns the same result function each time that the function is invoked with the same input arguments. An example of a deterministic function is a function that calculates the square root of the input. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function. DETERMINISTIC is not the default. If applicable, specify DETERMINISTIC to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that DB2 does not manage. An example of an external action is sending a message or writing a record to a file.

EXTERNAL ACTION

The function can take an action that changes the state of an object that DB2 does not manage.

Some SQL statements that invoke functions with external actions can result in incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

If you specify EXTERNAL ACTION, then DB2:

- Materializes the views and table expressions in SELECT and SQL data change statements that refer to the function. This materialization can adversely affect the access paths that are chosen for the SQL statements that refer to this function. Do not specify EXTERNAL ACTION if the function does not have an external action.
- Does not move the function from one task control block (TCB) to another between FETCH operations.
- Does not allow another function or stored procedure to use the TCB until the cursor is closed. This is also applicable for cursors declared WITH HOLD.

The only changes to resources made outside of DB2 that are under the control of commit and rollback operations are those changes made under RRS control.

EXTERNAL ACTION must be specified implicitly or explicitly specified if the SQL routine body invokes a function that is defined with EXTERNAL ACTION. EXTERNAL ACTION is the default.

NO EXTERNAL ACTION

The function does not take any action that changes the state of an object that DB2 does not manage. DB2 uses this information to enable the merging of views and table expressions for SELECT and SQL data change statements that refer to this function. NO EXTERNAL ACTION is not the

default. If applicable, specify NO EXTERNAL ACTION to prevent non-optimal access paths from being chosen for SQL statements that refer to this function.

DB2 does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

READS SQL DATA or CONTAINS SQL

Specifies which SQL statements, if any, can be executed in the function or any routine that is called from this function. The default is READS SQL DATA. For the data access classification of each statement, see Table 138 on page 1605.

READS SQL DATA

Specifies that the function can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the function can execute only SQL statements with a data access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

STATIC DISPATCH

At function resolution time, DB2 chooses a function based on the static (or declared) types of the function parameters. **STATIC DISPATCH** is the default.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments are null, making the function responsible for testing for null arguments. The function can return null. **CALLED ON NULL INPUT**

RETURN-statement

Specifies the return value of the function. For description of the statement, see “RETURN statement” on page 1580.

Notes

Choosing data types for parameters: When you choose the data types of the input and output parameters for your function, consider the rules of promotion that can affect the values of the parameters. (See “Promotion of data types” on page 95). For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, using the following data types for parameters is recommended:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters: The implicitly or explicitly specified encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII or all EBCDIC.

Determining the uniqueness of functions in a schema: At the current server, the function signature of each function, which is the qualified function name combined with the number and data types of the input parameters, must be unique. If the function has more than 30 input parameters, only the data types of the first 30 are used to determine uniqueness. This means that two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a single schema must not contain multiple functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, DB2 does not consider any length, precision, scale, subtype or encoding scheme attributes in the comparison. DB2 considers the synonyms of data types (DECIMAL and NUMERIC, REAL and FLOAT, and DOUBLE and FLOAT) a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), DECIMAL(4,3), or DECFLOAT(16) and DECFLOAT(34).

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (A INT, B CHAR(15)) ...  
CREATE FUNCTION PART (A INTEGER, B CHAR(40)) ...  
CREATE FUNCTION ANGLE (A DECIMAL(12,2)) ...  
CREATE FUNCTION ANGLE (A DEC(10,7)) ...
```

Overriding a built-in function: Giving an SQL function the same name as a built-in function is not a recommended practice unless you are trying to change the functionality of the built-in function.

If you do intend to create an SQL function with the same name as a built-in function, be careful to maintain the uniqueness of its function signature. If your function has the same name and data types of the corresponding parameters of the built-in function but implements different logic, DB2 might choose the wrong function when the function is invoked with an unqualified function name. Thus, the application might fail, or perhaps even worse, run successfully but provide an inappropriate result.

Resolution of function invocations: DB2 resolves function invocations inside the body of the function according to the SQL path that is in effect for the CREATE FUNCTION statement. This SQL path does not change after the function is created.

Referencing date and time special registers: If an SQL function contains multiple references to any of the date or time special registers, all references return the same value. Further, this value is the same value returned by the register invocation in the statement that invoked the function.

Self-referencing function: The body of an SQL function (that is, the *expression* or NULL in the RETURN clause of the CREATE FUNCTION statement) cannot contain a recursive invocation of itself or to another function that invokes it, because such a function would not exist to be referenced.

Scrollable cursors specified with user-defined functions: A row can be fetched more than one time with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NOT NULL CALL as a synonym for RETURNS NULL ON NULL INPUT
- NULL CALL as a synonym for CALLED ON NULL INPUT

Examples

Example 1: Define a scalar function that returns the tangent of a value using existing SIN and COS built-in functions:

```
CREATE FUNCTION TAN (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X);
```

CREATE GLOBAL TEMPORARY TABLE

The CREATE GLOBAL TEMPORARY TABLE statement creates a description of a temporary table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privilege set that is defined below must include at least one of the following:

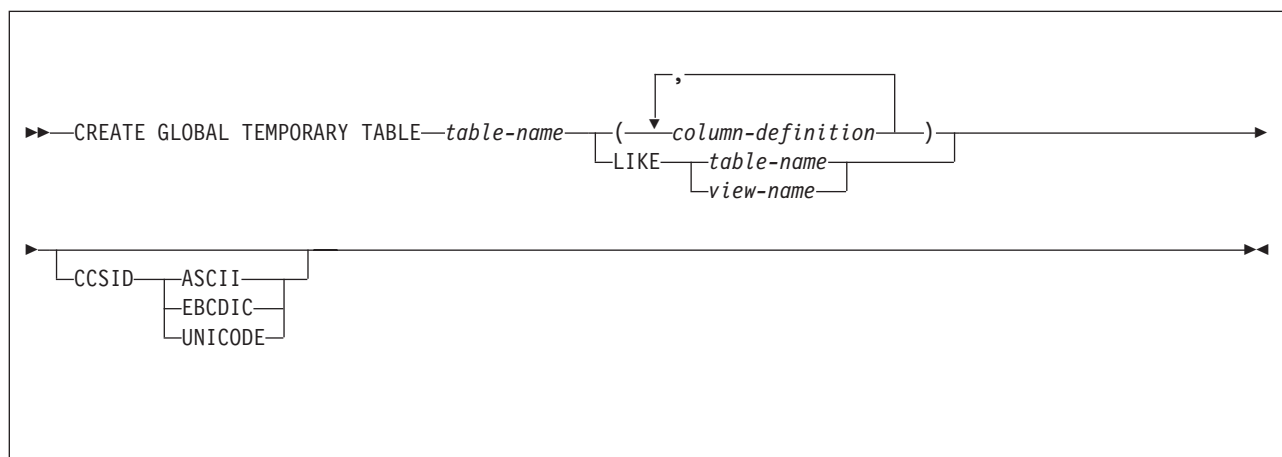
- The CREATETMTAB system privilege
- The CREATETAB database privilege for any database
- DBADM, DBCTRL, or DBMAINT authority for any database
- SYSADM or SYSCTRL authority

However, DABADM, DBCTRL, or DBMAINT authority is not sufficient authority if you are creating a temporary table for someone else and the table qualifier is not your authorization ID.

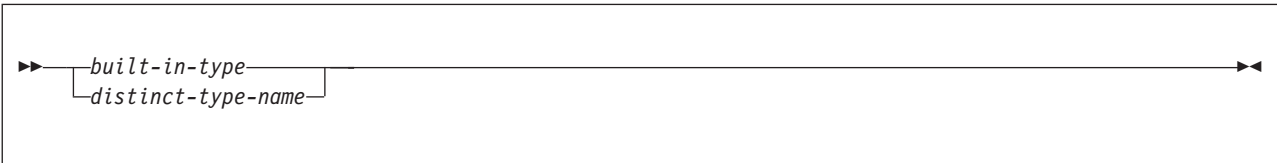
Additional privileges might be required when the data type of a column is a distinct type or the LIKE clause is specified. See the description of *distinct-type* and LIKE for the details.

Privilege set: The privilege set is the same as the privilege set for the CREATE TABLE statement. See Privilege Set for details.

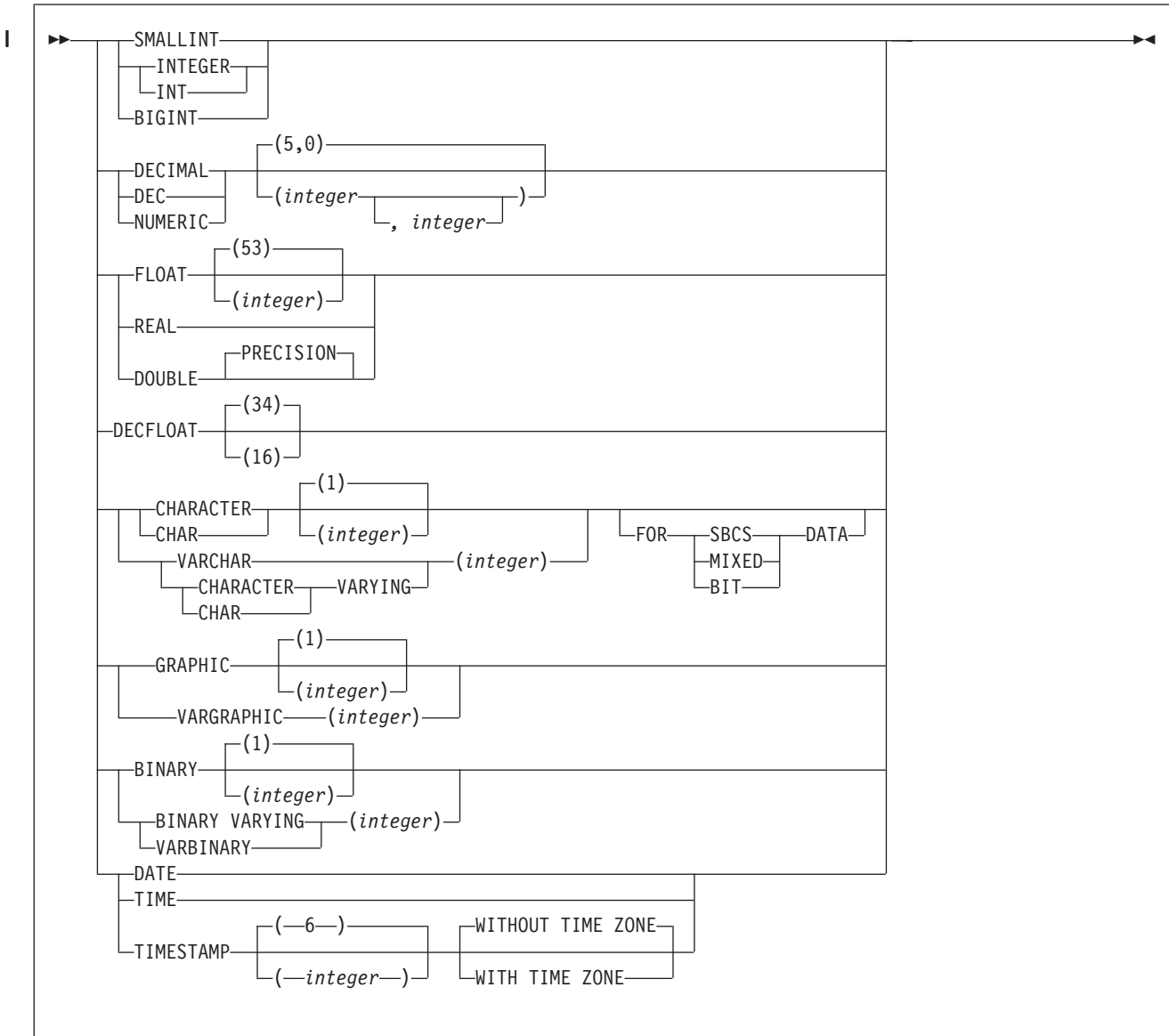
Syntax



data-type:



built-in-type:



Description

table-name
Names the temporary table. The name, including the implicit or explicit qualifier, must not identify a table, view, alias, synonym, or temporary table that exists at the database server.

The qualification rules for a temporary table are the same as for other tables.

The owner acquires ALL PRIVILEGES on the table WITH GRANT OPTION and the authority to drop the table.

column-definition

Defines the attributes of a column for each instance of the table. The number of columns defined must not exceed 750. The maximum record size must not exceed 32714 bytes. The maximum row size must not exceed 32706 bytes (8 bytes less than the maximum record size).

column-name

Names the column. The name must not be qualified and must not be the same as the name of another column in the table.

data-type

Specifies the data type of the column. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the column is a built-in data type.

For more information on and the rules that apply to the data types, see built-in-type.

distinct-type

Any distinct type except one that is based on a LOB or ROWID data type. The privilege set must implicitly or explicitly include the USAGE privilege on the distinct type.

NOT NULL

Specifies that the column cannot contain nulls. Omission of NOT NULL indicates that the column can contain nulls.

LIKE *table-name* **or** *view-name*

Specifies that the columns of the table have exactly the same name and description as the columns of the identified table or view. The name specified after LIKE must identify a table, view, or temporary table that exists at the current server. The privilege set must implicitly or explicitly include the SELECT privilege on the identified table or view.

This clause is similar to the LIKE clause on CREATE TABLE, but it has the following differences:

- If any column of the identified table or view has an attribute value that is not allowed for a column in a temporary table, that attribute value is ignored. The corresponding column in the new temporary table has the default value for that attribute unless otherwise indicated.
- If any column of the identified table or view allows a default value other than null, that default value is ignored and the corresponding column in the new temporary table has no default value. A default value other than null is not allowed for any column in a temporary table.

CCSID *encoding-scheme*

Specifies the encoding scheme for string data stored in the table.

ASCII Specifies that the data must be encoded by using the ASCII CCSIDs of the server.

An error occurs if a valid ASCII CCSID has not been specified for the installation.

EBCDIC

Specifies that data must be encoded by using the EBCDIC CCSIDs of the server.

An error occurs if a valid EBCDIC CCSID has not been specified for the installation.

UNICODE

Specifies that data must be encoded by using the CCSIDs of the server for Unicode.

An error occurs if a valid CCSID for Unicode has not been specified for the installation.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or Unicode data is used. An error occurs if CCSIDs have not been defined.

For the creation of temporary tables, the CCSID clause can be specified whether or not the LIKE clause is specified. If the CCSID clause is specified, the encoding scheme of the new table is the scheme that is specified in the CCSID clause. If the CCSID clause is not specified, the encoding scheme of the new table is the same as the scheme for the table specified in the LIKE clause.

Notes

Owner privileges: The owner of the table has all table privileges (see “GRANT (table or view privileges)” on page 1355) with the ability to grant these privileges to others. For more information about ownership of the object, see “Authorization, privileges, and object ownership” on page 60.

Instantiation and termination: Let T be a temporary table defined at the current server and let P denote an application process:

- An empty instance of T is created as a result of the first implicit or explicit reference to T in an OPEN, SELECT INTO or SQL data change operation that is executed by any program in P.
- Any program in P can reference T and any reference to T by a program in P is a reference to that instance of T.

When a commit operation terminates a unit of work in P and no program in P has an open WITH HOLD cursor that is dependent on T, the commit includes the operation DELETE FROM T.

- When a rollback operation terminates a unit of work in P, the rollback includes the operation DELETE FROM T.
- When the connection to the database server at which an instance of T was created terminates, the instance of T is destroyed. However, the definition of T remains. A DROP TABLE statement must be executed to drop the definition of T.

Restrictions and extensions: Let T denote a temporary table:

- Columns of T cannot have default values other than null.
- A column of T cannot have a LOB or ROWID data type (or a distinct type based on one).
- T cannot have unique constraints, referential constraints, or check constraints.
- T cannot be defined as the parent in a referential constraint.
- T cannot be referenced in:
 - A CREATE INDEX statement.

- A LOCK TABLE statement.
- As the object of an UPDATE statement in which the object is T or a view of T. However, you can reference T in the WHERE clause of an UPDATE statement (including the update operation of the MERGE statement).
- DB2 utility commands.

- As with all tables stored in a work file, query parallelism cannot be considered for any query that references T.
- If T is referenced in the *fullselect* of a CREATE VIEW statement, you cannot specify a WITH CHECK OPTION clause in the CREATE VIEW statement.
- ALTER TABLE T is valid only if the statement is used to add a column to T. Any column that you add to T must have a default value of null.
When you alter T, any plans and packages that refer to the table are invalidated, and DB2 automatically rebinds the plans and packages the next time they are run.
- DELETE FROM T or a *view of T* is valid only if the statement does not include a WHERE or WHERE CURRENT OF clause. In addition, DELETE FROM *view of T* is valid only if the view was created (CREATE VIEW) without the WHERE clause. A DELETE FROM statement deletes all the rows from the table or view.
- You can refer to T in the FROM clause of any subselect. If you refer to T in the first FROM clause of a *select-statement*, you cannot specify a FOR UPDATE clause.
- You cannot use a DROP DATABASE statement to implicitly drop T. To drop T, reference T in a DROP TABLE statement.
- A temporary table instantiated by an SQL statement using a three-part table name can be accessed by another SQL statement using the same name in the same application process for as long as the DB2 connection which established the instantiation is not terminated.
- GRANT ALL PRIVILEGES ON T is valid, but you cannot grant specific privileges on T.
Of the ALL privileges, only the ALTER, INSERT, DELETE, and SELECT privileges can actually be used on T.
- REVOKE ALL PRIVILEGES ON T is valid, but you cannot revoke specific privileges from T.
- A COMMIT operation deletes all rows of every temporary table of the application process, but the rows of T are not deleted if any program in the application process has an open WITH HOLD cursor that is dependent on T. In addition, if RELEASE(COMMIT) is in effect and no open WITH HOLD cursors are dependent on T, all logical work files for T are also deleted.
- A ROLLBACK operation deletes all rows and all logical work files of every temporary table of the application process.
- You can reuse threads when using a temporary table, and a logical work file for a temporary table name remains available until deallocation. A new logical work file is not allocated for that temporary table name when the thread is reused.
- You can refer to T in the following statements:

ALTER FUNCTION	CREATE PROCEDURE	DECLARE TABLE
ALTER PROCEDURE	CREATE SYNONYM	DELETE (if it does not
COMMENT	CREATE TABLE LIKE	include a WHERE clause)
CREATE ALIAS	CREATE VIEW	DROP TABLE
CREATE FUNCTION	DESCRIBE TABLE	INSERT
		LABEL
		SELECT INTO

Alternative syntax and synonyms: For compatibility with previous releases of DB2, you can specify LONG VARCHAR as a synonym for VARCHAR(*integer*) and LONG VARGRAPHIC as a synonym for VARGRAPHIC(*integer*) when defining the data type of a column. However, the use of these synonyms is not encouraged because after the statement is processed, DB2 considers a LONG VARCHAR column to be VARCHAR and a LONG VARGRAPHIC column to be VARGRAPHIC.

Examples

Example 1: Create a temporary table, CURRENTMAP. Name two columns, CODE and MEANING, both of which cannot contain nulls. CODE contains numeric data and MEANING has character data. Assuming a value of NO for the field MIXED DATA on installation panel DSNTIPF, column MEANING has a subtype of SBCS:

```
CREATE GLOBAL TEMPORARY TABLE CURRENTMAP
  (CODE INTEGER NOT NULL, MEANING VARCHAR(254) NOT NULL);
```

Example 2: Create a temporary table, EMP:

```
CREATE GLOBAL TEMPORARY TABLE EMP
  (TMPDEPTNO  CHAR(3)      NOT NULL,
   TMPDEPTNAME VARCHAR(36) NOT NULL,
   TMPMGRNO   CHAR(6)      ,
   TMPLOCATION  CHAR(16)     );
```

CREATE INDEX

The CREATE INDEX statement creates a partitioning index or a secondary index and an index space at the current server. The columns included in the key of the index are columns of a table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The INDEX privilege on the table
- Ownership of the table
- DBADM authority for the database that contains the table
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If the index is created using an expression, the EXECUTE privilege is required on any user-defined function that is invoked in the index expression.

Additional privileges might be required, as explained in the description of the BUFFERPOOL and USING STOGROUP clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the specified index name includes a qualifier that is not the same as this owner, the privilege set must include SYSADM or SYSCTRL authority, or DBADM or DBCTRL authority for the database.

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, or SYSCTRL authority.

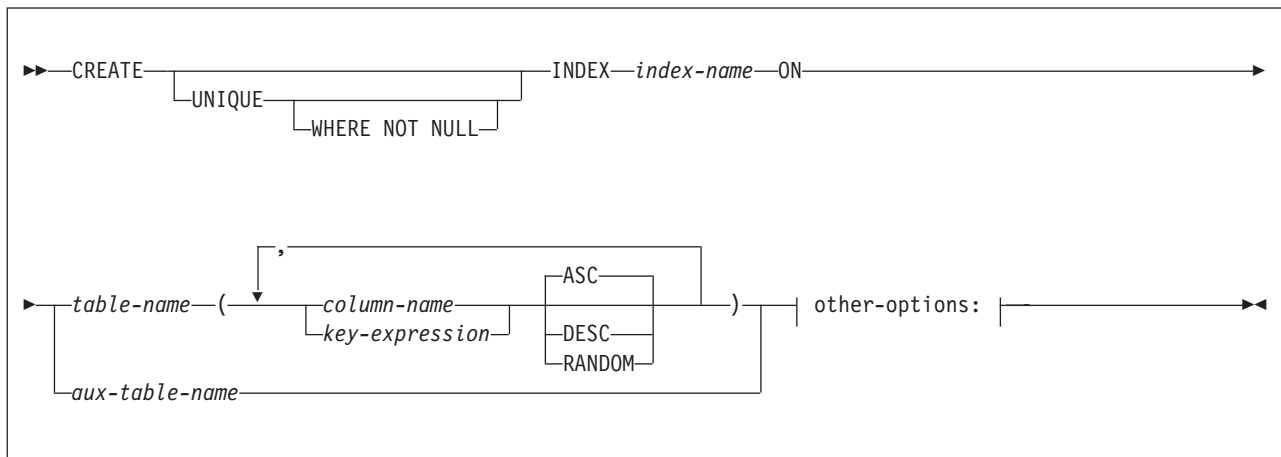
If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

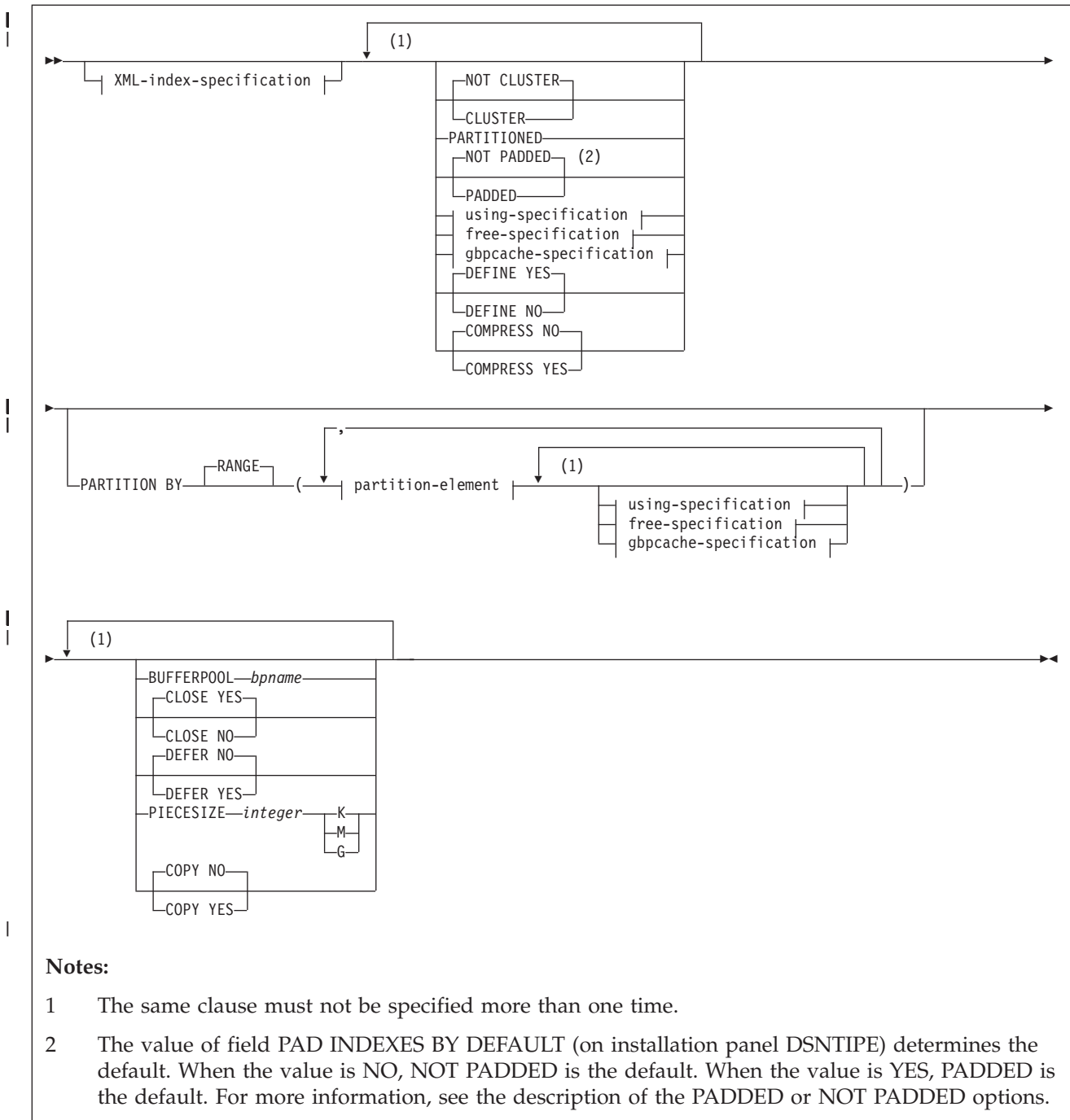
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privilege set is the set of privileges that are held by the role that is associated with the primary authorization ID of the process. However, if the specified index name includes a qualifier that is not the same as this authorization ID, the following rules apply:

- If the privilege set includes SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier can be any valid schema name.
- If the privilege set lacks SYSADM or SYSCTRL authority (or DBADM authority for the database, or DBCTRL authority for the database when creating a table), the schema qualifier is valid only if it is the same as one of the authorization IDs of the process and the privilege set that are held by that authorization ID includes all privileges needed to create the index. This is an exception to the rule that the privilege set is the privileges that are held by the SQL authorization ID of the process.

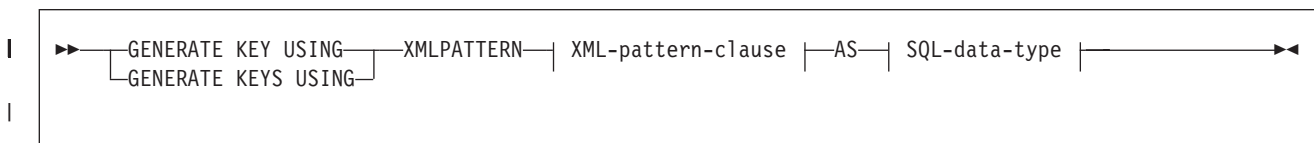
Syntax



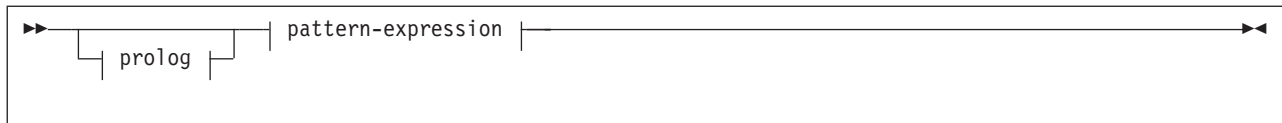
other-options:



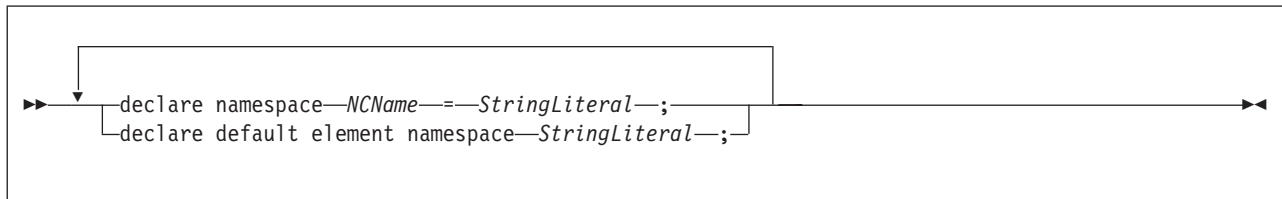
XML-index-specification:



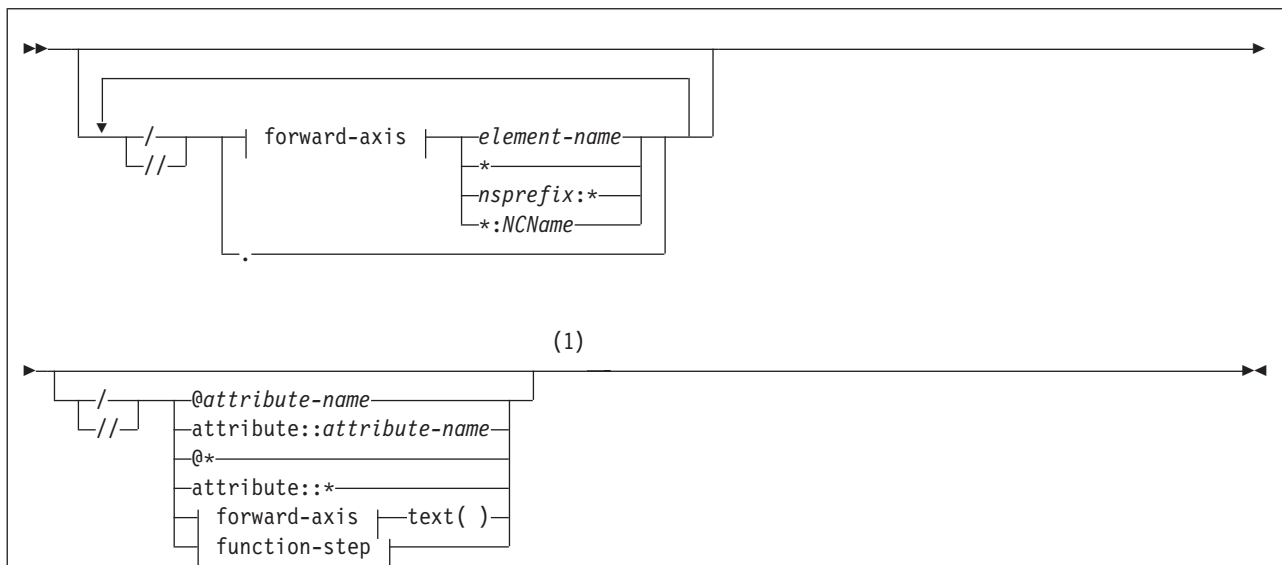
XML-pattern-clause:



prolog:



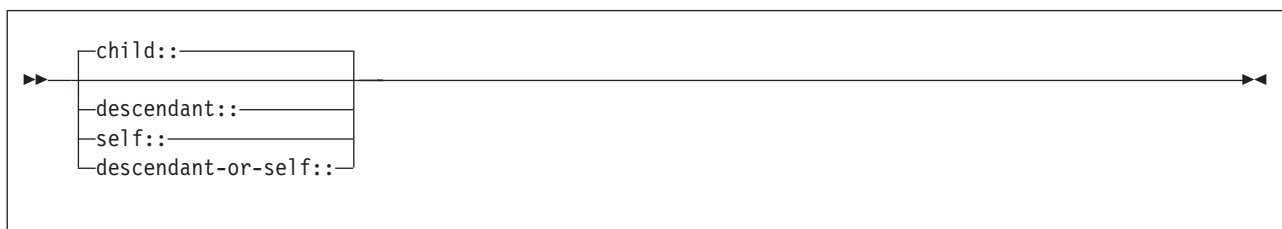
pattern-expression:



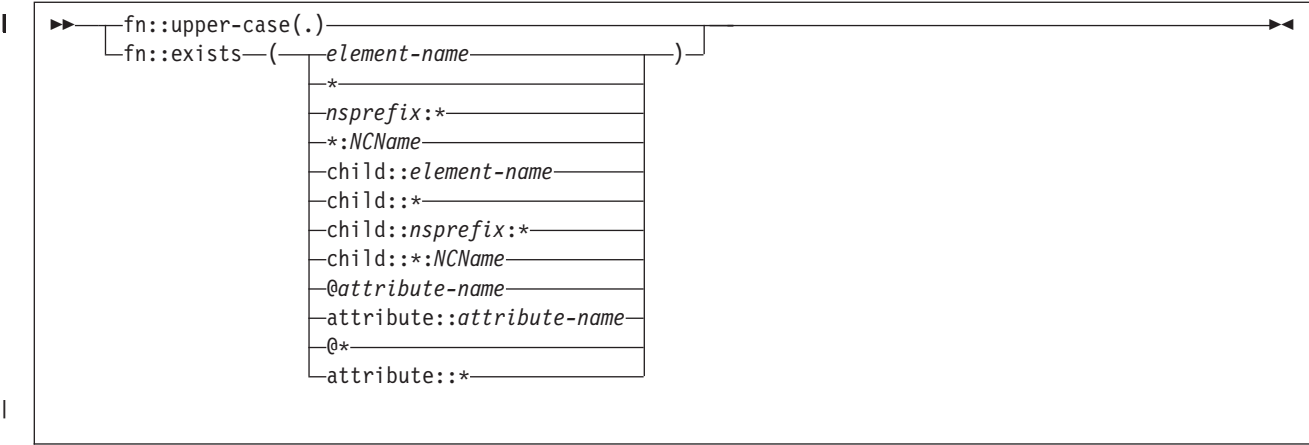
Notes:

- 1 *pattern-expression* cannot be an empty string.

forward-axis:



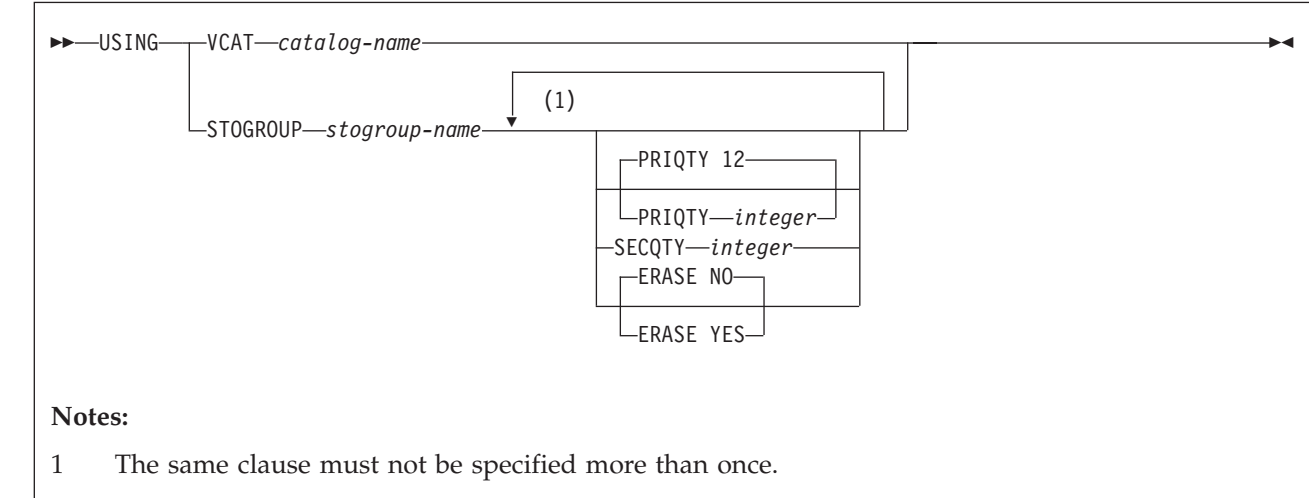
function-step:



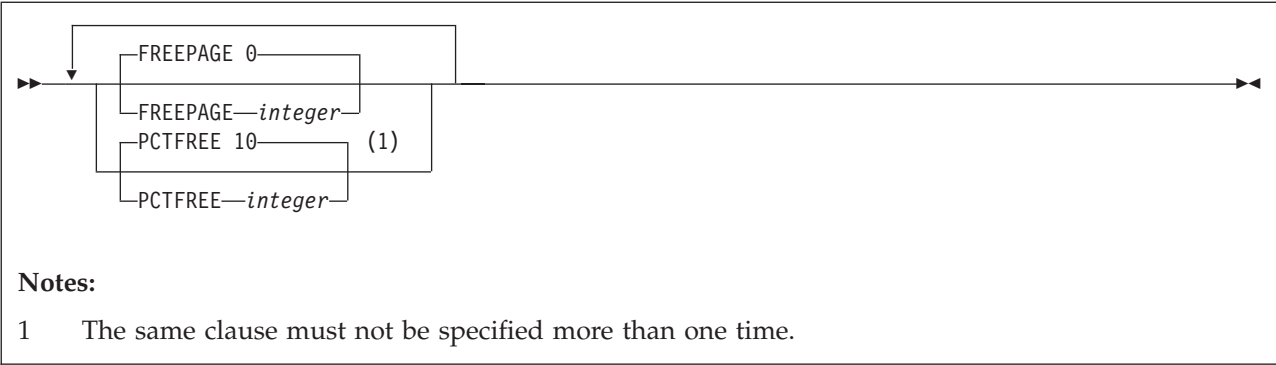
SQL-data-type:



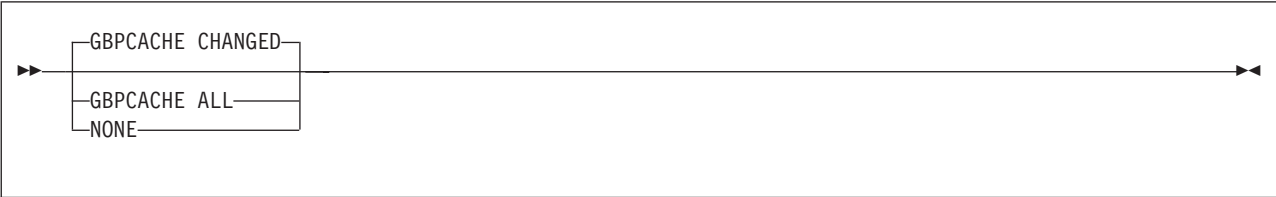
using-specification:



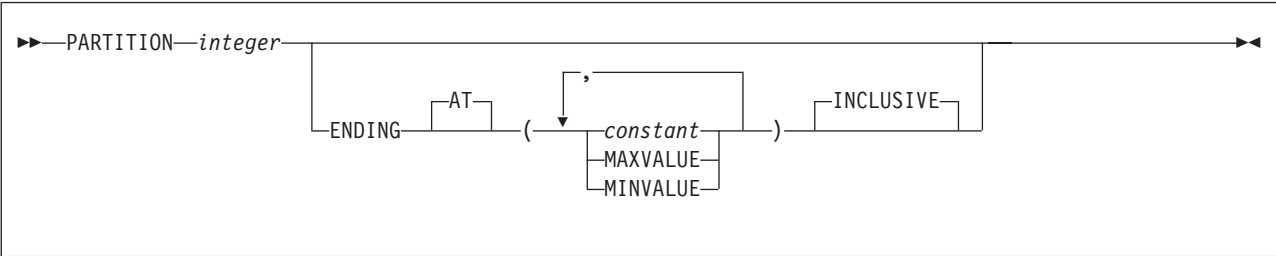
free-specification:



gbpcache-specification:



partition-element:



Description

UNIQUE

Prevents the table from containing two or more rows with the same value of the index key. When **UNIQUE** is used, all null values for a column are considered equal. For example, if the key is a single column that can contain null values, that column can contain only one null value. The constraint is enforced when rows of the table are updated or new rows are inserted.

The constraint is also checked during the execution of the **CREATE INDEX** statement. If the table already contains rows with duplicate key values, the index is not created.

UNIQUE WHERE NOT NULL

Prevents the table from containing two or more rows with the same value of the index key where all null values for a column are not considered equal. Multiple null values are allowed. Otherwise, this is identical to **UNIQUE**.

INDEX *index-name*

Names the index. The name must not identify an index that exists at the current server.

The associated index space also has a name. That name appears as a qualifier in the names of data sets defined for the index. If the data sets are managed by

the user, the name is the same as the second (or only) part of *index-name*. If this identifier consists of more than eight characters, only the first eight are used. The name of the index space must be unique among the names of the index spaces and table spaces of the database for the identified table. If the data sets are defined by DB2, DB2 derives a unique name.

If the index is an index on a declared temporary table, the qualifier, if explicitly specified, must be SESSION. If the index name is unqualified, DB2 uses SESSION as the implicit qualifier.

ON *table-name* or *aux-table-name*

Identifies the table on which the index is created. The name can identify a base table, a materialized query table, a declared temporary table, or an auxiliary table.

table-name

Identifies the base table, materialized query table, or declared temporary table on which the index is created. The name must identify a table that exists at the current server. (The name of a declared temporary table must be qualified with SESSION.) The name must not identify a clone table. The name must not identify a created temporary table or a table that is implicitly created for an XML column. If the index that is being created is for XML values, the table can contain an XML column, otherwise, the table must not contain an XML column. The name cannot identify a catalog table or declared temporary table if the index is created using expressions.

***column-name*,...**

Specifies the columns of the index key.

Each *column-name* must identify a column of the table. Do not specify more than 64 columns or the same column more than one time. Do not qualify *column-name*. Do not specify a column for *column-name* that is defined as follows:

- a LOB column (or a column with a distinct type that is based on a LOB data type)
- a DECFLOAT column (or a column with a distinct type that is based on a DECFLOAT data type)
- a BINARY or VARBINARY column (or a column with a distinct type that is based on a BINARY or VARBINARY data type)
- a row change timestamp column when the PARTITION BY RANGE clause is also specified.

A column with an XML type can only be specified if the XMLPATTERN clause is also specified. If the XMLPATTERN clause is specified, only one column can be identified and the column must be an XML type. The resulting index is an XML index.

The sum of the length attributes of the columns must not be greater than the following limits, where *n* is the number of columns that can contain null values and *m* is the number of varying-length columns in the key:

- $2000 - n$ for a padded, nonpartitioning index
- $2000 - n - 2m$ for a nonpadded, nonpartitioning index
- $255 - n$ for a partitioning index (padded or nonpadded)
- $255 - n - 2m$ for a nonpadded, partitioning index

ASC

Puts the index entries in ascending order by the column. ASC cannot be specified with the GENERATE KEY USING clause.

ASC is the default.

DESC

Puts the index entries in descending order by the column. DESC cannot be specified with the GENERATE KEY USING clause or if the ON clause contains *key-expression*.

RANDOM

Index entries are put in a random order by the column. RANDOM cannot be specified in the following cases:

- For an index key column that is varying length in an index that is created with the NOT PADDED option
- With the GENERATE KEY USING clause
- If the index is part of the partitioning key

key-expression

Specifies an expression that returns a scalar value. *key-expression* cannot be specified with the GENERATE KEY USING clause. *key-expression* has the following restrictions:

- Each *key-expression* must contain at least one reference to a column of *table-name*.

All references to columns of *table-name* must be unqualified.

Referenced columns cannot be LOB, XML, or DECFLOAT data types or a distinct type that is based on one of these data types.

Referenced columns cannot include any FIELDPROCs or a SECURITY LABEL. Referenced columns cannot be implicitly hidden (that is, defined with the IMPLICITLY HIDDEN attribute).

- *key-expression* must not include the following:
 - A subquery
 - An aggregate function
 - A function that is not deterministic function
 - A function that has an external action
 - A user-defined function
 - A sequence reference
 - A host variable
 - A parameter marker
 - A special register
 - A CASE expression
 - An OLAP specification
- If *key-expression* references a cast function, the privilege set must implicitly include EXECUTE authority on the generated cast functions for the distinct type.
- If *key-expression* references the LOWER or UPPER functions, the input *string-expression* cannot be FOR BIT DATA, and the function invocation must contain the *locale-name* argument.
- If *key-expression* references the TRANSLATE function, the function invocation must contain the *to-string* argument.
- The same expression cannot be used more than one time in the same index.
- The data type of the result of the expression (including the intermediate result) cannot be a LOB, XML, or DECFLOAT value.

- The CCSID encoding schema of the result of a *key-expression* must be the same encoding scheme as the table.

The maximum length of the text string of each *key-expression* is 4000 bytes after conversion to UTF-8. The maximum number of *key-expression* in an extended index is 64.

aux-table-name

Identifies the auxiliary table on which the index is created. The name must identify an auxiliary table that exists at the current server. If the auxiliary table already has an index, do not create another one. An auxiliary table can only have one index.

Do not specify any columns for the index key. The key value is implicitly defined as a unique 19 byte value that is system generated.

If qualified, *table-name* or *aux-table-name* can be a two-part or three-part name. If a three-part name is used, the first part must match the value of the field DB2 LOCATION NAME of installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.) Whether the name is two-part or three-part, the authorization ID that qualifies the name is the owner of the index.

The table space that contains the named table must be available to DB2 so that its data sets can be opened. If the table space is EA-enabled, the data sets for the index must be defined to belong to a DFSMS data class that has the extended format and addressability attributes.

GENERATE KEY USING

Along with XMLPATTERN, GENERATE KEY USING is required to generate an XML index.

XMLPATTERN

When an XML column is indexed, only parts of the documents will be indexed. To identify those parts, a path expression that follows the XMLPATTERN clause is specified. Only values of those element, attribute, or text nodes which match the specified pattern are indexed. An XML pattern can be specified using an optional namespace declaration where namespace prefixes are mapped to namespace URIs and by providing a path expression. The path expression is similar to a path expression in XQuery except that the paths that are specified for the XML index can support child axis, self-or-descendant axis, wildcard expressions, or attribute only. The maximum length of an XML pattern text is 4000 bytes after being converted to UTF-8. Refer to *DB2 XML Guide* for more information about XQuery.

prolog

To use qualified names in the *pattern-expression*, namespace prefixes need to be declared. A default namespace can also be declared for use with unqualified names.

declare namespace *NCName=StringLiteral*

The namespace prefix, *NCName*, is mapped to a namespace URI that is identified in *StringLiteral*. Multiple namespaces can be declared, but each namespace prefix must be unique within the list of namespace declarations. *NCName* is an XML name as defined by the XML 1.0 standard. *NCName* cannot include a colon character. The namespace URI cannot be <http://www.w3.org/XML/1998/namespace> or <http://w3.org/2000/xmlns/>.

declare default element namespace *StringLiteral*

Specifies the default namespace URI for unqualified names of elements and types. *StringLiteral* is a namespace URI. If no default element namespace is declared, unqualified names of element and types are in no namespace. Only one default namespace can be declared.

pattern-expression

Pattern-expression is used to identify those nodes in an XML document that are indexed. *Pattern-expression* cannot be an empty or invalid string, and the XPath expression cannot be nested more than 50 levels.

/ (forward slash)

Separates path expression steps.

// (double forward slash)

Abbreviated syntax for `/descendant-or-self::node()/`

. (dot)

Abbreviated syntax for `/self::node()/`

child::

Specifies children of the context node. `child::` is the default if no forward axis is specified.

descendant::

Specifies the descendants of the context node.

self::

Specifies the current context node.

descendant-or-self::

Specifies the context node and the descendants of the context node.

element-name

Identifies an element in an XML document. *element-name* is an XML QName that can have one of the following forms:

namespace-prefix:NCName

namespace-prefix explicitly specifies a namespace prefix that must be declared.

NCName

An unqualified XML name that uses the default namespace.

*** (an asterisk)**

Indicates any element name. If `*` is prefixed by `attribute::` or `@`, `*` indicates any attribute name.

*namespace-prefix:**

Indicates any NCName within the specified namespace.

**:NCName*

Indicates a specific XML name in any of the currently declared namespaces.

attribute:: or @

Specifies attributes of the context node.

attribute-name

Identifies an attribute in an XML document. *attribute-name* is an XML QName that can have one of the following forms:

nsprefix:NCName

nsprefix explicitly specifies a namespace prefix that must be declared.

NCName

An unqualified XML name that uses the default namespace.

text()

Matches any text node.

fn:upper-case(.)

Specifies an element node or an attribute node that identifies the key value for the index for each node that is specified by the context step (the part of *pattern-expression* that is specified prior to fn:upper-case).

The context step of fn:upper-case() must specify an element node or an attribute node. The argument of fn:upper-case() must be a self step. The key values of an XML value index must be specified as the SQL data type VARCHAR. The length of the VARCHAR value can be any value that is allowed in DB2.

fn:exists()

Specifies an element node that identifies the key value for the index for each node that is specified by the context step (the part of *pattern-expression* that is specified prior to fn:exists).

The context step of fn:exists() must specify an element node. The argument of fn:exists() must be either a single step of a child element node or an attribute node. The name test part can be a wildcard character for either the namespace prefix or NCName. The key values of an XML value index for an XPath expression that ends with fn:exists() must be specified as the SQL data type VARCHAR(1). The key value will be "T" or "F". "T" implies that fn:exists() evaluates to true and "F" implies that fn:exists() evaluates to false.

AS SQL data-type

Specifies that indexed values are stored as an instance of the specified SQL data type. Casting to the specified data type can result in a loss of precision of the values. For example, a loss of precision can occur when an XML integer value is cast to the SQL data type DECFLOAT. If the cast causes a loss of precision, the result will be rounded to the approximate value when it is stored in the index. The cast result cannot be outside of the range that is supported by the SQL data type. If the value cannot be cast to the specified data type, the document is still inserted into the table, but the index entry for that value is not created. No error or warning code is returned.

If the index is unique, the uniqueness is enforced on the value after it is cast to the specified type. Because rounding can occur during the cast to the SQL data type, if a value is cast to the same key value as a document that the table already contains, DB2 will return duplicate key errors at insert time, or fail to create the index.

VARCHAR (integer)

The length *integer* is a value in the range of 1 to 1000 bytes. If VARCHAR is specified with a length, the specified length is treated as a constraint. If documents are inserted into a table (or exist in the table at create index time) that have nodes with values that are longer than the specified length, the insert or index creation will fail.

DECFLOAT

DECFLOAT can be specified to index numeric values. For the cast to

succeed, the string must be a valid XML numeric type. Otherwise the value will be ignored and no insert to the index will occur. The result of the cast cannot be outside of the range that DECFLOAT can represent. Because the XML Schema data type for numeric values allows greater precision than the SQL data type, the result might be rounded to fit into the SQL data type. The DECFLOAT values that are stored in the index are the normalized numeric values.

CLUSTER or NOT CLUSTER

Specifies whether the index is the clustering index for the table. This clause must not be specified for an index on an auxiliary table.

CLUSTER

The index is to be used as the clustering index of the table. CLUSTER cannot be specified if XMLPATTERN or *key-expression* is specified.

NOT CLUSTER

The index is not to be used as the clustering index of the table.

PARTITIONED

Specifies that the index is data partitioned (that is, partitioned according to the partitioning scheme of the underlying data). A partitioned index can be created only on a partitioned table space, not of a partition-by-growth table space. PARTITIONED cannot be specified if XMLPATTERN is specified. The types of partitioned indexes are partitioning and secondary.

An index is considered a partitioning index if the specified index key columns match or comprise a superset of the columns specified in the partitioning key, are in the same order, and have the same ascending or descending attributes.

If PARTITION BY was not specified when the table was created, the CREATE INDEX statement must have the ENDING AT clause specified to define a partitioning index and use index-controlled partitioning. This index is created as a partitioned index even if the PARTITIONED keyword is not specified. When a partitioning index is created, if both the PARTITIONED and ENDING AT keywords are omitted, the index will be nonpartitioned. If PARTITIONED is specified, the USING specification with PRIQTY and SECQTY specifications are optional. If these space parameters are not specified, default values are used.

A secondary index is any index defined on a partitioned table space that does not meet the definition of the partitioning index. For partitioned secondary indexes (data-partitioned secondary indexes), the ENDING AT clause is not allowed because the partitioning scheme of the index is predetermined by that of the underlying data. UNIQUE and UNIQUE WHERE NOT NULL are not allowed unless the columns in the index are a superset of the partitioning columns. If a partitioned secondary index is created on a table that uses index-controlled partitioning, the table is converted to use table-controlled partitioning.

NOT PADDED or PADDED

Specifies how varying-length string columns are to be stored in the index. If the index contains no varying-length columns, this option is ignored, and a warning message is returned. Indexes that do not have varying-length string columns are always created as physically padded indexes.

NOT PADDED

Specifies that varying-length string columns are not to be padded to their maximum length in the index. The length information for a varying-length column is stored with the key.

NOT PADDED is ignored and has no effect if the index is being created on an auxiliary table. Indexes on auxiliary tables are always padded.

PADDED

Specifies that varying-length string columns within the index are always padded with the default pad character to their maximum length. PADDED cannot be specified if XMLPATTERN is specified. PADDED cannot be specified for indexes that are defined on VARBINARY columns.

When the index contains at least one varying-length column, the default for the option depends on the value of field PAD INDEXES BY DEFAULT on installation panel DSNTIPE:

- When the value of this field is NO, new indexes are not padded unless PADDED is specified.
- When the value of this field is YES, new indexes are padded unless NOT PADDED is specified.

The components of the USING clause are discussed below, first for nonpartitioned indexes and then for partitioned indexes.

Using clause for nonpartitioned indexes

For nonpartitioned indexes, the USING clause indicates whether the data sets for the index are to be managed by the user or managed by DB2. If DB2 definition is specified, the clause also gives space allocation parameters (PRIQTY and SECQTY) and an erase rule (ERASE).

If you omit USING, the data sets will be managed by DB2 on volumes listed in the default storage group of the database that is associated with the table. That default storage group must exist. With no USING clause, PRIQTY, SECQTY, and ERASE assume their default values.

VCAT *catalog-name*

Specifies that the first data set for the index is managed by the user, and that following data sets, if needed, are also managed by the user.

The data sets defined for the index are linear VSAM data sets cataloged in an integrated catalog facility catalog identified by *catalog-name*. An alias²⁹ must be used if *catalog-name* is longer than eight characters.

Conventions for index data set names are given in *DB2 Administration Guide*. *catalog-name* is the first qualifier for each data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

Do not specify VCAT for an index on a declared temporary table or if the table space is partition-by-growth.

STOGROUP *stogroup-name*

Specifies that DB2 will define and manage the data sets for the index. Each data set will be defined on a volume listed in the identified storage group. The values specified (or the defaults) for PRIQTY and SECQTY determine the primary and secondary allocations for the data set. If

29. The alias of an integrated catalog facility catalog

PRIQTY+118×SECQTY is 2 gigabytes or greater, more than one data set could eventually be used, but only the first is defined during execution of this statement.

To use USING STOGROUP, the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for that storage group. Moreover, *stogroup-name* must identify a storage group that exists at the current server and includes in its description at least one volume serial number. The description can indicate that the choice of volumes will be left to Storage Management Subsystem (SMS). Each volume specified in the storage group must be accessible to z/OS for dynamic allocation of the data set, and all these volumes must be of the same device type.

The integrated catalog facility catalog used for the storage group must *not* contain an entry for the first data set of the index. If the catalog is password protected, the description of the storage group must include a valid password.

The storage group supplies the data set name. The first level qualifier is also the name of, or an alias for, the integrated catalog facility catalog on which the data set is to be cataloged. The naming convention for the data set is the same as if the data set is managed by the user.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set. When you specify PRIQTY (with a value other than -1), the primary space allocation is at least *n* kilobytes, where *n* is:

12 If *integer* is less than 12

integer

If *integer* is between 12 and 4194304

2097152

If both of the following conditions are true:

- *integer* is greater than 2097152.
- The index is a non-partitioned index on a table space that is not defined with the LARGE or DSSIZE attribute.

4194304

If *integer* is greater than 4194304

If you do not specify PRIQTY or specify PRIQTY -1, DB2 uses a default value for the primary space allocation; for information on how DB2 determines the default value, see Rules for primary and secondary space allocation.

If you specify PRIQTY and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of 4KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see the description of the DEFINE CLUSTER command in *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

When determining a suitable value for PRIQTY, be aware that two of the pages of the primary space could be used by DB2 for purposes other than storing index entries.

SECQTY *integer*

Specifies the minimum secondary space allocation for a DB2-managed data set. If you do not specify SECQTY, DB2 uses a formula to determine a value. For information on the actual value that is used for

secondary space allocation, whether you specify a value or not, see Rules for primary and secondary space allocation.

If you specify SECQTY and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of 4KB not less than *n*. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the space requested. To more closely estimate the actual amount of storage, see the description of the DEFINE CLUSTER command in .

ERASE

Indicates whether the DB2-managed data sets are to be erased when they are deleted during the execution of a utility or an SQL statement that drops the index. Refer to for more information.

NO Does not erase the data sets. Operations involving data set deletion will perform better than ERASE YES. However, the data is still accessible, though not through DB2. This is the default.

YES

Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

USING clause for partitioned indexes:

If the index is partitioned, there is a PARTITION clause for each partition. Within a PARTITION clause, a USING clause is optional. If a USING clause is present, it applies to that partition in the same way that a USING clause for a secondary index applies to the entire index.

When a USING specification is absent from a PARTITION clause, the USING clause parameters for the partition depend on whether a USING clause is specified before the PARTITION clauses.

- If the USING clause is specified, it applies to every PARTITION clause that does not include a USING clause.
- If the USING clause is not specified, the following defaults apply to the partition:
 - Data sets are managed by DB2
 - The default storage group for the database is used
 - A value of 12 is used for PRIQTY and SECQTY
 - A value of NO is used for ERASE

VCAT *catalog-name*

Specifies a user-managed data set with a name that starts with the specified catalog name. You must specify an alias for the integrated catalog facility catalog if the name of the integrated catalog facility catalog is longer than eight characters.

If *n* is the number of the partition, the identified integrated catalog facility catalog must already contain an entry for the *n*th data set of the index, conforming to the DB2 naming convention for data sets set forth in *DB2 Administration Guide*.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DB2 assumes one and only one data set for each partition.

STOGROUP *stogroup-name*

If USING STOGROUP is used, explicitly or by default, for a partition *n*, DB2 defines the data set for the partition during the execution of the CREATE INDEX statement, using space from the named storage group. The privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for that storage group. The integrated catalog facility catalog used for the storage group must NOT contain an entry for the *n*th data set of the index.

stogroup-name must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group.

If you omit PRIQTY, SECQTY, or ERASE from a USING STOGROUP clause for some partition, their values are given by the next USING STOGROUP clause that governs that partition: either a USING clause that is not in any PARTITION clause, or a default USING clause. DB2 assumes one and only one data set for each partition.

FREEPAGE *integer*

Specifies how often to leave a page of free space when index entries are created as the result of executing a DB2 utility or when creating an index for a table with existing rows. One free page is left for every *integer* pages. The value of *integer* can range from 0 to 255. The default is 0, leaving no free pages.

Do not specify FREEPAGE for an index on a declared temporary table.

PCTFREE *integer*

Determines the percentage of free space to leave in each nonleaf page and leaf page when entries are added to the index or index partition as the result of executing a DB2 utility or when creating an index for a table with existing rows. The first entry in a page is loaded without restriction. When additional entries are placed in a nonleaf or leaf page, the percentage of free space is at least as great as *integer*.

The value of *integer* can range from 0 to 99, however, if a value greater than 10 is specified, only 10 percent of free space will be left in nonleaf pages. The default is 10.

Do not specify PCTFREE for an index on a declared temporary table.

If the index is partitioned , the values of FREEPAGE and PCTFREE for a particular partition are given by the first of these choices that applies:

- The values of FREEPAGE and PCTFREE given in the PARTITION clause for that partition. Do not use more than one *free-specification* in any PARTITION clause.
- The values given in a *free-specification* that is not in any PARTITION clause.
- The default values FREEPAGE 0 and PCTFREE 10.

GBPCACHE

In a data sharing environment, specifies what index pages are written to the group buffer pool. In a non-data-sharing environment, the option is ignored unless the index is on a declared temporary table. Do not specify GBPCACHE for an index on a declared temporary table in either environment (data sharing or non-data-sharing).

CHANGED

Specifies that updated pages are written to the group buffer pool, when

there is inter-DB2 R/W interest on the index or partition. When there is no inter-DB2 R/W interest, the group buffer pool is not used. Inter-DB2 R/W interest exists when more than one member in the data sharing group has the index or partition open, and at least one member has it open for update. GBPCACHE CHANGED is the default.

If the index is in a group buffer pool that is defined as GBPCACHE(NO), CHANGED is ignored and no pages are written to the group buffer pool.

ALL

Indicates that pages are written to the group buffer pool as they are read in from DASD.

Exception: In the case of a single updating DB2 subsystem when no other DB2 subsystems have any interest in the page set, no pages are written to the group buffer pool.

If the index is in a group buffer pool that is defined as GBPCACHE(NO), ALL is ignored and no pages are written to the group buffer pool.

NONE

Indicates that no pages are written to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If the index is **partitioned**, the value of GBPCACHE for a particular partition is given by the first of these choices that applies:

1. The value of GBPCACHE given in the PARTITION clause for that partition. Do not use more than one *gbpcache-specification* in any PARTITION clause.
2. The value given in a *gbpcache-specification* that is not in any PARTITION clause.
3. GBPCACHE CHANGED is the default value.

DEFINE

Specifies when the underlying data sets for the index are physically created. The SPACE column in catalog table SYSINDEXPART is used to record the status of the data sets (undefined or allocated).

YES

The data sets are created when the index is created (the CREATE INDEX statement is executed). YES is the default.

NO The data sets are not created until data is inserted into the index.

DEFINE NO is applicable only for DB2-managed data sets (USING STOGROUP is specified). Use DEFINE NO especially when performance of the CREATE INDEX statement is important or DASD resource is constrained.

Do not use DEFINE NO on an index if you use a program outside of DB2 to propagate data into a table on which that index is defined. If you use DEFINE NO on an index of a table and data is then propagated into the table from a program that is outside of DB2, the index space data sets are allocated, but the DB2 catalog will not reflect this fact. As a result, DB2 treats the data sets for the index space as if they have not yet been allocated. The resulting inconsistency causes DB2 to deny application programs access to the data until the inconsistency is resolved.

DEFINE NO is ignored for user-managed data sets (USING VCAT is specified). DEFINE NO is also ignored if the index is being created on a table that is not empty or on an auxiliary table.

Do not specify **DEFINE NO** if the index is created on a base table that is involved in a clone relationship.

Do not specify **DEFINE NO** for an index on a declared temporary table.

Do not specify **DEFINE NO** if **XMLPATTERN** is specified.

COMPRESS NO or COMPRESS YES

Specifies whether compression for index data will be used. If the index is partitioned, the clause will apply to all partitions.

COMPRESS NO

Specifies that no index compression will be used.

COMPRESS NO is the default.

COMPRESS YES

Specifies that index compression will be used. The bufferpool that is used to create the index must be 8K, 16K, or 32K in size. The physical page size on disk will be 4K. The index compression will take place immediately.

Index compression is recommended for applications that do sequential insert operations with few or no delete operations. Random inserts and deletes can adversely effect compression. Index compress is also recommended for applications where the indexes are created primarily for scan operations.

PARTITION BY RANGE

Specifies the partitioning index for the table, which determines the partitioning scheme for the data in the table.

PARTITION BY RANGE should only be specified if the table space is partitioned and the partitioning schema has not already been established.

PARTITION BY RANGE must not be specified if the index is an extended index or if the table is in a universal table space (ranged-partitioned or partition-by-growth table space).

partition-element

Specifies the range for each partition.

PARTITION integer

A **PARTITION** clause specifies the highest value of the index key in one partition of a partitioning index. In this context, highest means highest in the sorting sequences of the index columns. In a column defined as *ascending* (ASC), highest and lowest have their usual meanings. In a column defined as *descending* (DESC), the lowest actual value is highest in the sorting sequence.

If you use **CLUSTER**, and the table is contained in a partitioned table space, you must use exactly one **PARTITION** clause for each partition (defined with **NUMPARTS** on **CREATE TABLESPACE**). If there are *p* partitions, the value of *integer* must range from 1 through *p*.

The length of the highest value of a partition (also called the limit key) is the same as the length of the partitioning index.

ENDING AT(constant, MAXVALUE, or MINVALUE...)

Specifies that this is the partitioning index and indicates how the data will be partitioned. The table space is marked complete after this partitioning index is created. You must use at least one value (*constant*, **MAXVALUE**, or **MINVALUE**) after **ENDING AT** in each **PARTITION** clause. You can use as many as there are columns in the key. The

concatenation of all the values is the highest value of the key in the corresponding partition of the index unless the VALUES statement was already specified when the table or previous index was created.

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must be MAXVALUE.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MAXVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on. Using fewer values than there are columns in the key has the same effect as using the highest or lowest values for the omitted columns, depending on whether they are ascending or descending.
- If a key includes a ROWID column or a column with a distinct type that is based on a ROWID data type, 17 bytes of the constant that is specified for the corresponding ROWID column are considered.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition.
- The combination of the number of table space partitions and the corresponding limit key size cannot exceed the number of partitions * (106 + limit key size in bytes) < 65394.
- If the concatenation of all the values exceeds 255 bytes, only the first 255 bytes are considered.
- The highest value of the key in the last partition depends on how the table space is defined. For table spaces that are created without the LARGE or DSSIZE options, the values that you specify after VALUES are not enforced. The highest value of the key that can be placed in the table is the highest possible value of the key.

For large partitioned table space, the values you specify are enforced. The value specified for the last partition is the highest value of the key that can be placed in the table. Any key values greater than the value that is specified for the last partition are out of range.

ENDING AT can be specified only if the ENDING AT clause was not specified on a previous CREATE or ALTER TABLE statement for the underlying table.

INCLUSIVE

Specifies that the specified range values are included in the data partition.

BUFFERPOOL *bpname*

Identifies the buffer pool that is to be used for the index. The *bpname* must identify an activated 4KB, 8KB, 16KB, or 32KB buffer pool and the privilege set must include SYSADM or SYSCTRL authority or the USE privilege for the buffer pool.

The default is the default 4KB buffer pool for indexes in the database. A buffer pool with a smaller size should be chosen for indexes with random insert patterns. A buffer pool with a larger size should be chosen for indexes with sequential insert patterns.

See “Naming conventions” on page 51 for more details about *bpname*. See *DB2 Command Reference* for a description of active and inactive buffer pools.

CLOSE

Specifies whether or not the data set is eligible to be closed when the index is not being used and the limit on the number of open data sets is reached.

YES

Eligible for closing. This is the default unless the index is on a declared temporary table.

NO Not eligible for closing.

If the limit on the number of open data sets is reached and there are no page sets that specify CLOSE YES to close, page sets that specify CLOSE NO will be closed.

For an index on a declared temporary table, DB2 uses CLOSE NO regardless of the value specified.

DEFER

Indicates whether the index is built during the execution of the CREATE INDEX statement. Regardless of the option specified, the description of the index and its index space is added to the catalog. If the table is determined to be empty and DEFER YES is specified, the index is neither built nor placed in a rebuild-pending status. Refer to *DB2 Administration Guide* for more information about using DEFER. Do not specify DEFER for an index on a declared temporary table or an auxiliary table.

NO The index is built. This is the default.

YES

The index is not built. If the table is populated, the index is placed in a rebuild-pending status and a warning message is issued; the index must be rebuilt by the REBUILD INDEX utility.

PIECESIZE *integer*

Specifies the maximum addressability of each data set for a secondary index. The subsequent keyword K, M, or G, indicates the units of the value that is specified in *integer*.

- K** Indicates that the *integer* value is to be multiplied by 1024 to specify the maximum data set size in bytes. *integer* must be a power of two between 256 and 67,108,864.
- M** Indicates that the *integer* value is to be multiplied by 1,048,576 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 65,536.
- G** Indicates that the *integer* value is to be multiplied by 1,073,741,824 to specify the maximum data set size in bytes. *integer* must be a power of two between 1 and 64.

Table 95 shows the valid values for the data set size, which depend on the size of the table space.

Table 95. Valid values of *PIECESIZE* clause

K units	M units	G units	Size attribute of table space
256K			
512K			
1024K	1M		
2048K	2M		
4096K	4M		
8192K	8M		
16384K	16M		
32768K	32M		
65536K	64M		
131072K	128M		
262144K	256M		
524288K	512M		
1048576K	1024M	1G	
2097152K	2048M	2G	
4194304K	4096M	4G	LARGE, DSSIZE 4G (or greater)
8388608K	8192M	8G	DSSIZE 8G (or greater)
16777216K	16384M	16G	DSSIZE 16G (or greater)
33554432K	32768M	32G	DSSIZE 32G (or greater)
67108864K	65536M	64G	DSSIZE 64G

PIECESIZE has no effect on primary and secondary space allocation as it is only a specification of the maximum amount of data that a data set can hold and not the actual allocation of storage.

The default for PIECESIZE size is 2G (2GB) for indexes that are backed by table spaces that were created without the LARGE or DSSIZE option. For tables spaces that were created with the LARGE or DSSIZE option, the index piece size is the result of the following:

$$\text{MIN}(x, 2^{32} / (\text{MIN}(4096, 2^{32} / (x / y))) * z)$$

where

x is the DSSIZE of the table space

y is the page size of the table space

z is the page size of the index

If you change the PIECESIZE value with the ALTER INDEX statement, note that the index is put into REBUILD-pending status.

COPY

Indicates whether the COPY utility is allowed for the index. Do not specify COPY for an index on a declared temporary table.

NO Does not allow full image or concurrent copies or the use of the RECOVER utility on the index. NO is the default.

YES

Allows full image or concurrent copies and the use of the RECOVER utility on the index.

Notes

Owner privileges: The owner of the table has all table privileges (see “GRANT (table or view privileges)” on page 1355) with the ability to grant these privileges to others. For more information about ownership of the object, see “Authorization, privileges, and object ownership” on page 60.

Effects of the DEFER clause: If DEFER NO is implicitly or explicitly specified, the CREATE INDEX statement cannot be executed while a DB2 utility has control of the table space that contains the identified table.

If the identified table already contains data and if the index build is not deferred, CREATE INDEX creates the index entries for it. If the table does not yet contain data, CREATE INDEX creates a description of the index; the index entries are created when data is inserted into the table.

Errors evaluating the expressions for an index: Errors that occur during the evaluation of an expression for an index are returned when the expression is evaluated. This can occur on an SQL data change statement, SELECT from an SQL data change statement, or the REBUILD INDEX utility. For example, the evaluation of the expression $10 / column_1$ returns an error if the value in *column_1* is 0. The error is returned during CREATE INDEX processing if the table is not empty and contains a row with a value of zero in *column_1*, otherwise the error is returned during the processing of the insert or update operation when a row with a value of zero in *column_1* is inserted or updated.

Result length of expressions that return a string type: If the result data type of *key-expression* is a string type and the result length cannot be calculated at bind time, the length is set to the maximum allowable length of that data type or the largest length that DB2 can estimate. In this case, the CREATE INDEX statement can fail because the total key length might exceed the limit of an index key.

For example, the result length of the expression REPEAT('A', CEIL(1.1)) is VARCHAR(32767) and the result length of the expression SUBSTR(DESCRIPTION,1,INTEGER(1.2)) is the length of the DESCRIPTION column. Therefore, a CREATE INDEX statement that uses any of these expressions as a *key-expression* might not be created because the total key length might exceed the limit of an index key.

Use of ASC or DESC on key columns: There are no restrictions on the use of ASC or DESC for the columns of a parent key or foreign key. An index on a foreign key does not have to have the same ascending and descending attributes as the index of the corresponding parent key.

EBCDIC, ASCII, and UNICODE encoding schemes for an index: An index has the same encoding scheme as its associated table.

Choosing a value for PIECESIZE: To choose a value for PIECESIZE, divide the size of the secondary index by the number of data sets that you want. For example, to ensure that you have five data sets for the secondary index, and your index is 10MB (and not likely to grow much), specify PIECESIZE 2 M. If your secondary index is likely to grow, choose a larger value.

Remember that 32 data sets is the limit if the underlying table space is not defined as LARGE (or as DSSIZE 4G or greater) and that the limit is 4096 for objects with greater than 254 parts.

Keep the PIECESIZE value in mind when you are choosing values for primary and secondary quantities. Ideally, the value of your primary quantity plus the secondary quantities should be evenly divisible into PIECESIZE.

Dropping an index: Partitioning indexes can be dropped. If the table space is using index-controlled partitioning, the table space is converted to table-controlled partitioning. Secondary indexes that are not indexes on auxiliary tables can be dropped simply by dropping the indexes. An empty index on an auxiliary table can be explicitly dropped; a populated index can be dropped only by dropping other objects. For details, see Dropping an index on an auxiliary table and an auxiliary table.

If the index is a unique index that enforces a primary key, unique key, or referential constraint, the constraint must be dropped before the index is dropped. See “DROP” on page 1256.

Unique indexes and enforcement of UNIQUE or PRIMARY KEY specifications for a table: A table requires a unique index (that is not defined as UNIQUE WHERE NOT NULL) if you use the UNIQUE or PRIMARY KEY clause in the CREATE or ALTER TABLE statements, or if there is a ROWID column that is defined as GENERATED BY DEFAULT. DB2 implicitly creates those unique indexes if the table space is explicitly created and the CREATE or ALTER TABLE statement is processed by the schema processor or if the table space is implicitly created; otherwise, you must explicitly create them. If any of the unique indexes that must be explicitly defined do not exist, the definition of the table is incomplete, and the following rules apply:

- Let K denote a key for which a required unique index does not exist and let n denote the number of unique indexes that remain to be created before the definition of the table is complete. (For a new table that has no indexes, K is its primary key or any of the keys defined in the CREATE or ALTER TABLE statement as UNIQUE and n is the number of such keys. After the definition of a table is complete, an index cannot be dropped if it is enforcing a primary key or unique key.)
- The creation of the unique index reduces n by one if the index key is identical to K . The keys are identical only if they have the same columns in the same order.
- If n is now zero, the creation of the index completes the definition of the table.

- If K is a primary key, the description of the index indicates that it is a primary index. If K is not a primary key, the description of the index indicates that it enforces the uniqueness of a key defined as UNIQUE in the CREATE or ALTER TABLE statement.

A unique index cannot be created on a materialized query table.

Unique indexes and XML columns: If the index is an XML index on a unique XML column, the uniqueness applies to values of the specified pattern across all documents of that column, and the uniqueness is enforced on the value after the value is cast to the specified SQL data type. Because the data type conversion might result in a loss of precision and normalization, multiple values that appear unique in the XML document might still result in duplicate errors. If the index is defined using an expression, the uniqueness is enforced against the values that are stored in the index, not against the original values of the columns. The **WHERE NOT NULL** specification is ignored with a warning if XMLPATTERN is also specified, and the index is treated as if **UNIQUE** had been specified.

Defining an XML index using an XPath pattern-expression that includes functions: An XPath *pattern-expression* that includes functions (including fn:exists() or fn:upper-case()) will have two parts. The first part is referred to as the *context step* and specifies the XPath of the element node or attribute node for which an index entry will be created (the element or attributes NodeID will be included in the index). The context step follows the same syntax as the XPath *pattern-expression* for an XML index, except that for fn:exists() it has to specify an element node, and for fn:upper-case() it has to specify an element node or an attribute node.

The second part is referred to as the *function expression step* and specifies the fn:exists() or fn:upper-case() XPath function. The function expression step is the right-most part of an XPath *pattern-expression*. For each node specified by the context step, the function expression step specifies the key value for the index. For example, in the XPath *pattern-expression* /purchaseOrder/items/item/fn:exists(shipDate), the context step is /purchaseOrder/items/item, and the function expression step is fn:exists(shipDate).

Use of PARTITIONED keyword: When a partitioned index is created and no additional keywords are specified, the index is nonpartitioned. If the keyword PARTITIONED is specified, the index is partitioned. If PARTITION BY RANGE is specified, the index is both data-partitioned and key-partitioned because it is defined on the partitioning columns of the table. Any index on a partitioned table space that does not meet the definition of a partitioning index is a secondary index. When a secondary index is created and no additional keywords are specified, the secondary index is nonpartitioned (NPSI). If the keyword PARTITIONED is specified, the index is a data-partitioned secondary index (DPSI).

Creating a partitioning index for a table created without partition boundaries: When a table is created without specifying partition boundaries using the ENDING AT clause, the table is incomplete until a partitioning index is created. The first index that is created for a table must specify both the PARTITION and the ENDING AT clauses.

When the PARTITION clause is specified while creating an index, either the PARTITIONED clause, or the ENDING AT clause must also be specified.

Considerations for tables that are involved in a clone relationship: If an index is created on a base table that is involved in a clone relationship, an index with the

same name is also created on the clone table. The index on the clone table will be placed in rebuild-pending status unless the clone table is empty when the index is created.

Considerations for tables that contain a row change timestamp column: To create an index that refers to a row change timestamp column in the table, values must already exist in the column for all rows. Values are stored in row change timestamp columns whenever a row is inserted or updated in the table. If the row change timestamp column is added to an existing table that contains rows, the values for the row change timestamp column is not materialized and stored at the time of the ALTER TABLE statement. Values are materialized for these rows when they are updated, or when a REORG or a LOAD REPLACE utility is run on the table or table space.

Creating indexes on DB2 catalog tables: For details on creating indexes on catalog tables, see “SQL statements allowed on the catalog” on page 1689.

EA-enabled index data sets: If an index is created for an EA-enabled table space, the data sets for the index must be set up to belong to a DFSMS data class that has the extended format and extended addressability attributes.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords when creating a partitioned index:

- **PART integer VALUES** as an alternative syntax for **PARTITION integer ENDING**. The **PARTITION BY RANGE** keyword that precedes the *partition-element* clause is optional.

Although these keywords are supported as alternatives, they are not the preferred syntax.

Examples

Example 1: Create a unique index, named DSN8910.XDEPT1, on table DSN8910.DEPT. Index entries are to be in ascending order by the single column DEPTNO. DB2 is to define the data sets for the index, using storage group DSN8G910. Each data set should hold 1 megabyte of data at most. Use 512 kilobytes as the primary space allocation for each data set and 64 kilobytes as the secondary space allocation. These specifications enable each data set to be extended up to 8 times before a new data set is used—512KB + (8*64KB)= 1024KB. Make the index padded.

The data sets can be closed when no one is using the index and do not need to be erased if the index is dropped.

```
CREATE UNIQUE INDEX DSN8910.XDEPT1
ON DSN8910.DEPT
  (DEPTNO ASC)
PADDED
USING STOGROUP DSN8G910
  PRIQTY 512
  SECQTY 64
ERASE NO
BUFFERPOOL BP1
CLOSE YES
PIECESIZE 1M;
```


For the above example, the underlying data sets for the index will be created immediately, which is the default (DEFINE YES). Assuming that table DSN8910.DEPT is empty, if you wanted to defer the creation of the data sets until data is first inserted into the index, you would specify DEFINE NO instead of accepting the default behavior. Specifying PADDED ensures that the varying-length character string columns in the index are padded with blanks.

Example 2: Create a cluster index, named XEMP2, on table EMP in database DSN8910. Put the entries in ascending order by column EMPNO. Let DB2 define the data sets for each partition using storage group DSN8G910. Make the primary space allocation be 36 kilobytes, and allow DB2 to use the default value for SECQTY, which for this example is 12 kilobytes (3 times 4KB). If the index is dropped, the data sets need not be erased.

There are to be 4 partitions, with index entries divided among them as follows:

- Partition 1: entries up to H99
- Partition 2: entries above H99 up to P99
- Partition 3: entries above P99 up to Z99
- Partition 4: entries above Z99

Associate the index with buffer pool BP1 and allow the data sets to be closed when no one is using the index. Enable the use of the COPY utility for full image or concurrent copies and the RECOVER utility.

```
CREATE INDEX DSN8910.XEMP2
ON DSN8910.EMP
(EMPNO ASC)
USING STOGROUP DSN8G910
PRIQTY 36
ERASE NO
CLUSTER
PARTITION BY RANGE
(PARTITION 1 ENDING AT('H99'),
 PARTITION 2 ENDING AT('P99'),
 PARTITION 3 ENDING AT('Z99'),
 PARTITION 4 ENDING AT('999'))
BUFFERPOOL BP1
CLOSE YES
COPY YES;
```

Example 3: Create a secondary index, named DSN8910.XDEPT1, on table DSN8910.DEPT. Put the entries in ascending order by column DEPTNO. Assume that the data sets are managed by the user with catalog name DSNCAT and each data set is to hold 1GB of data, at most, before the next data set is used.

```
CREATE UNIQUE INDEX DSN8910.XDEPT1
ON DSN8910.DEPT
(DEPTNO ASC)
USING VCAT DSNCAT
PIECESIZE 1048576K;
```

Example 4: Assume that a column named EMP_PHOTO with a data type of BLOB(110K) was added to the sample employee table for each employee's photo and auxiliary table EMP_PHOTO_ATAB was created in LOB table space DSN8D91A.PHOTOLTS to store the BLOB data for the column. Create an index named XPHOTO on the auxiliary table. The data sets are to be user-managed with catalog name DSNCAT.

```
CREATE UNIQUE INDEX DSN8910.XPHOTO
ON DSN8910.EMP_PHOTO_ATAB
USING VCAT DSNCAT
COPY YES;
```

In this example, no columns are specified for the key because auxiliary indexes have implicitly generated keys.

CREATE PROCEDURE

The CREATE PROCEDURE statement registers a stored procedure with a database server. You can register the following types of procedures with this statement, each of which is described separately.

- External

The procedure is written in a programming language such as C, COBOL, or Java. The external executable is referenced by a procedure defined at the server along with various attributes of the procedure. See “CREATE PROCEDURE (external)” on page 1016.

- SQL (external)

The procedure is written exclusively in SQL statements, but is implemented as an external program. The body of an SQL procedure is written in the SQL procedural language. The procedure body is defined at the current server along with various attributes of the procedure. See “CREATE PROCEDURE (SQL - external)” on page 1035.

- SQL (native)

The procedure is written exclusively in SQL statements, but is implemented natively in DB2. The body of an SQL procedure is written in the SQL procedural language. The procedure body is defined at the current server along with various attributes of the procedure. See “CREATE PROCEDURE (SQL - native)” on page 1046.

CREATE PROCEDURE (external)

The CREATE PROCEDURE statement defines an external stored procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is specified implicitly or explicitly.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

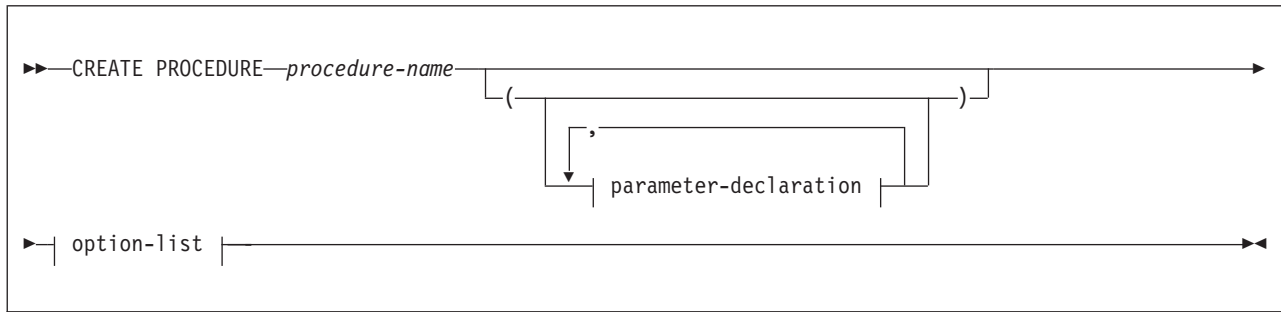
If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

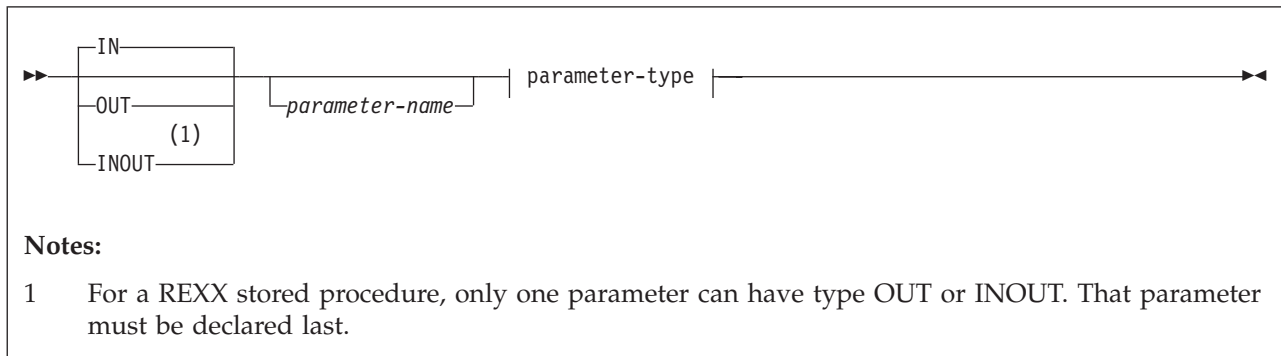
The authorization ID that is used to create the stored procedure must have authority to define programs that run in the specified WLM environment. In addition, if the stored procedure uses a distinct type as a parameter, this authorization ID must have the USAGE privilege on each distinct type that is a parameter.

When LANGUAGE is JAVA and a *jar-name* is specified in the EXTERNAL NAME clause, the privilege set must include USAGE on the JAR file, the Java archive file.

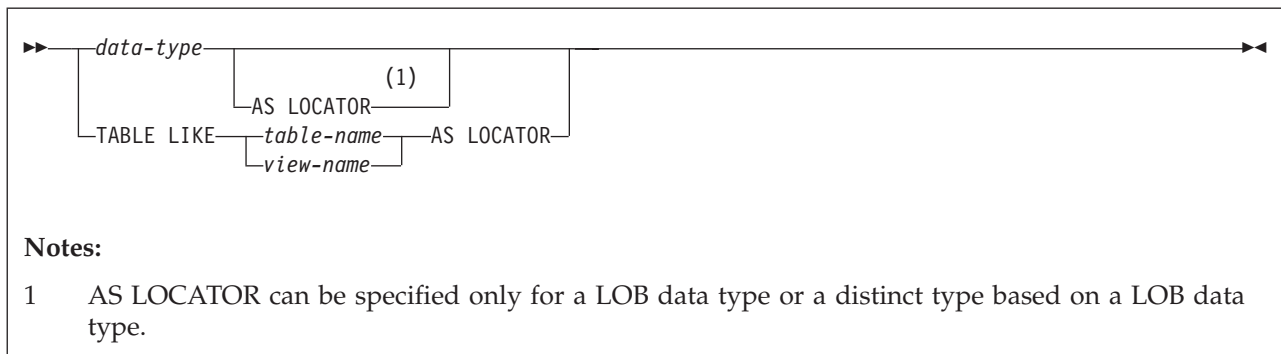
Syntax



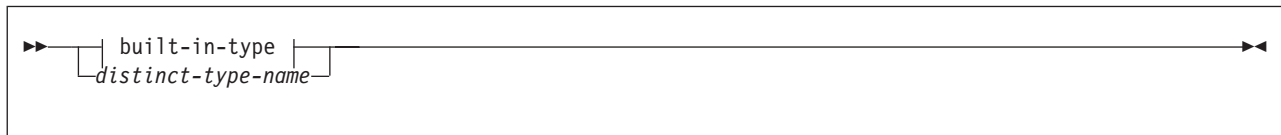
parameter-declaration:



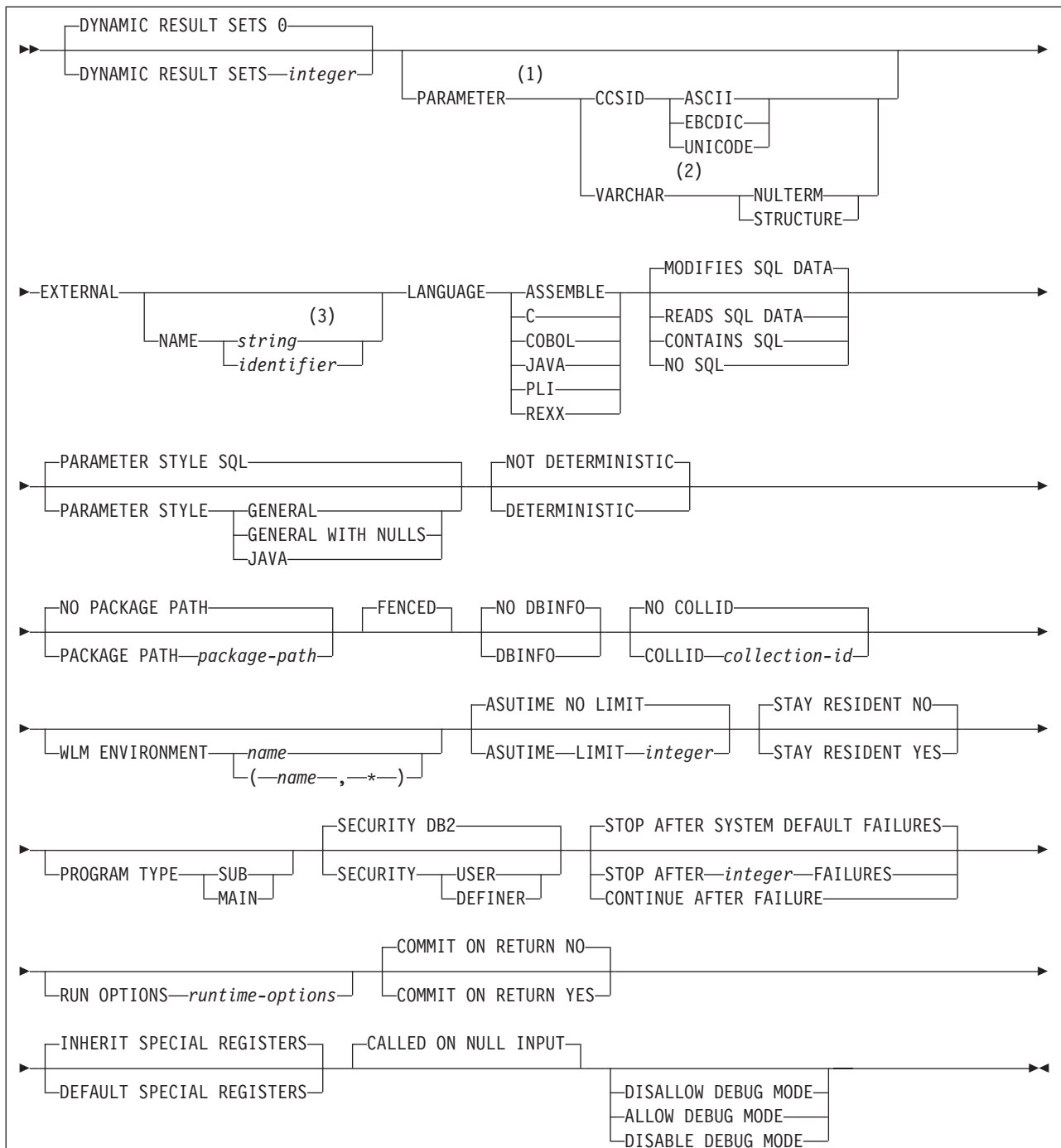
parameter-type:



data-type:



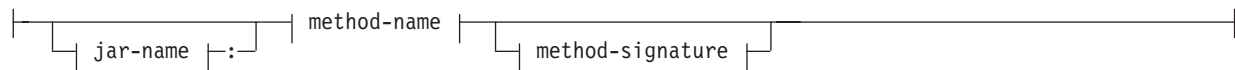
built-in-type:



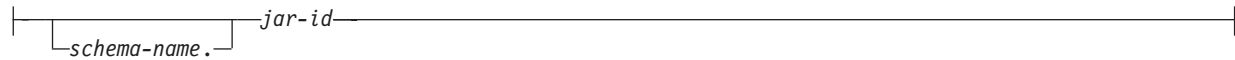
Notes:

- 1 The same clause must not be specified more than one time.
- 2 The VARCHAR clause can only be specified is LANGUAGE C is specified.
- 3 With LANGUAGE JAVA, use a valid *external-java-routine-name*.

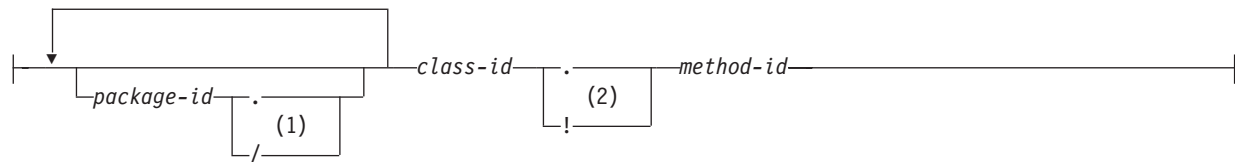
external-java-routine-name:



jar-name:



method-name:



method-signature:



Notes:

- 1 The slash (/) is supported for compatibility with previous releases of DB2 for z/OS.
- 2 The exclamation point (!) is supported for compatibility with other products in the DB2 family.

Description

procedure-name

Names the stored procedure. The name cannot be a single asterisk, even if you specify it as a delimited identifier ("*"). The name, including the implicit or explicit qualifier, must not identify an existing stored procedure at the current server.

The schema name can be 'SYSIBM' or 'SYSPROC'. It can also be 'SYSTOOLS' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

(parameter-declaration,...)

Specifies the number of parameters of the stored procedure and the data type of each parameter, and optionally, the name of each parameter. A parameter for a stored procedure can be used only for input, only for output, or for both input and output. If an error is returned by the procedure, OUT parameters are undefined and INOUT parameters are unchanged.

All parameters are nullable except for numeric parameters in Java procedures, where numeric parameters, other than the DECIMAL types are not nullable in order to conform to the SQL/JRT standard.

IN Identifies the parameter as an input parameter to the procedure. The value of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT

Identifies the parameter as an output parameter that is returned by the stored procedure.

INOUT

Identifies the parameter as both an input and output parameter for the stored procedure.

parameter-name

Names the parameter for use as an SQL variable. The name cannot be the same as any other *parameter-name* for the procedure.

data-type

Specifies the data type of the parameter. The data type can be a built-in data type or a distinct type.

built-in-type

The data type of the parameter is a built-in data type.

For more information on the data types, see built-in-type.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

If you specify the name of the distinct type without a schema name, DB2 resolves the schema name by searching the schemas in the SQL path.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input argument on the CALL statement can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the procedure is called. With ASCII or EBCDIC, an error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

Parameters with a datetime data type or a distinct type are passed to the function as a different data type:

- A datetime type parameter is passed as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is the same as the implicitly or explicitly specified encoding scheme of any character or

graphic string parameters. If no character or graphic string parameters are passed, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

- A distinct type parameter is passed as the source type of the distinct type.

AS LOCATOR

Specifies that a locator to the value of the parameter is passed to the procedure instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to the procedure, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted.

TABLE LIKE *table-name* or *view-name* AS LOCATOR

Specifies that the parameter is a transition table. However, when the procedure is called, the actual values in the transition table are not passed to the stored procedure. A single value is passed instead. This single value is a locator to the table, which the procedure uses to access the columns of the transition table. A procedure with a table parameter can only be invoked from the triggered action of a trigger.

The use of TABLE LIKE provides an implicit definition of the transition table. It specifies that the transition table has the same number of columns as the identified table or view. If a table is specified, the transition table includes columns that are defined as implicitly hidden in the table. The columns have the same data type, length, precision, scale, subtype, and encoding scheme as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACES. The number of columns and the attributes of those columns are determined at the time the CREATE PROCEDURE statement is processed. Any subsequent changes to the number of columns in the table or the attributes of those columns do not affect the parameters of the procedure.

table-name or *view-name* must identify a table or view that exists at the current server. The name must not identify a declared temporary table. The table that is identified can contain XML columns; however, the procedure cannot reference those XML columns. The name does not have to be the same name as the table that is associated with the transition table for the trigger. An unqualified table or view name is implicitly qualified according to the following rules:

- If the CREATE PROCEDURE statement is embedded in a program, the implicit qualifier is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the implicit qualifier is the owner of the plan or package.
- If the CREATE PROCEDURE statement is dynamically prepared, the implicit qualifier is the SQL authorization ID in the CURRENT SCHEMA special register.

When the procedure is called, the corresponding columns of the transition table identified by the table locator and the table or view identified in the TABLE LIKE clause must have the same definition. The data type, length, precision, scale, and encoding scheme of these columns must match exactly. The description of the table or view at the time the CREATE PROCEDURE statement was executed is used.

Additionally, a character FOR BIT DATA column of the transition table cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is not defined as character FOR BIT DATA. (The definition occurs with the CREATE PROCEDURE statement.) Likewise, a character column of the transition table that is not FOR BIT DATA cannot be passed as input for a table parameter for which the corresponding column of the table specified at the definition is defined as character FOR BIT DATA.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

FENCED

Specifies that the procedure runs in an external address space.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of query result sets that the stored procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that there are no result sets. The value must be between 0 and 32767.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE

Specifies whether the procedure can be run in debugging mode. When DYNAMICRULES run behavior is in effect, the default is determined by using the value of the CURRENT DEBUG MODE special register. Otherwise the default is DISALLOW DEBUG MODE.

Do not specify this option unless LANGUAGE JAVA is in effect.

ALLOW DEBUG MODE

Specifies that the JAVA procedure can be run in debugging mode.

DISALLOW DEBUG MODE

Specifies that the JAVA procedure cannot be run in debugging mode.

You can use an ALTER PROCEDURE statement to change this option to ALLOW DEBUG MODE.

DISABLE DEBUG MODE

Specifies that the JAVA procedure can never be run in debugging mode.

The procedure cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE once the procedure has been created or altered using DISABLE DEBUG MODE. To change this option, you must drop and recreate the procedure using the desired option.

PARAMETER CCSID or PARAMETER VARCHAR

Specifies the encoding scheme for string parameters, and in the case of LANGUAGE C, specifies the representation of variable length string parameters.

CCSID

Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme to be used for system-generated parameters of the routine such as message tokens and DBINFO.

VARCHAR

Specifies that the representation of the values of varying length character string-parameters for procedures that specify LANGUAGE C.

This option can only be specified if LANGUAGE C is also specified.

NULTERM

Specifies that variable length character string parameters are represented in a NUL-terminated string form.

STRUCTURE

Specifies that variable length character string parameters are represented in a VARCHAR structure form.

Using the PARAMETER VARCHAR clause, there is no way to specify the VARCHAR form of an individual parameter as there is with PARAMETER CCSID. The PARAMETER VARCHAR clause only applies to parameters in the parameter list of a procedure and in the RETURNS clause. It does not apply to system-generated parameters of the routine such as message tokens and DBINFO.

In a data sharing environment, you should not specify the PARAMETER VARCHAR clause until all members of the data sharing group support the clause. If some group members support this clause and others do not, and PARAMETER VARCHAR is specified in an external routine, the routine will encounter different parameter forms depending on which group member invokes the routine.

EXTERNAL

Specifies that the CREATE PROCEDURE statement is being used to define a new procedure that is based on code written in an external programming language. If the NAME clause is not specified, 'NAME *procedure-name*' is assumed. The NAME clause is required for a LANGUAGE JAVA procedure because the default name is not valid for a Java procedure. In some cases, the default name will not be valid. To avoid invalid names, specify the NAME clause for the following types of procedures:

- A procedure that is defined as LANGUAGE JAVA
- A procedure that has a name that is greater than 8 bytes in length, contains an underscore, or does not conform to the rules for an ordinary identifier.

NAME *string* or *identifier*

Identifies the user-written code that implements the stored procedure.

If LANGUAGE is JAVA, *string* must be specified and enclosed in single quotation marks, with no extraneous blanks within the single quotation marks. It must specify a valid *external-java-routine-name*. If multiple *strings* are specified, the total length of all of them must not be greater than 1305 bytes and they must be separated by a space or a line break.

An *external-java-routine-name* contains the following parts:

jar-name

Identifies the name given to the JAR file when it was installed in the database. The name contains *jar-id*, which can optionally be qualified

with a schema. Examples are "myJar" and "mySchema.myJar." The unqualified *jar-id* is implicitly qualified with a schema name according to the following rules:

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER bind option when the package or plan was created or last rebound. If the QUALIFIER was not specified, the schema name is the owner of the package or plan.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SCHEMA special register.

If *jar-name* is specified, it must exist when the CREATE PROCEDURE statement is processed. Do not specify a *jar-name* for a JAVA procedure for which NO SQL is also specified.

If *jar-name* is not specified, the procedure is loaded from the class file directly instead of being loaded from a JAR file. DB2 searches the directories in the CLASSPATH associated with the WLM Environment. Environmental variables for Java routines are specified in a data set identified in a JAVAENV DD card on the JCL used to start the address space for a WLM-managed stored procedure.

method-name

Identifies the name of the method and must not be longer than 254 bytes. Its package, class, and method ID's are specific to Java and as such are not limited to 18 bytes. In addition, the rules for what these can contain are not necessarily the same as the rules for an SQL ordinary identifier.

package-id

Identifies a package. The concatenated list of *package-ids* identifies the package that the class identifier is part of. If the class is part of a package, the method name must include the complete package prefix, such as "myPacks.StoredProcs." The Java virtual machine looks in the directory "/myPacks/StoredProcs/" for the classes.

class-id

Identifies the class identifier of the Java object.

method-id

Identifies the method identifier with the Java class to be invoked.

method-signature

Identifies a list of zero or more Java data types for the parameter list and must not be longer than 1024 bytes. Specify the *method-signature* if the procedure involves any input or output parameters that can be NULL. When the stored procedure being created is called, DB2 searches for a Java method with the exact *method-signature*. The number of *java-datatype* elements specified indicates how many parameters that the Java method must have.

A Java procedure can have no parameters. In this case, you code an empty set of parentheses for *method-signature*. If a Java *method-signature* is not specified, DB2 searches for a Java method with a signature derived from the default JDBC types associated with the SQL types specified in the parameter list of the CREATE PROCEDURE statement.

For other values of LANGUAGE, the value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

LANGUAGE

This mandatory clause is used to specify the language interface convention to which the procedure body is written. All programs must be designed to run in the server's environment. Assembler, C, COBOL, and PL/I programs must be designed to run in IBM's Language Environment.

ASSEMBLE

The stored procedure is written in Assembler.

C The stored procedure is written in C or C++.

COBOL

The stored procedure is written in COBOL, including the OO-COBOL language extensions.

JAVA

The stored procedure is written in Java and is executed in the Java Virtual Machine. When LANGUAGE JAVA is specified, the EXTERNAL NAME clause must be specified with a valid *external-java-routine-name* and PARAMETER STYLE must be specified with JAVA. The procedure must be a public static method of the specified Java class.

Do not specify LANGUAGE JAVA when DBINFO, PROGRAM TYPE MAIN, or RUN OPTIONS is specified.

PLI

The stored procedure is written in PL/I.

REXX

The stored procedure is written in REXX. Do not specify LANGUAGE REXX when PARAMETER STYLE SQL is in effect. When REXX is specified, the procedure must use PARAMETER STYLE GENERAL or GENERAL WITH NULLS.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL

Specifies which SQL statements, if any, can be executed in the procedure or any routine that is called from this procedure. The default is MODIFIES SQL DATA. For the data access classification of each statement, see Table 138 on page 1605.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

READS SQL DATA

Specifies that the procedure can execute statements with a data access indication of READS SQL DATA, CONTAINS SQL, or NO SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the procedure can execute only SQL statements with an access indication of CONTAINS SQL or NO SQL. The procedure cannot execute statements that read or modify data.

NO SQL

Specifies that the procedure can execute only SQL statements with a data access classification of NO SQL. Do not specify NO SQL for a JAVA procedure that uses a JAR file.

PARAMETER STYLE

Identifies the linkage convention used to pass parameters to and return values from the stored procedure. All of the linkage conventions provide arguments to

the stored procedure that contain the parameters specified on the CALL statement. Some of the linkage conventions pass additional arguments to the stored procedure that provide more information to the stored procedure. For more information on linkage conventions, see *DB2 Application Programming and SQL Guide*.

SQL

Specifies that, in addition to the parameters on the CALL statement, several additional parameters are passed to the stored procedure. The following parameters are passed:

- The first *n* parameters that are specified on the CREATE PROCEDURE statement.
- *n* parameters for indicator variables for the parameters.
- The SQLSTATE to be returned.
- The qualified name of the stored procedure.
- The specific name of the stored procedure.
- The SQL diagnostic string to be returned to DB2.
- If DBINFO is specified, the DBINFO structure.

PARAMETER STYLE SQL is the default. Do not specify PARAMETER STYLE SQL when LANGUAGE REXX or LANGUAGE JAVA is in effect.

GENERAL

Specifies that the stored procedure uses a parameter passing mechanism where the stored procedure receives only the parameters specified on the CALL statement. Arguments to procedures defined with this parameter style cannot be null.

GENERAL WITH NULLS

Specifies that, in addition to the parameters on the CALL statement as specified in GENERAL, another argument is also passed to the stored procedure. The additional argument contains an indicator array with an element for each of the parameters on the CALL statement. In C, this is an array of short INTS. The indicator array enables the stored procedure to accept or return null parameter values.

JAVA

Specifies that the stored procedure uses a parameter passing convention that conforms to the Java and SQLJ Routines specifications. PARAMETER JAVA can be specified only if LANGUAGE is JAVA. JAVA must be specified for PARAMETER STYLE when LANGUAGE is JAVA.

INOUT and OUT parameters are passed as single-entry arrays. The INOUT and OUT parameters are declared in the Java method as single-element arrays of the Java type.

For REXX stored procedures (LANGUAGE REXX), GENERAL and GENERAL WITH NULLS are the only valid values for PARAMETER STYLE; therefore, specify one of these values and do not allow PARAMETER STYLE to default to SQL.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the stored procedure returns the same results each time the stored procedure is called with the same IN and INOUT arguments.

DETERMINISTIC

The stored procedure always returns the same results each time the stored

procedure is called with the same IN and INOUT arguments, if the referenced data in the database has not changed.

NOT DETERMINISTIC

The stored procedure might not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed. NOT DETERMINISTIC is the default.

DB2 does not verify that the stored procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

NO PACKAGE PATH or PACKAGE PATH *package-path*

Specifies the package path to use when the procedure is run. This is the list of the possible package collections into which the DBRM this is associated with the procedure is bound.

NO PACKAGE PATH

Specifies that the list of package collections for the procedure is the same as the list of package collection IDs for the calling program. If the calling program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For information about how DB2 uses these three items, see *DB2 Application Programming and SQL Guide*.

PACKAGE PATH *package-path*

Specifies a list of package collections, in the same format as the SET CURRENT PACKAGE PATH special register.

If the COLLID clause is specified with PACKAGE PATH, the COLLID clause is ignored when the routine is invoked.

The *package-path* value that is provided when the procedure is created is checked when the CALL statement is prepared. If *package-path* contains SESSION_USER (or USER), PATH, or PACKAGE PATH, an error is returned when the *package-path* value is checked.

NO DBINFO or DBINFO

Specifies whether additional status information is passed to the stored procedure when it is invoked.

NO DBINFO

Additional information is not passed. NO DBINFO is the default.

DBINFO

An additional argument is passed when the stored procedure is invoked. The argument is a structure that contains information such as the name of the current server, the application runtime authorization ID and identification of the version and release of the database manager that invoked the procedure. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

DBINFO can be specified only if PARAMETER STYLE SQL is specified.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is to be used when the stored procedure is executed. This is the package collection into which the DBRM that is associated with the stored procedure is bound.

NO COLLID

The package collection for the stored procedure is the same as the package

collection of the calling program. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*

The package collection for the stored procedure is the one specified.

For REXX stored procedures, *collection-id* can be DSNREXRR, DSNREXRS, DSNREXCR, or DSNREXCS.

WLM ENVIRONMENT

Identifies the WLM (workload manager) environment in which the stored procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the stored procedure runs in the default WLM-established stored procedure address space specified at installation time.

name

The WLM environment in which the stored procedure must run. If another stored procedure or a user-defined function calls the stored procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the stored procedure request to a different address space.

(name,*)

When an SQL application program directly calls a stored procedure, the WLM environment in which the stored procedure runs.

If another stored procedure or a user-defined function calls the stored procedure, the stored procedure runs in the same WLM environment that the calling routine uses.

To define a stored procedure that is to run in a specified WLM environment, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see *Running stored procedures*.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of a stored procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table. This option is ignored if LANGUAGE JAVA is specified.

When you are debugging a stored procedure, setting a limit can be helpful in case the stored procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the service units. NO LIMIT is the default.

LIMIT *integer*

The limit on the service units is a positive *integer* in the range of 1 to 2 147 483 647. If the stored procedure uses more service units than the specified value, DB2 cancels the stored procedure.

STAY RESIDENT

Specifies whether the stored procedure load module is to remain resident in memory when the stored procedure ends. This option is ignored if LANGUAGE JAVA is specified.

NO The load module is deleted from memory after the stored procedure ends. NO is the default.

YES

The load module remains resident in memory after the stored procedure ends.

PROGRAM TYPE

Specifies whether the stored procedure runs as a main routine or a subroutine.

SUB

The stored procedure runs as a subroutine. With LANGUAGE JAVA, PROGRAM TYPE SUB is the only valid option.

MAIN

The stored procedure runs as a main routine. With LANGUAGE REXX, PROGRAM TYPE MAIN is always in effect.

The default for PROGRAM TYPE is:

- MAIN with LANGUAGE REXX
- SUB with LANGUAGE JAVA
- For other languages, the default depends on the value of the CURRENT RULES special register:
 - MAIN when the value is DB2
 - SUB when the value is STD

SECURITY

Specifies how the stored procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2

The stored procedure does not require a special external security environment. If the stored procedure accesses resources that an external security product protects, the access is performed using the authorization ID associated with the stored procedure address space. DB2 is the default.

USER

An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the stored procedure.

DEFINER

An external security environment should be established for the stored procedure. If the stored procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the stored procedure.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE

Specifies whether the routine is to be put in a stopped state after some number of failures.

STOP AFTER SYSTEM DEFAULT FAILURES

Specifies that this routine should be placed in a stopped state after the

number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX. This is the default.

STOP AFTER *nn* FAILURES

Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE

Specifies that this routine should not be placed in a stopped state after any failure.

RUN OPTIONS *runtime-options*

Specifies the Language Environment runtime options to be used for the stored procedure. For a REXX stored procedure, specifies the Language Environment runtime options to be passed to the REXX language interface to DB2. You must specify *runtime-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any runtime options to Language Environment, and Language Environment uses its installation defaults.

Do not specify RUN OPTIONS when LANGUAGE JAVA is in effect.

For a description of the Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

COMMIT ON RETURN

Indicates whether DB2 commits the transaction immediately on return from the stored procedure.

NO DB2 does not issue a commit when the stored procedure returns. NO is the default.

YES

DB2 issues a commit when the stored procedure returns if the following statements are true:

- The SQLCODE that is returned by the CALL statement is not negative.
- The stored procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the stored procedure.

If the stored procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine. The default is INHERIT SPECIAL REGISTERS.

INHERIT SPECIAL REGISTERS

Specifies that the values of special registers are inherited according to the rules listed in the table for characteristics of special registers in a stored procedure in Table 37 on page 151.

DEFAULT SPECIAL REGISTERS

Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a stored procedure in Table 37 on page 151.

CALLED ON NULL INPUT

Specifies that the procedure is to be called even if any or all argument values

are null, which means that the procedure must be coded to test for null argument values. The procedure can return null or nonnull values. CALLED ON NULL INPUT is the default.

Notes

Owner privileges: The owner is authorized to call the procedure (EXECUTE privilege) and grant others the privilege to call the procedure. See “GRANT (function or procedure privileges)” on page 1340. For more information about ownership of the object, see “Authorization, privileges, and object ownership” on page 60.

Choosing data types for parameters: When you choose the data types of the parameters for your stored procedure, consider the rules of promotion that can affect the values of the parameters. (See “Promotion of data types” on page 95). For example, a constant that is one of the input arguments to the stored procedure might have a built-in data type that is different from the data type that the procedure expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, using the following data types for parameters is recommended:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- VARBINARY instead of BINARY

For portability of functions across platforms that are not DB2 for z/OS, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying the encoding scheme for parameters: The encoding scheme of all of the parameters with a character or graphic string data type (both input and output parameters) must be the same—either all ASCII, all EBCDIC, or all UNICODE. If you specify the encoding scheme on the individual parameters, instead of using the PARAMETER CCSID to specify it for all parameters at once or allowing the encoding scheme to default to the system value, ensure that they all agree.

Character string representation considerations: The PARAMETER VARCHAR clause is specific to LANGUAGE C routines because of the native use of NUL-terminated strings in C. VARCHAR structure representation is useful when character string data is known to contain embedded NUL-terminators. It is also useful when it cannot be guaranteed that character string data does not contain embedded NUL-terminators.

PARAMETER VARCHAR does not apply to fixed length character strings, VARCHAR FOR BIT DATA, CLOB, DBCLOB, or implicitly generated parameters. The clause does not apply to VARCHAR FOR BIT DATA because BIT DATA can contain X'00' characters, and its value representation starts with length information. It does not apply to LOB data because a LOB value representation starts with length information.

PARAMETER VARCHAR does not apply to optional parameters that are implicitly provided to an external procedure. For example, a CREATE PROCEDURE statement for LANGUAGE C must also specify PARAMETER STYLE SQL, which returns an SQLSTATE NUL-terminated character string; that SQLSTATE will not be represented in VARCHAR structured form. Likewise, none of the parameters that represent the qualified name of the procedure, the specific name of the procedure, or the SQL diagnostic string that is returned to the database manager will be represented in VARCHAR structured form.

Running stored procedures: You can use the WLM ENVIRONMENT clause to identify the address space in which a stored procedure is to run. Using different WLM environments lets you isolate one group of programs from another. For example, you might choose to isolate programs based on security requirements and place all payroll applications in one WLM environment because those applications deal with sensitive data, such as employee salaries.

Regardless of where the stored procedure is to run, DB2 invokes RACF to determine whether you have appropriate authorization. You must have authorization to issue CREATE PROCEDURE statements that refer to the specified WLM environment or the DB2-established stored procedure address space. For example, the following RACF command authorizes DB2 user DB2USER1 to define stored procedures on DB2 subsystem DB2A that run in the WLM environment named PAYROLL.

```
PERMIT DB2A.WLMENV.PAYROLL CLASS(DSNR) ID(DB2USER1) ACCESS(READ)
```

Accessing result sets from nested stored procedures: When another stored procedure or a user-defined function calls a stored procedure, only the calling routine can access the result sets that the stored procedure returns. The result sets are not returned to the application that contains the outermost stored procedure or user-defined function in the sequence of nested calls.

When a stored procedure is nested, the result sets that are returned by the stored procedure are accessible only by the calling routine. The result sets are not returned to the application that contains the outermost stored procedure or user-defined function in the sequence of nested calls.

Restrictions for nested stored procedures: A stored procedure, user-defined function, or trigger cannot call a stored procedure that is defined with the COMMIT ON RETURN clause.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET as a synonym for DYNAMIC RESULT SET
- RESULT SETS as a synonym for DYNAMIC RESULT SETS
- STANDARD CALL as a synonym for DB2SQL
- SIMPLE CALL as a synonym for GENERAL
- SIMPLE CALL WITH NULLS as a synonym for GENERAL WITH NULLS
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NULL CALL as a synonym for CALLED ON NULL INPUT
- PARAMETER STYLE DB2SQL as a synonym for PARAMETER STYLE SQL

Examples

Example 1: Create the definition for a stored procedure that is written in COBOL. The procedure accepts an assembly part number and returns the number of parts that make up the assembly, the total part cost, and a result set. The result set lists the part numbers, quantity, and unit cost of each part. Assume that the input parameter cannot contain a null value and that the procedure is to run in a WLM environment called PARTSA.

```
CREATE PROCEDURE SYSPROC.MYPROC(IN INT, OUT INT, OUT DECIMAL(7,2))
  LANGUAGE COBOL
  EXTERNAL NAME MYMODULE
  PARAMETER STYLE GENERAL
  WLM ENVIRONMENT PARTSA
  DYNAMIC RESULT SETS 1;
```

Example 2: Create the definition for the stored procedure described in Example 1, except use the linkage convention that passes more information than the parameter specified on the CALL statement. Specify Language Environment runtime options HEAP, BELOW, ALL31, and STACK.

```
CREATE PROCEDURE SYSPROC.MYPROC(IN INT, OUT INT, OUT DECIMAL(7,2))
  LANGUAGE COBOL
  EXTERNAL NAME MYMODULE
  PARAMETER STYLE SQL
  WLM ENVIRONMENT PARTSA
  DYNAMIC RESULT SETS 1
  RUN OPTIONS 'HEAP(,,ANY),BELOW(4K,,),ALL31(ON),STACK(,,ANY,)' ;
```

Example 3: Create the procedure definition for a stored procedure, written in Java, that is passed a part number and returns the cost of the part and the quantity that is currently available.

```
CREATE PROCEDURE PARTS_ON_HAND(IN PARTNUM INT,
                                OUT COST DECIMAL(7,2),
                                OUT QUANTITY INT)
  LANGUAGE JAVA
  EXTERNAL NAME 'PARTS.ONHAND'
  PARAMETER STYLE JAVA;
```

CREATE PROCEDURE (SQL - external)

The CREATE PROCEDURE statement defines an external SQL procedure at the current server and specifies the source statements for the procedure. This is the only type of SQL procedure that is available for versions of DB2 prior to Version 9.

For information about the SQL control statements that are supported in external SQL procedures, refer to “SQL control statements for external SQL procedures” on page 1610.

Invocation

This statement can only be dynamically prepared, but the DYNAMICRULES run behavior must be specified implicitly or explicitly. It is intended to be processed using one of the following methods:

- JCL
- The DB2 for z/OS SQL procedure processor (DSNTPSMP) (IBM Optim™ Development Studio uses this method.)

Issuing the CREATE PROCEDURE statement from another context will result in an incomplete procedure definition even though the statement processing returns without error. For more information on preparing SQL procedures for execution, see *DB2 Application Programming and SQL Guide*.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

The authorization ID that is used to create the stored procedure must have authority to create programs that are to be run in the specified WLM environment.

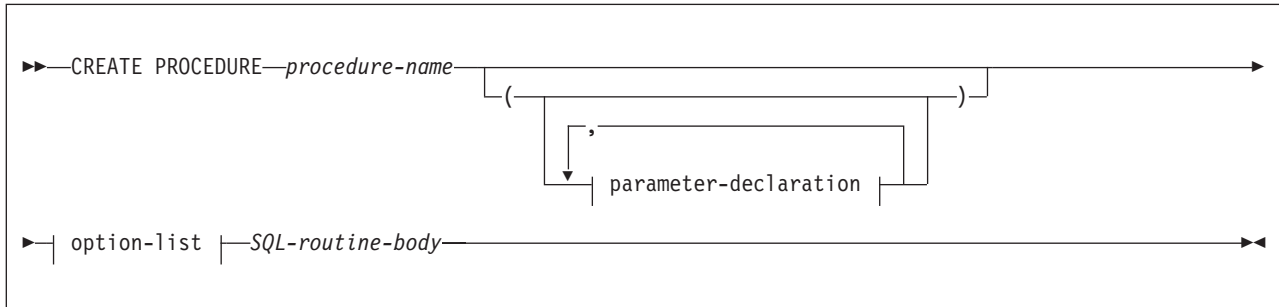
The owner of the procedure is determined by how the CREATE PROCEDURE statement is invoked:

- If the statement is embedded in a program, the owner is the authorization ID of the owner of the plan or package.

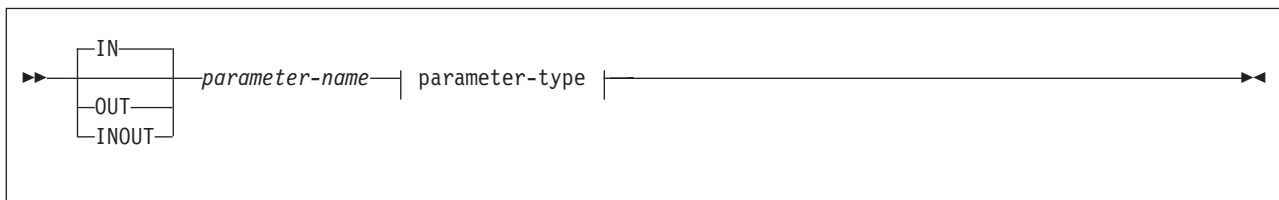
- If the statement is dynamically prepared, the owner is the SQL authorization ID in the CURRENT SQLID special register.

The owner is implicitly given the EXECUTE privilege with the GRANT option for the procedure.

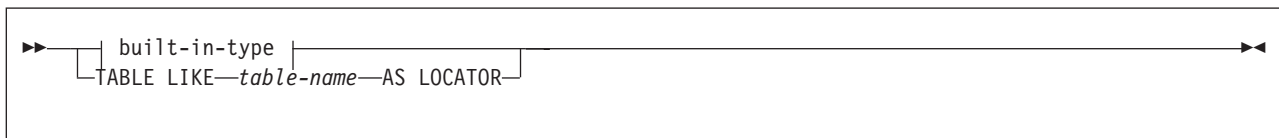
Syntax



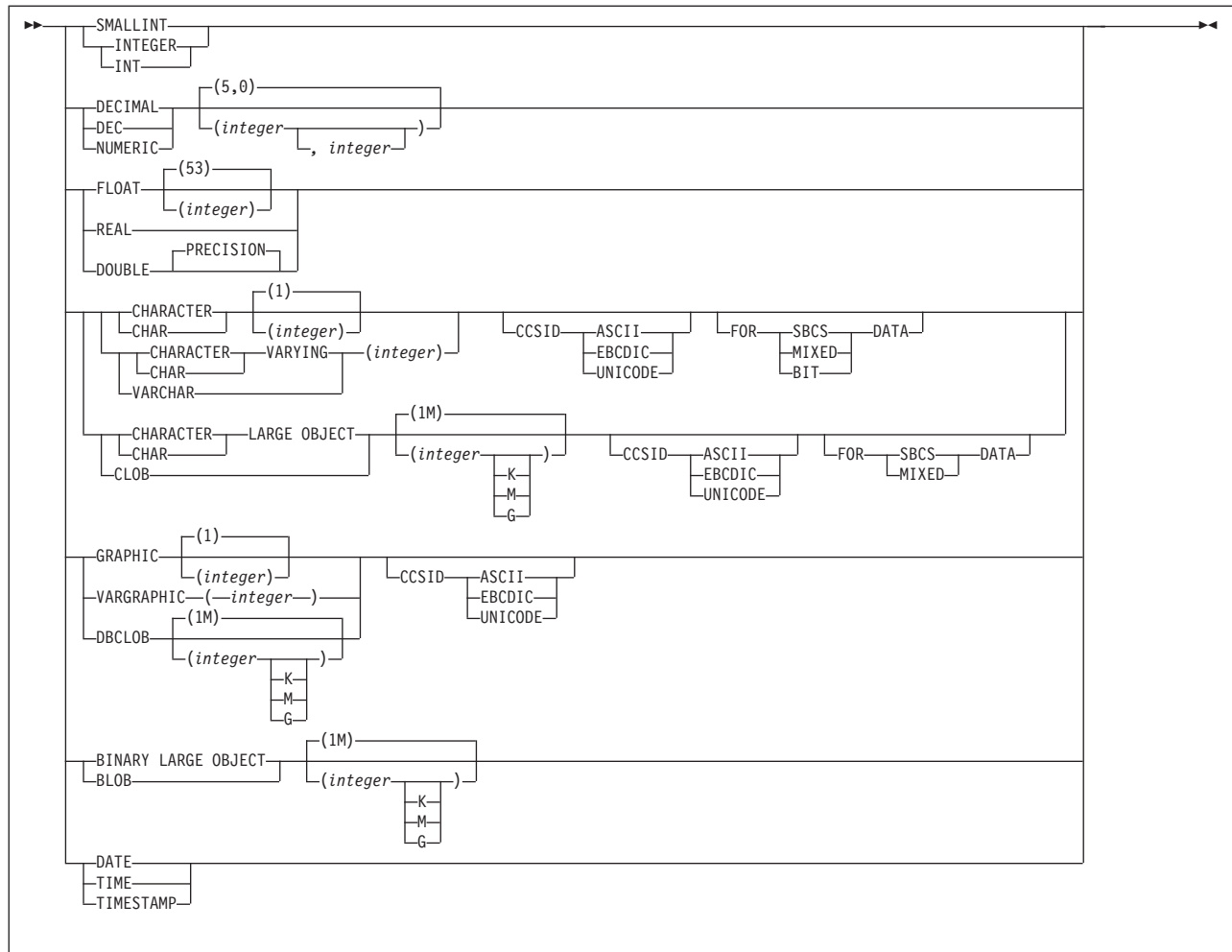
parameter-declaration:



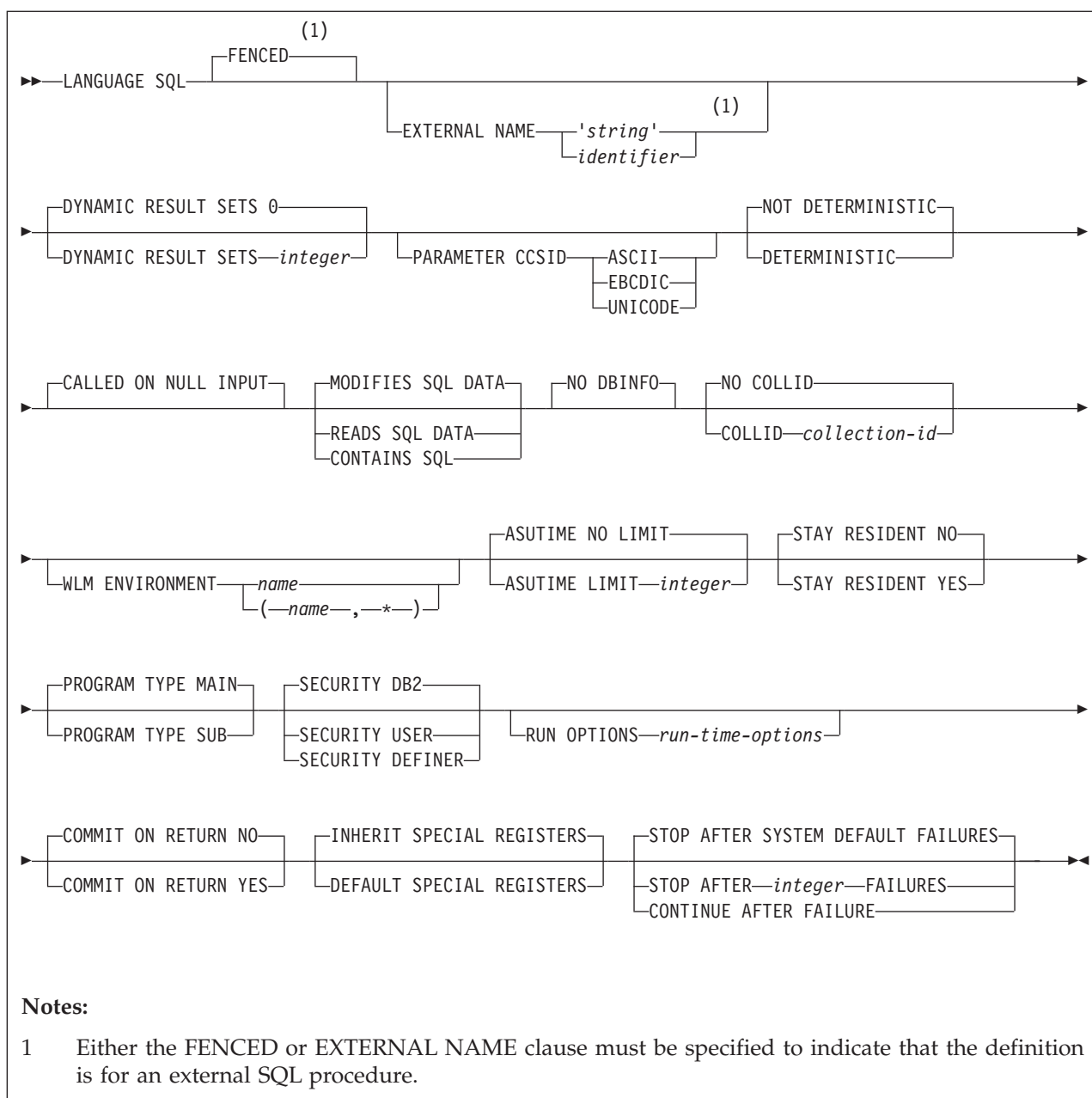
parameter-type:



built-in-type:



option-list: (The options can be specified in any order, but each option can be specified only one time)



Description

procedure-name

Names the procedure. The name, including the implicit or explicit qualifier, must not identify an existing stored procedure at the current server.

(parameter-declaration,...)

Specifies the number of parameters of the procedure, the data type of each parameter, and the name of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. If an error is returned by the procedure, OUT parameters are undefined, and INOUT parameters are unchanged. All of the parameters are nullable.

IN Identifies the parameter as an input parameter to the procedure. The value

of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT

Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT

Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned.

parameter-name

Names the parameter for use as an SQL variable. *parameter-name* is an SQL identifier and must not be a delimited identifier that includes lowercase letters or special characters. A parameter name cannot be the same as the name of any other parameter for this version of the procedure.

parameter-type

Specifies the data type of the parameter.

built-in-type

The data type of the parameter is a built-in data type.

For more information on the data types, including the subtype of character data types (the FOR *subtype* DATA clause), see built-in-type. For external SQL procedures, the maximum limit for VARCHAR is 32767 and for VARGRAPHIC is 16382.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input parameter can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the procedure is called. With ASCII or EBCDIC, an error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

A parameter with a datetime data type is passed to the SQL procedure as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is determined as follows:

- If there are one or more parameters with a character or graphic data type, the encoding scheme of the datetime type parameter is the same as the encoding scheme of the character or graphic parameters.
- Otherwise, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

TABLE LIKE *table-name* **AS LOCATOR**

Specifies that the parameter is a transition table. However, when the

procedure is called, the actual values in the transition table are not passed to the procedure. A single value is passed instead. This single value is a locator to the table, which the procedure uses to access the columns of the transition table. A procedure with a table parameter can only be invoked from the triggered action of a trigger.

The transition table includes columns that are defined as implicitly hidden in the table. The table that is identified can contain XML columns; however, the procedure cannot reference those XML columns.

For more information about the TABLE LIKE clause, see TABLE LIKE. For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

LANGUAGE

Specifies the application programming language in which the procedure is written.

SQL

The procedure is written in DB2 SQL procedural language.

FENCED

Specifies that the procedure runs in an external address space. FENCED also specifies that the SQL procedure program is an MVS load module with an external name.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of query result sets that the procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that the procedure can return no result sets. The value of *integer* must be between 0 and 32767.

PARAMETER CCSID

Indicates whether the encoding scheme for character and graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value that is specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character and graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value that is specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme that is to be used for system-generated parameters of the routine such as message tokens and DBINFO.

EXTERNAL NAME '*string*' or *identifier*

Specifies the name of the MVS load module for the program that runs when the procedure name is specified in an SQL CALL statement. The value must conform to the naming conventions for MVS load modules: the value must be less than or equal to 8 bytes, and it must conform to the rules for an ordinary identifier with the exception that it must not contain an underscore.

EXTERNAL NAME *procedure-name* is the default. In some cases, the default name will not be valid. To avoid an invalid name, specify EXTERNAL NAME for a procedure that has a name that is greater than 8 bytes in length, contains an underscore, or does not conform to the rules for an ordinary identifier.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

NOT DETERMINISTIC

The procedure might not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

NOT DETERMINISTIC is the default.

DETERMINISTIC

The procedure always returns the same results each time the stored procedure is called with the same IN and INOUT arguments, if the referenced data in the database has not changed.

DB2 does not verify that the procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

CALLED ON NULL INPUT

Specifies that the procedure is to be called even if any or all argument values are null, which means that the procedure must be coded to test for null argument values. The procedure can return null or non-null values.

CALLED ON NULL INPUT is the default.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL

Specifies the classification of SQL statements that the procedure can execute. For the data access classification of each statement, see Table 138 on page 1605. Statements that are not supported in any procedure return an error.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

MODIFIES SQL DATA is the default.

READS SQL DATA

Specifies that the procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the procedure can execute only SQL statements with a data access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

NO DBINFO

Specifies that no additional status information that is known by DB2 is passed to the procedure when it is invoked.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is to be used when the procedure is executed. This is the package collection into which the DBRM that is associated with the procedure is bound.

NO COLLID

Specifies that the package collection for the procedure is the same as the package collection of the calling program. If the invoking program does not use a package, DB2 resolves the package by using the CURRENT PACKAGE PATH special register, the CURRENT PACKAGESET special register, or the PKLIST bind option (in this order). For details about how

DB2 uses these three items, see the information on package resolution in *DB2 Application Programming and SQL Guide*.

NO COLLID is the default.

COLLID *collection-id*

Specifies the package collection for the procedure.

WLM ENVIRONMENT *name* **or** (*name*,*)

Identifies the WLM (workload manager) environment in which the stored procedure is to run when the DB2 stored procedure address space is WLM-established. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT, the procedure runs in the default WLM-established stored procedure address space that is specified at installation time.

name

The WLM environment in which the procedure must run. If another procedure or a user-defined function calls the procedure and that calling routine is running in an address space that is not associated with the specified WLM environment, DB2 routes the procedure request to a different address space.

(name,*)

When an SQL application program directly calls a procedure, *name* specifies the WLM environment in which the procedure runs.

If another procedure or a user-defined function calls the stored procedure, the procedure runs in the same WLM environment that the calling routine uses.

To define a procedure that is to run in a specified WLM environment, you must have appropriate authority for the WLM environment. For an example of a RACF command that provides this authorization, see *Running stored procedures*.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of a procedure can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a procedure, setting a limit can be helpful in case the procedure gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

There is no limit on the number of CPU service units that the procedure can run.

NO LIMIT is the default.

LIMIT *integer*

The limit on the number of CPU service units is a positive *integer* in the range of 1 to 2 147 483 647. If the procedure uses more service units than the specified value, DB2 cancels the procedure.

STAY RESIDENT

Specifies whether the load module for the procedure remains resident in memory when the procedure ends.

NO The load module is deleted from memory after the procedure ends.

NO is the default.

YES

The load module remains resident in memory after the procedure ends.

PROGRAM TYPE

Specifies whether the procedure runs as a main routine or a subroutine.

MAIN

The procedure runs as a main routine.

MAIN is the default.

SUB

The procedure runs as a subroutine.

SECURITY

Specifies how the procedure interacts with an external security product, such as RACF, to control access to non-SQL resources.

DB2

The procedure does not require a special external security environment. If the procedure accesses resources that an external security product protects, the access is performed using the authorization ID that is associated with the address space in which the procedure runs.

DB2 is the default.

USER

An external security environment should be established for the procedure. If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the user who invoked the procedure.

DEFINER

An external security environment should be established for the procedure. If the procedure accesses resources that the external security product protects, the access is performed using the authorization ID of the owner of the procedure.

RUN OPTIONS *run-time-options*

Specifies the Language Environment runtime options that are to be used for the procedure. You must specify *run-time-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any runtime options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment runtime options, see *z/OS Language Environment Programming Reference*.

COMMIT ON RETURN

Indicates whether DB2 commits the transaction immediately on return from the procedure.

NO DB2 does not issue a commit when the procedure returns.

NO is the default.

YES

DB2 issues a commit when the procedure returns if the following statements are true:

- A positive SQLCODE is returned by the CALL statement.
- The procedure is not in a must abort state.

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a procedure in Table 37 on page 151.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS

Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a procedure in Table 37 on page 151.

STOP AFTER SYSTEM DEFAULT FAILURES, STOP AFTER *nn* FAILURES, or CONTINUE AFTER FAILURE

Specifies the routine is stopped after failures.

STOP AFTER SYSTEM DEFAULT FAILURES

Specifies that this routine should be placed in a stopped state after the number of failures indicated by the value of field MAX ABEND COUNT on installation panel DSNTIPX.

STOP AFTER SYSTEM DEFAULT FAILURES is the default.

STOP AFTER *nn* FAILURES

Specifies that this routine should be placed in a stopped state after *nn* failures. The value *nn* can be an integer from 1 to 32767.

CONTINUE AFTER FAILURE

Specifies that this routine should not be placed in a stopped state after any failure.

SQL-routine-body

Specifies the statements that define the body of the SQL procedure. For information on the SQL control statements that are supported in external SQL procedures, see “SQL control statements for external SQL procedures” on page 1610. For information on the SQL statements that are allowed in external SQL procedures, see “SQL statements allowed in SQL procedures” on page 1608.

Notes

See “Notes” on page 1032 for information about:

- Owner privileges
- Choosing data types for parameters
- Specifying the encoding scheme for parameters
- Environments for running stored procedures
- Accessing result sets from nested stored procedures

Error handling in SQL procedures: You should consider the possible exceptions that can occur for each SQL statement in the body of a procedure. Any exception

SQLSTATE that is not handled within the procedure using a handler within a compound statement results in the exception SQLSTATE being returned to the caller of the procedure.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET and RESULT SETS as synonyms for DYNAMIC RESULT SETS
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC

Examples

Example 1: Create the definition for an SQL procedure. The procedure accepts an employee number and a multiplier for a pay raise as input. The following tasks are performed in the procedure body:

- Calculate the employee's new salary.
- Update the employee table with the new salary value.

```
| CREATE PROCEDURE UPDATESALARY
| (IN EMPLOYEE_NUMBER CHAR(10),
| IN RATE DECIMAL(6,2))
| LANGUAGE SQL
| FENCED
| EXTERNAL NAME 'USALARY1'
| MODIFIES SQL DATA
| UPDATE EMP
| SET SALARY = SALARY * RATE
| WHERE EMPNO = EMPLOYEE_NUMBER
```

Example 2: Create the definition for the SQL procedure described in example 1, but specify that the procedure has these characteristics:

- The procedure runs in a WLM environment called PARTSA.
- The same input always produces the same output.
- SQL work is committed on return to the caller.
- The Language Environment runtime options to be used when the SQL procedure executes are 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'.

```
| CREATE PROCEDURE UPDATESALARY
| (IN EMPLOYEE_NUMBER CHAR(10),
| IN RATE DECIMAL(6,2))
| LANGUAGE SQL
| FENCED
| EXTERNAL NAME 'USALARY2'
| MODIFIES SQL DATA
| WLM ENVIRONMENT PARTSA
| DETERMINISTIC
| RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
| COMMIT ON RETURN YES
| UPDATE EMP
| SET SALARY = SALARY * RATE
| WHERE EMPNO = EMPLOYEE_NUMBER
```

For more examples of SQL procedures, see “SQL control statements for external SQL procedures” on page 1610.

CREATE PROCEDURE (SQL - native)

The CREATE PROCEDURE statement defines an SQL procedure at the current server and specifies the source statements for the procedure. You can define multiple versions of the procedure. CREATE PROCEDURE is used to define the initial version, and ALTER PROCEDURE is used to define subsequent versions.

For information about the SQL control statements that are supported in native SQL procedures, refer to Chapter 6, “SQL control statements for native SQL procedures,” on page 1541.

Invocation

This statement can only be dynamically prepared, and the DYNAMICRULES run behavior must be specified implicitly or explicitly.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema and the required authorization to add a new package or a new version of an existing package depending on the value of the BIND NEW PACKAGE field on installation panel DSNTIPP
- SYSADM authority
- SYSCTRL authority

The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: The privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process and the owner is that role.

If the statement is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the set of privileges that are held by the SQL authorization ID of the process. If the schema name is not the same as the SQL authorization ID of the process, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

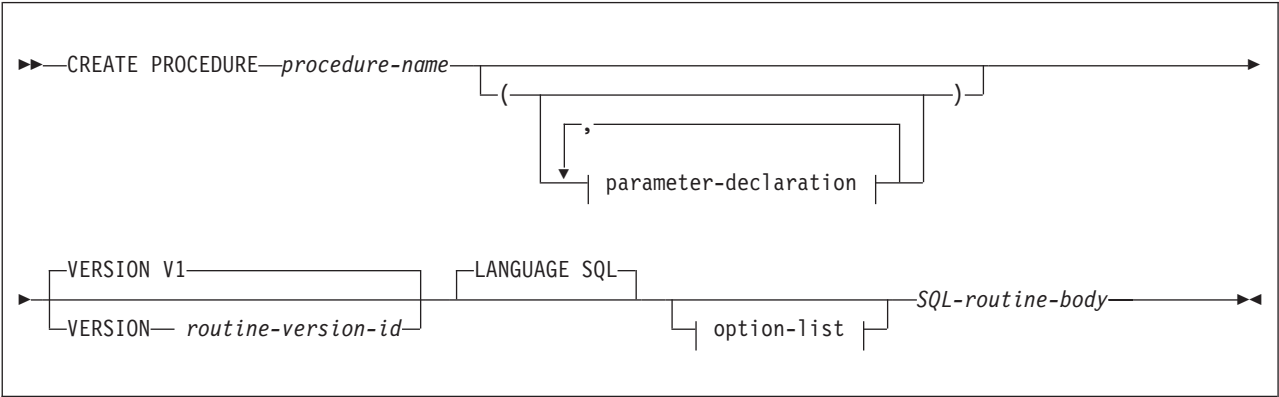
The privilege set must also include the privileges required to execute the statements in *SQL-routine-body*.

If the WLM ENVIRONMENT FOR DEBUG MODE clause is specified, the privilege set must have authority to define programs that run in the specified WLM environment.

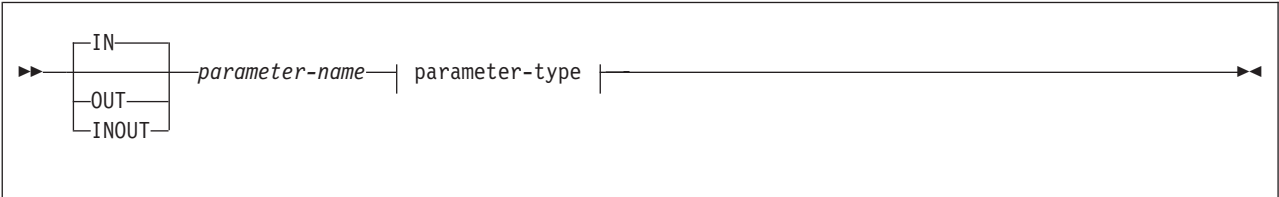
The owner of the procedure is the SQL authorization ID in the CURRENT SQLID special register unless the process is running within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the owner of the procedure is the role that is associated with the primary authorization ID of the process.

The owner is implicitly given the EXECUTE privilege with the GRANT option for the procedure.

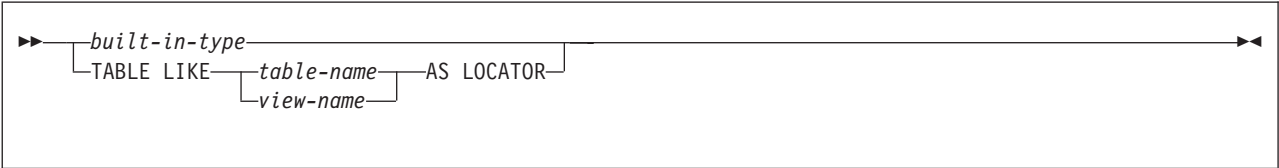
Syntax



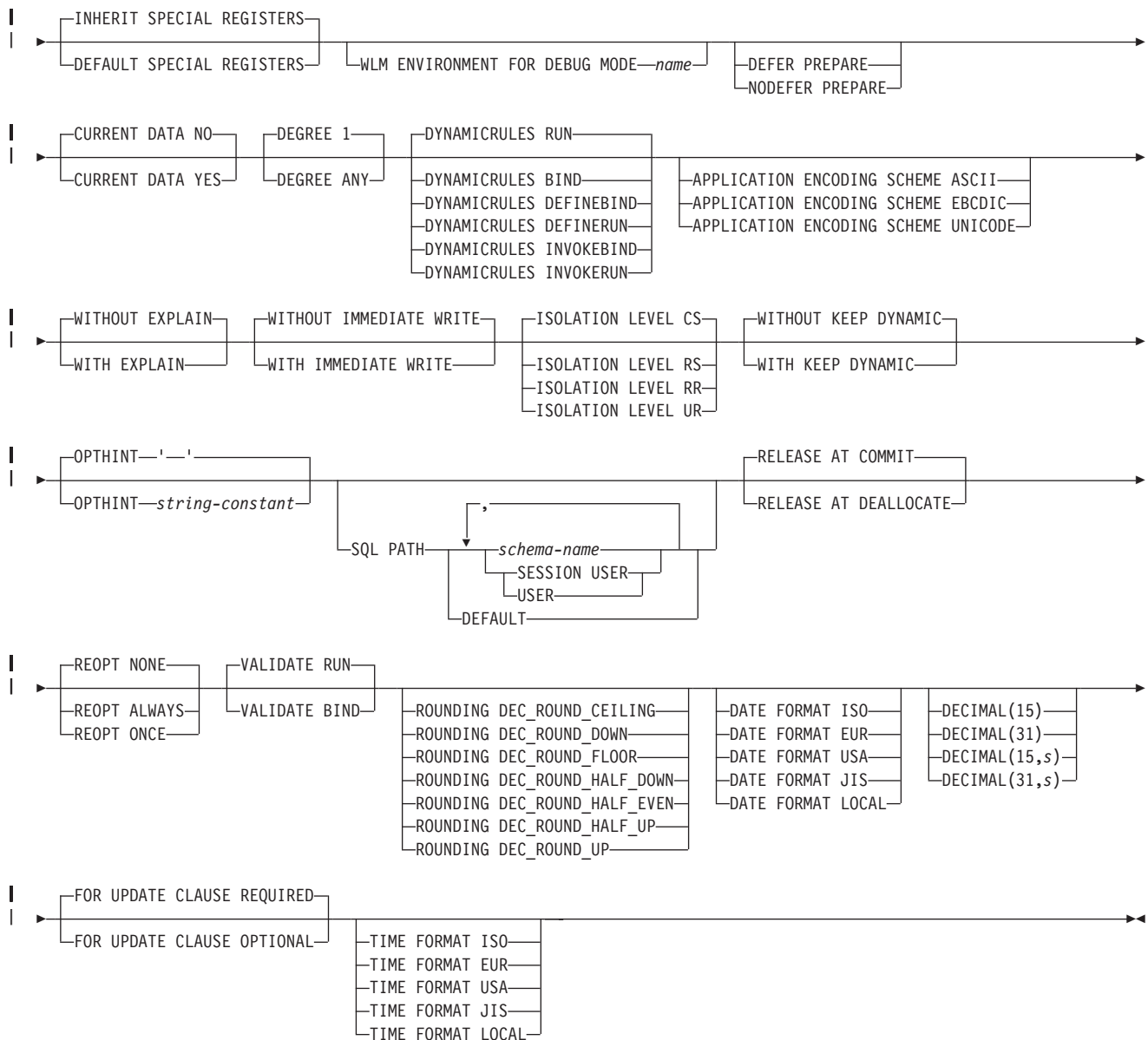
parameter-declaration:



parameter-type:



built-in-type:



Description

procedure-name

Names the procedure. If *procedure-name* already exists, an error is returned even if **VERSION** is specified with a *version-id* that is different from any existing version identifier for the procedure that is specified in *procedure-name*.

(parameter-declaration,...)

Specifies the number of parameters of the procedure, the data type and usage of each parameter, and the name of each parameter for the version of the procedure that is being defined. The number of parameters and the specified data type and usage of each parameter must match the data types in the corresponding position of the parameter for all other versions of this procedure. Synonyms for data types are considered to be a match.

IN, **OUT**, and **INOUT** specify the usage of the parameter. The usage of the parameters must match the implicit or explicit usage of the parameters of other versions of the same procedure.

IN Identifies the parameter as an input parameter to the procedure. The value of the parameter on entry to the procedure is the value that is returned to the calling SQL application, even if changes are made to the parameter within the procedure.

IN is the default.

OUT

Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT

Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned.

parameter-name

Names the parameter for use as an SQL variable. A parameter name cannot be the same as the name of any other parameter for this version of the procedure. The name of the parameter in this version of the procedure can be different than the name of the corresponding parameter for other versions of this procedure.

built-in-type

The data type of the parameter is a built-in data type.

For more information on the data types, including the subtype of character data types (the *FOR subtype DATA* clause), see *built-in-type*. However, the varying length string data types have different maximum lengths than for the CREATE TABLE statement. The maximum lengths for parameters (and SQL variables) are as follows: 32704 for VARCHAR or VARBINARY, and 16352 for VARGRAPHIC.

For parameters with a character or graphic data type, the PARAMETER CCSID clause or CCSID clause indicates the encoding scheme of the parameter. If you do not specify either of these clauses, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input parameter can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the procedure is called. With ASCII or EBCDIC, an error occurs if mixed data that actually contains DBCS characters is used as the value for an input parameter that is declared with an SBCS subtype.

A parameter with a datetime data type is passed to the SQL procedure as a character data type, and the data is passed in ISO format.

The encoding scheme for a datetime type parameter is determined as follows:

- If there are one or more parameters with a character or graphic data type, the encoding scheme of the datetime type parameter is the same as the encoding scheme of the character or graphic parameters.
- Otherwise, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

TABLE LIKE *table-name* AS LOCATOR

Specifies that the parameter is a transition table. However, when the procedure

is called, the actual values in the transition table are not passed to the procedure. A single value is passed instead. This single value is a locator to the table, which the procedure uses to access the columns of the transition table. The table that is identified can contain XML columns; however, the procedure cannot reference those XML columns. A procedure with a table parameter can only be invoked from the triggered action of a trigger. For additional information about using table locators, refer to *DB2 Application Programming and SQL Guide*.

VERSION *routine-version-id*

Specifies the version identifier for the first version of the procedure that is to be generated. See “Naming conventions” on page 51 for information about specifying *routine-version-id*. You can use an ALTER PROCEDURE statement with the ADD VERSION clause or the BIND DEPLOY command to create additional versions of the procedure.

V1 is the default version identifier.

See Versions of a procedure for more information about the use of versions for procedures.

LANGUAGE SQL

Specifies that the procedure is written in the DB2 SQL procedural language.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the procedure returns the same results each time it is called with the same IN and INOUT arguments.

DETERMINISTIC

The procedure always returns the same results each time it is called with the same IN and INOUT arguments if the data that is referenced in the database has not changed.

NOT DETERMINISTIC

The procedure might not return the same result each time it is called with the same IN and INOUT arguments, even when the data that is referenced in the database has not changed.

NOT DETERMINISTIC is the default.

DB2 does not verify that the procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL

Specifies the classification of SQL statements that the procedure can execute.

For the data access classification of each statement, see “SQL statements allowed in SQL procedures” on page 1608.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

MODIFIES SQL DATA is the default.

READS SQL DATA

Specifies that the procedure can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The procedure cannot execute SQL statements that modify data.

CONTAINS SQL

Specifies that the procedure can execute only SQL statements with a data access indication of CONTAINS SQL. The procedure cannot execute statements that read or modify data.

CALLED ON NULL INPUT

Specifies that the procedure will be called if any, or even if all parameter values are null.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of query result sets that the procedure can return. The default is DYNAMIC RESULT SETS 0, which indicates that there are no result sets. The value must be between 0 and 32767.

ALLOW DEBUG MODE, DISALLOW DEBUG MODE, or DISABLE DEBUG MODE

Specifies whether this version of the routine can be run in debugging mode. The default is determined using the value of the CURRENT DEBUG MODE special register.

ALLOW DEBUG MODE

Specifies that this version of the routine can be run in debugging mode. When this version of the routine is invoked and debugging is attempted, a WLM environment must be available.

DISALLOW DEBUG MODE

Specifies that this version of the routine cannot be run in debugging mode.

You can use an ALTER statement to change this option to ALLOW DEBUG MODE for this initial version of the routine.

DISABLE DEBUG MODE

Specifies that this version of the routine can never be run in debugging mode.

This version of the routine cannot be changed to specify ALLOW DEBUG MODE or DISALLOW DEBUG MODE after this version of the routine has been created or altered to use DISABLE DEBUG MODE. To change this option, drop the routine and create it again using the desired option. An alternative to dropping and recreating the routine is to create a version of the routine that uses the desired option and make that version the active version.

When DISABLE DEBUG MODE is in effect, the WLM ENVIRONMENT FOR DEBUG MODE is ignored.

PARAMETER CCSID

Indicates whether the encoding scheme for character or graphic string parameters is ASCII, EBCDIC, or UNICODE. The default encoding scheme is the value that is specified in the CCSID clauses of the parameter list or in the field DEF ENCODING SCHEME on installation panel DSNTIPF.

This clause provides a convenient way to specify the encoding scheme for character or graphic string parameters. If individual CCSID clauses are specified for individual parameters in addition to this PARAMETER CCSID clause, the value that is specified in *all* of the CCSID clauses must be the same value that is specified in this clause.

This clause also specifies the encoding scheme that will be used for system-generated parameters of the routine.

QUALIFIER *schema-name*

Specifies the implicit qualifier that is used for unqualified names of tables,

views, indexes, and aliases that are referenced in the routine body. The default value is the same as the default schema.

PACKAGE OWNER *authorization-name*

Specifies the owner of the package that is associated with the first version of the routine. The SQL authorization ID of the process is the default value.

The authorization ID must have the privileges that are required to execute the SQL statements that are contained in the routine body. The value of PACKAGE OWNER is subject to translation when it is sent to a remote system.

If the privilege set lacks SYSADM or SYSCTRL authority, *authorization-name* must be the same as one of the authorization IDs of the process or the authorization ID of the process. If the privilege set includes SYSADM or SYSCTRL authority, *authorization-name* can be any authorization ID.

ASUTIME

Specifies the total amount of processor time, in CPU service units, that a single invocation of a routine can run. The value is unrelated to the ASUTIME column of the resource limit specification table.

When you are debugging a routine, setting a limit can be helpful in case the routine gets caught in a loop. For information on service units, see *z/OS MVS Initialization and Tuning Guide*.

NO LIMIT

Specifies that there is no limit on the service units.

NO LIMIT is the default.

LIMIT *integer*

Specifies that the limit on the service units is a positive *integer* in the range of 1 to 2 147 483 647. If the routine uses more service units than the specified value, DB2 cancels the routine.

COMMIT ON RETURN NO or COMMIT ON RETURN YES

Indicates whether DB2 commits the transaction immediately on return from the procedure.

COMMIT ON RETURN NO

DB2 does not issue a commit when the procedure returns. NO is the default.

COMMIT ON RETURN YES,

DB2 issues a commit when the procedure returns if the following statements are true:

- The SQLCODE that is returned by the CALL statement is not negative.
- The procedure is not in a must-abort state.

The commit operation includes the work that is performed by the calling application process and by the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

INHERIT SPECIAL REGISTERS or DEFAULT SPECIAL REGISTERS

Specifies how special registers are set on entry to the routine.

INHERIT SPECIAL REGISTERS

Specifies that the values of special registers are inherited, according to the rules that are listed in the table for characteristics of special registers in a routine in Table 37 on page 151.

INHERIT SPECIAL REGISTERS is the default.

DEFAULT SPECIAL REGISTERS

Specifies that special registers are initialized to the default values, as indicated by the rules in the table for characteristics of special registers in a routine in Table 37 on page 151.

WLM ENVIRONMENT FOR DEBUG MODE *name*

Specifies the WLM (workload manager) application environment that is used by DB2 when debugging the routine. The *name* of the WLM environment is an SQL identifier.

If you do not specify WLM ENVIRONMENT FOR DEBUG MODE, DB2 uses the default WLM-established stored procedure address space specified at installation time.

To define a routine that is to run in a specified WLM application environment, you must have the appropriate authority for the WLM application environment. For an example of a RACF command that provides this authorization, see Running stored procedures.

The WLM ENVIRONMENT FOR DEBUG MODE value is ignored when DISABLE DEBUG MODE is in effect.

DEFER PREPARE or NODEFER PREPARE

Specifies whether to defer preparation of dynamic SQL statements that refer to remote objects, or to prepare them immediately.

The default depends on the value in effect for the REOPT option. If REOPT NONE is in effect, the default is inherited from the plan at run time. Otherwise, the default is DEFER PREPARE.

DEFER PREPARE

Specifies that the preparation of dynamic SQL statements that refer to remote objects will be deferred.

Refer to the DEFER(PREPARE) option in *DB2 Command Reference* for considerations with distributed processing.

NODEFER PREPARE

Specifies that the preparation of dynamic SQL statements that refer to remote objects will not be deferred.

CURRENT DATA(YES) or CURRENT DATA(NO)

Specifies whether to require data currency for read-only and ambiguous cursors when the isolation level of cursor stability is in effect. CURRENT DATA also determines whether block fetch can be used for distributed, ambiguous cursors.

YES

Specifies that data currency is required for read-only and ambiguous cursors. DB2 acquired page or row locks to ensure data currency. Block fetch is ignored for distributed, ambiguous cursors.

NO

Specifies that data currency is not required for read-only and ambiguous cursors. Block fetch is allowed for distributed, ambiguous cursors. Use of **CURRENT DATA(NO)** is not recommended if the routine attempts to dynamically prepare and execute a DELETE WHERE CURRENT OF statement against an ambiguous cursor after that cursor is opened. You receive a negative SQLCODE if your routine attempts to use a DELETE WHERE CURRENT OF statement for any of the following cursors:

- A cursor that is using block fetch

- A cursor that is using query parallelism
- A cursor that is positioned on a row that is modified by this or another application process

NO is the default.

DEGREE

Specifies whether to attempt to run a query using parallel processing to maximize performance.

1 Specifies that parallel processing should not be used.

1 is the default.

ANY

Specifies that parallel processing can be used.

DYNAMICRULES

Specifies the values that apply, at run time, for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements

DYNAMICRULES also specifies whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements.

In addition to the value of the DYNAMICRULES clause, the runtime environment of a routine controls how dynamic SQL statements behave at run time. The combination of the DYNAMICRULES value and the runtime environment determines the value for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The following values can be specified:

RUN

Specifies that dynamic SQL statements are to be processed using run behavior.

RUN is the default.

BIND

Specifies that dynamic SQL statements are to be processed using bind behavior.

DEFINEBIND

Specifies that dynamic SQL statements are to be processed using either define behavior or bind behavior.

DEFINERUN

Specifies that dynamic SQL statements are to be processed using either define behavior or run behavior.

INVOKEBIND

Specifies that dynamic SQL statements are to be processed using either invoke behavior or bind behavior.

INVOKERUN

Specifies that dynamic SQL statements are to be processed using either invoke behavior or run behavior.

See “Authorization IDs and dynamic SQL” on page 64 for information on the effects of these options.

APPLICATION ENCODING SCHEME

Specifies the default encoding scheme for SQL variables in static SQL statements in the routine body. The value is used for defining an SQL variable in a compound statement if the CCSID clause is not specified as part of the data type, and the PARAMETER CCSID routine option is not specified.

ASCII

Specifies that the data is encoded using the ASCII CCSIDs of the server.

EBCDIC

Specifies that the data is encoded using the EBCDIC CCSIDs of the server.

UNICODE

Specifies that the data is encoded using the Unicode CCSIDs of the server.

See the ENCODING bind option in *DB2 Command Reference* for information about how the default for this option is determined.

WITH EXPLAIN or WITHOUT EXPLAIN

Specifies whether information will be provided about how SQL statements in the routine will execute.

WITHOUT EXPLAIN

Specifies that information will not be provided about how SQL statements in the routine will execute.

You can get EXPLAIN output for a statement that is embedded in a routine that is specified using WITHOUT EXPLAIN by embedding the SQL statement EXPLAIN in the routine body. Otherwise, the value of the EXPLAIN option applies to all explainable SQL statements in the routine body, and to the fullselect portion of any DECLARE CURSOR statements.

WITHOUT EXPLAIN is the default.

WITH EXPLAIN

Specifies that information will be provided about how SQL statements in the routine will execute. Information is inserted into the table *owner.PLAN_TABLE*. *owner* is the authorization ID of the owner of the routine. Alternatively, the authorization ID of the owner of the routine can have an alias as *owner.PLAN_TABLE* that points to the base table, *PLAN_TABLE*. *owner* must also have the appropriate SELECT and INSERT privileges on that table. WITH EXPLAIN does not obtain information for statements that access remote objects. *PLAN_TABLE* must have a base table and can have multiple aliases with the same table name, *PLAN_TABLE*, but have different schema qualifiers. It cannot be a view or a synonym and should exist before the version is added or replaced. In all inserts to *owner.PLAN_TABLE*, the value of QUERYNO is the statement number that is assigned by DB2.

The WITH EXPLAIN option also populates two optional tables if they exist: *DSN_STATEMNT_TABLE* and *DSN_FUNCTION_TABLE*. *DSN_STATEMNT_TABLE* contains an estimate of the processing cost for an SQL statement. See *DB2 Application Programming and SQL Guide* for more information. *DSN_FUNCTION_TABLE* contains information about function resolution. See *DB2 Application Programming and SQL Guide* for more information.

For a description of the tables that are populated by the WITH EXPLAIN option, see “EXPLAIN” on page 1283.

WITH IMMEDIATE WRITE or WITHOUT IMMEDIATE WRITE

Specifies whether immediate writes are to be done for updates that are made to group buffer pool dependent page sets or partitions. This option is only applicable for data sharing environments. The IMMEDIATEWRITE subsystem parameter has no effect of this option. *DB2 Command Reference* shows the implied hierarchy of the IMMEDIATEWRITE bind option (which is similar to this routine option) as it affects run time.

WITHOUT IMMEDIATE WRITE

Specifies that normal write activity is performed. Updated pages that are group buffer pool dependent are written at or before phase one of commit or at the end of abort for transactions that have been rolled back.

WITHOUT IMMEDIATE WRITE is the default.

WITH IMMEDIATE WRITE

Specifies that updated pages that are group buffer pool dependent are immediately written as soon as the buffer update completes. Updated pages are written immediately even if the buffer is updated during forward progress or during the rollback of a transaction. **WITH IMMEDIATE WRITE** might impact performance.

ISOLATION LEVEL RR, RS, CS, or UR

Specifies how far to isolate the routine from the effects of other running applications. For information about isolation levels, see *DB2 Performance Monitoring and Tuning Guide*.

RR Specifies repeatable read.

RS Specifies read stability.

CS Specifies cursor stability. **CS** is the default.

UR Specifies uncommitted read.

WITH KEEP DYNAMIC or WITHOUT KEEP DYNAMIC

Specifies whether DB2 keeps dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC

Specifies that DB2 does not keep dynamic SQL statements after commit points.

WITHOUT KEEP DYNAMIC is the default.

WITH KEEP DYNAMIC

Specifies that DB2 keeps dynamic SQL statements after commit points. If you specify WITH KEEP DYNAMIC, the application does not need to prepare an SQL statement after every commit point. DB2 keeps the dynamic SQL statement until one of the following occurs:

- The application process ends
- A rollback operations occurs
- The application executes an explicit PREPARE statement with the same statement identifier as the dynamic SQL statement

If you specify WITH KEEP DYNAMIC, and the prepared statement cache is active, the DB2 subsystem keeps a copy of the prepared statement in the cache. If the prepared statement cache is not active, the subsystem keeps

only the SQL statement string past a commit point. If the application executes an OPEN, EXECUTE, or DESCRIBE operation for that statement, the statement is implicitly prepared.

If you specify WITH KEEP DYNAMIC, DDF server threads that are used to execute procedures or packages that have this option in effect will remain active. Active DDF server threads are subject to idle thread timeouts, as described in *DB2 Installation Guide* for install panel DSNTIPR.

If you specify WITH KEEP DYNAMIC, you must not specify REOPT ALWAYS. WITH KEEP DYNAMIC and REOPT ALWAYS are mutually exclusive. However, you can specify WITH KEEP DYNAMIC and REOPT ONCE.

Use WITH KEEP DYNAMIC to improve performance if your DRDA client application uses a cursor that is defined as WITH HOLD. The DB2 subsystem automatically closes a held cursor when there are no more rows to retrieve, which eliminates an extra network message.

OPHTINT *string-constant*

Specifies whether query optimization hints are used for static SQL statements that are contained within the body of the routine.

string-constant is a character string of up to 128 bytes in length, which is used by the DB2 subsystem when searching the PLAN_TABLE for rows to use as input. The default value is an empty string, which indicates that the DB2 subsystem does not use optimization hints for static SQL statements.

Optimization hints are only used if optimization hints are enabled for your system. See *DB2 Installation Guide* for information about enabling optimization hints.

SQL PATH

Specifies the SQL path that DB2 uses to resolve unqualified user-defined distinct type, function, and procedure names in the procedure body. The default value is "SYSIBM," "SYSFUN," "SYSPROC," and *procedure-schema*, where *procedure-schema* is the schema qualifier for the procedure that is the target of the statement.

Schemas SYSIBM, SYSFUN, and SYSPROC do not need to be explicitly specified. If any of these schemas is not explicitly specified, it is implicitly assumed at the beginning the SQL path. See "SQL path" on page 56 for information about how to specify the path so that the intended procedure is selected when it is invoked with an unqualified name.

DB2 calculates the length by taking each *schema-name* specified and removing any trailing blanks from it, adding two delimiters around it, and adding one comma after each schema name except for the last one. The length of the resulting string cannot exceed the length of the CURRENT_SCHEMA special register. If you do not specify the SYSIBM, SYSFUN, and SYSPROC, schemas, they are not included in the length of the SQL path. If the total length of the SQL path exceeds the length of the CURRENT_PATH special register, DB2 returns an error for the CREATE statement.

See "CURRENT_SCHEMA" on page 147 for information about the length of the CURRENT_SCHEMA special register and "CURRENT_PATH" on page 142 for information about the length of the CURRENT_PATH special register.

schema-name

Specifies a schema. DB2 does not validate that the specified schema actually exists when the CREATE statement is processed.

| *schema-name-list*

| Specifies a comma separated list of schema names. The same schema name
| should not appear more than once in the list of schema names. The
| number of schema names that you can specify is limited by the maximum
| length of the resulting SQL path.

| **SESSION_USER or USER**

| Specifies the value of the SESSION_USER or USER special register, which
| represents a maximum 8 byte (in EBCDIC) *schema-name*. At the time the
| CREATE statement is processed, this length is included in the total length
| of the list of schema names that is specified for the PATH bind option.

| **DEFAULT**

| Specifies that the SQL PATH should be set to "SYSIBM", "SYSFUN",
| "SYSPROC", *procedure-schema*. *procedure-schema* is the schema qualifier for
| the procedure that is being created.

| If you specify DEFAULT, you do not need to explicitly specify the SYSIBM,
| SYSFUN, and SYSPROC schemas; these schemas are implicitly added to
| the beginning of the SQL path in the order listed.

| **RELEASE AT**

| Specifies when to release resources that the procedure uses: either at each
| commit point or when the procedure terminates.

| **COMMIT**

| Specifies that resources will be released at each commit point.

| COMMIT is the default.

| **DEALLOCATE**

| Specifies that resources will be released only when the procedure
| terminates. DEALLOCATE has no effect on packages that run on a DB2
| server through a DRDA connection with a client system. DEALLOCATE
| has no effect on dynamic SQL statements, which always use RELEASE AT
| COMMIT, with this exception: When you use the RELEASE AT
| DEALLOCATE clause and the WITH KEEP DYNAMIC clause, and the
| subsystem is installed with a value of YES for the field CACHE DYNAMIC
| SQL on installation panel DSNTIP8, the RELEASE AT DEALLOCATE
| option is honored for dynamic SELECT and data change statements.

| Locks that are acquired for dynamic statements are held until one of the
| following events occurs:

- | • The application process ends.
- | • The application process issues a PREPARE statement with the same
| statement identifier. (Locks are released at the next commit point).
- | • The statement is removed from the prepared statement cache because
| the statement has not been used. (Locks are released at the next commit
| point).
- | • An object that the statement is dependent on is dropped or altered, or a
| privilege that the statement needs is revoked. (Locks are released at the
| next commit point).

| RELEASE AT DEALLOCATE can increase the package or plan size because
| additional items become resident in the package or plan. For more
| information about how the RELEASE clause affects locking and
| concurrency, see *DB2 Performance Monitoring and Tuning Guide*.

REOPT

Specifies if DB2 will determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE

Specifies that DB2 does not determine the access path at run time by using the values of SQL variables or SQL parameters, parameter markers, and special registers.

NONE is the default.

ALWAYS

Specifies that DB2 always determines the access path at run time each time an SQL statement is run. Do not specify REOPT ALWAYS with the WITH KEEP DYNAMIC or NODEFER PREPARE clauses.

ONCE

Specifies that DB2 determine the access path for any dynamic SQL statements only once, at the first time the statement is opened. This access path is used until the prepared statement is invalidated or removed from the dynamic statement cache and need to be prepared again.

VALIDATE RUN or VALIDATE BIND

Specifies whether to recheck, at run time, errors of the type "OBJECT NOT FOUND" and "NOT AUTHORIZED" that are found during bind or rebind. The option has no effect if all objects and needed privileges exist.

VALIDATE RUN

Specifies that if needed objects or privileges do not exist when the CREATE statement is processed, warning messages are returned, but the CREATE statement succeeds. The DB2 subsystem rechecks for the objects and privileges at run time for those SQL statements that failed the checks during processing of the CREATE statement. The authorization checks the use of the authorization ID of the owner of the routine.

VALIDATE RUN is the default.

VALIDATE BIND

Specifies that if needed objects or privileges do not exist at the time the CREATE statement is processed, an error is issued and the CREATE statement fails.

ROUNDING

Specifies the desired rounding mode for manipulation of DECFLOAT data. The default value is taken from the DEFAULT DECIMAL FLOATING POINT ROUNDING MODE in DECP.

DEC_ROUND_CEILING

Specifies numbers are rounded towards positive infinity.

DEC_ROUND_DOWN

Specifies numbers are rounded towards 0 (truncation).

DEC_ROUND_FLOOR

Specifies numbers are rounded towards negative infinity.

DEC_ROUND_HALF_DOWN

Specifies numbers are rounded to nearest; if equidistant, round down.

DEC_ROUND_HALF_EVEN

Specifies numbers are rounded to nearest; if equidistant, round so that the final digit is even.

| **DEC_ROUND_HALF_UP**

| Specifies numbers are rounded to nearest; if equidistant, round up.

| **DEC_ROUND_UP**

| Specifies numbers are rounded away from 0.

| **DATE FORMAT ISO, EUR, USA, JIS, or LOCAL**

| Specifies the date format for result values that are string representations of
| date or time values. See “String representations of datetime values” on page 89
| for more information.

| The default format is specified in the DATE FORMAT field of installation panel
| DSNTIP4 of the system where the routine is defined. You cannot use the
| LOCAL option unless you have a date exit routine.

| **DECIMAL(15), DECIMAL(31), DECIMAL(15,s), or DECIMAL(31,s)**

| Specifies the maximum precision that is to be used for decimal arithmetic
| operations. See “Arithmetic with two decimal operands” on page 183 for more
| information. The default format is specified in the DECIMAL ARITHMETIC
| field of installation panel DSNTIPF of the system where the routine is defined.
| If the form *pp.s* is specified, *s* must be a number between 1 and 9. *s* represents
| the minimum scale that is to be used for division.

| **FOR UPDATE CLAUSE OPTIONAL or FOR UPDATE CLAUSE REQUIRED**

| Specifies whether the FOR UPDATE clause is required for a DECLARE
| CURSOR statement if the cursor is to be used to perform positioned updates.

| **FOR UPDATE CLAUSE REQUIRED**

| Specifies that a FOR UPDATE clause must be specified as part of the
| cursor definition if the cursor will be used to make positioned updates.

| **FOR UPDATE CLAUSE REQUIRED** is the default.

| **FOR UPDATE CLAUSE OPTIONAL**

| Specifies that the FOR UPDATE clause does not need to be specified in
| order for a cursor to be used for positioned updates. The routine body can
| include positioned UPDATE statements that update columns that the user
| is authorized to update.

| The FOR UPDATE clause with no column list applies to static or dynamic SQL
| statements. Even if you do not use this clause, you can specify FOR UPDATE
| OF with a column list to restrict updates to only the columns that are named
| in the FOR UPDATE clause and to specify the acquisition of update locks.

| **TIME FORMAT ISO, EUR, USA, JIS, or LOCAL**

| Specifies the time format for result values that are string representations of
| date or time values. See “String representations of datetime values” on page 89
| for more information.

| The default format is specified in the TIME FORMAT field of installation panel
| DSNTIP4 of the system where the routine is defined. You cannot use the
| LOCAL option unless you have a date exit routine.

| **SQL-routine-body**

| Specifies the statements that define the body of the SQL procedure. For
| information on the SQL control statements that are supported in native SQL
| procedures, see Chapter 6, “SQL control statements for native SQL procedures,”
| on page 1541. If an *SQL-procedure-statement* is the only statement in the
| procedure body, the statement must not end with a semicolon. For information
| on the SQL statements that are allowed in native SQL procedures, see “SQL
| statements allowed in SQL procedures” on page 1608.

Notes

See “Notes” on page 1032 for information about:

- Owner privileges
- Choosing data types for parameters
- Specifying the encoding scheme for parameters
- Environments for running stored procedures
- Accessing result sets from nested stored procedures

Versions of a procedure: The CREATE PROCEDURE statement for an SQL procedure defines the initial version of the procedure. You can define additional versions using the ADD VERSION clause of the ALTER PROCEDURE statement. All versions of a procedure share the same procedure signature and the same specific name. However, the parameters names can differ between versions of a procedure. Only one version of the procedure can be considered to be the active version of the procedure.

Characteristics of the package that is generated for a procedure: The package that is associated with the first version of a procedure is named as follows:

- *location* is set to the value of the CURRENT SERVER special register
- *collection-id* (schema) for the package is the same as the schema qualifier of the procedure.
- *package-id* is the same as the specific name of the procedure
- *version-id* is the same as the version identifier for the initial version of the procedure.

If you want to change the *collection-id* for the name of the package, you need to make a copy of the package.

The package is generated using the bind options that correspond to the implicitly or explicitly specified procedure options. See “Implicit rebinding and regeneration that occurs because of an ALTER PROCEDURE statement” in Chapter 6, “SQL control statements for native SQL procedures,” on page 1541 for more information. In addition to the corresponding bind options, the package is generated using the following bind options:

- DBPROTOCOL(DRDA)
- FLAG(1)
- SQLERROR(NOPACKAGE)
- ENABLE(*)

See Table 83 on page 775 for additional information about the correspondence of procedure options to bind options.

Identifier resolution: See Chapter 6, “SQL control statements for native SQL procedures,” on page 1541 for information on how names are resolved to columns, SQL variables, or SQL parameters within an SQL routine. Name resolution is unchanged for external SQL procedures.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the procedure name for parameters, and the label name for SQL variables.

Lines within the SQL procedure definition: When a procedure is created, information is retained on lines in the CREATE statement. Lines are determined by the presence of the *new line control character*.

Error handling in SQL procedures: You should consider the possible exceptions that can occur for each SQL statement in the body of a procedure. Any exception SQLSTATE that is not handled within the procedure using a handler within a compound statement or a compound statement that is nested within that compound statement results in the exception SQLSTATE being returned to the caller of the procedure.

Compatibilities: For compatibility with previous versions of DB2, the following clauses can be specified, but they will be ignored and a warning will be issued:

- STAY RESIDENT
- PROGRAM TYPE
- RUN OPTIONS
- NO DBINFO
- COLLID or NOCOLLID
- SECURITY
- PARAMETER STYLE GENERAL WITH NULLS
- STOP AFTER SYSTEM DEFAULT FAILURES
- STOP AFTER *nn* FAILURES
- CONTINUE AFTER FAILURES

If the FENCED or EXTERNAL clause is specified, an external SQL procedure will be generated. See “CREATE PROCEDURE (SQL - external)” on page 1035 for more information.

If WLM ENVIRONMENT is specified without the FOR DEBUG MODE keywords, and error is returned. If WLM ENVIRONMENT is specified for a native SQL procedure, WLM ENVIRONMENT FOR DEBUG MODE must be specified.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RESULT SET and RESULT SETS as synonyms for DYNAMIC RESULT SETS
- VARIANT as a synonym for NOT DETERMINISTIC
- NOT VARIANT as a synonym for DETERMINISTIC
- NULL CALL as a synonym for CALLED ON NULL

Examples

Example 1: Create the definition for an SQL procedure. The procedure accepts an employee number and a multiplier for a pay raise as input. The following tasks are performed in the procedure body:

- Calculate the employee's new salary.
- Update the employee table with the new salary value.

```
CREATE PROCEDURE UPDATE_SALARY_1
  (IN EMPLOYEE_NUMBER CHAR(10),
   IN RATE DECIMAL(6,2))
LANGUAGE SQL
MODIFIES SQL DATA
UPDATE EMP
SET SALARY = SALARY * RATE
WHERE EMPNO = EMPLOYEE_NUMBER
```

| *Example 2:* Create the definition for the SQL procedure described in example 1, but
| specify that the procedure has these characteristics:

- | • The same input always produces the same output.
- | • SQL work is committed on return to the caller.

```
| CREATE PROCEDURE UPDATE_SALARY_1  
|   (IN EMPLOYEE_NUMBER CHAR(10),  
|   IN RATE DECIMAL(6,2))  
|   LANGUAGE SQL  
|   MODIFIES SQL DATA  
|   DETERMINISTIC  
|   COMMIT ON RETURN YES  
|   UPDATE EMP  
|     SET SALARY = SALARY * RATE  
|     WHERE EMPNO = EMPLOYEE_NUMBER
```

| For more examples of SQL procedures, see Chapter 6, “SQL control statements for
| native SQL procedures,” on page 1541.

CREATE ROLE

The CREATE ROLE statement creates a role at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following authorities:

- SYSADM authority
- SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the set of privileges that are held by the owner of the plan or package.

If the statement is dynamically prepared, the privilege set is the set of privileges that are held by the SQL authorization ID of the process or by the role that is associated with the primary authorization ID, if the statement is run in a trusted context and the ROLE AS OBJECT OWNER clause is specified.

Syntax

▶▶—CREATE ROLE—*role-name*—▶▶

Description

role-name

Names the role. The name must not identify a role that exists at the current server. The name must not be 'PUBLIC'.

Examples

The following statement creates a role named TELLER.

```
CREATE ROLE TELLER;
```

CREATE SEQUENCE

The CREATE SEQUENCE statement creates a sequence at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

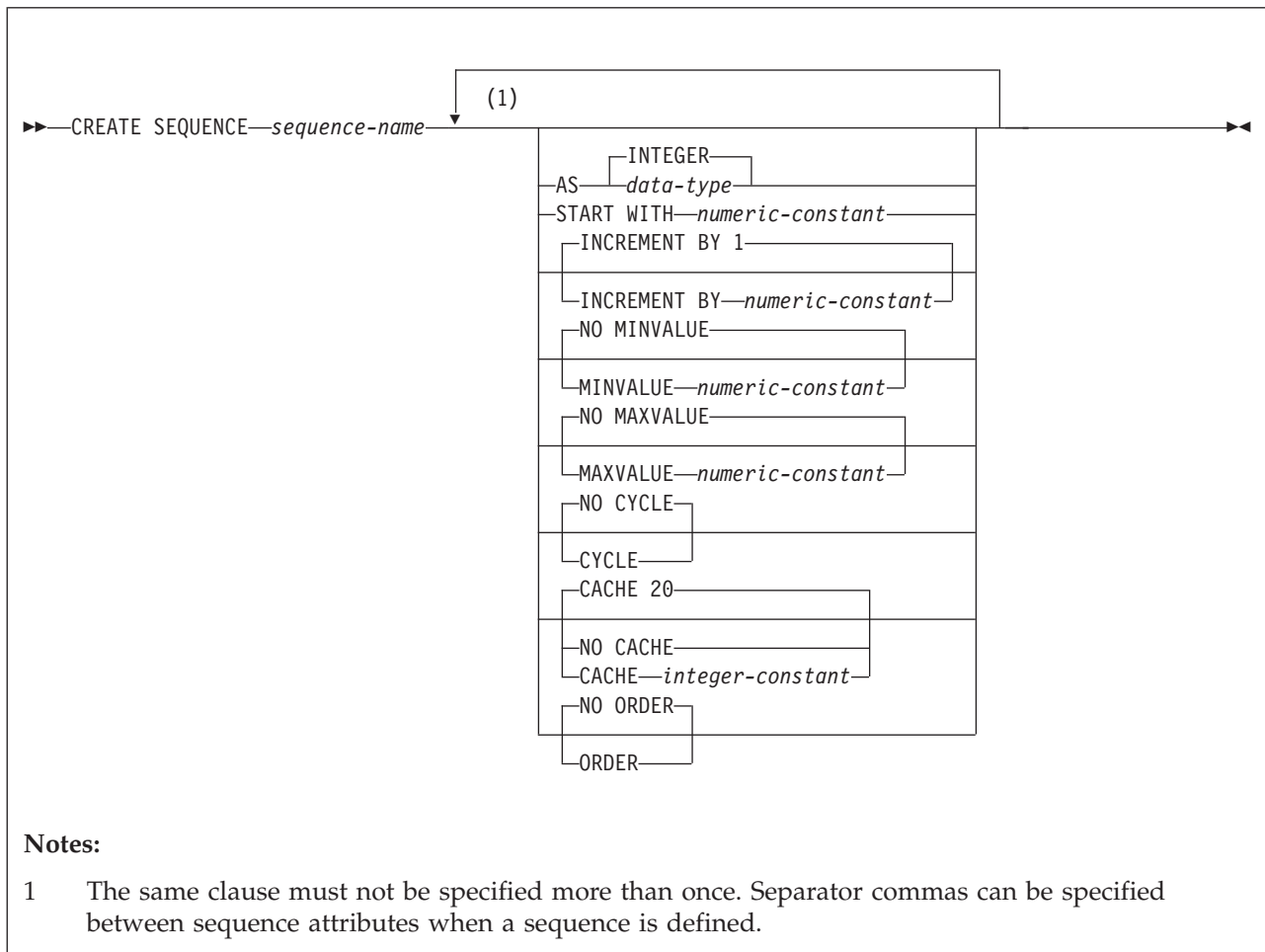
The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

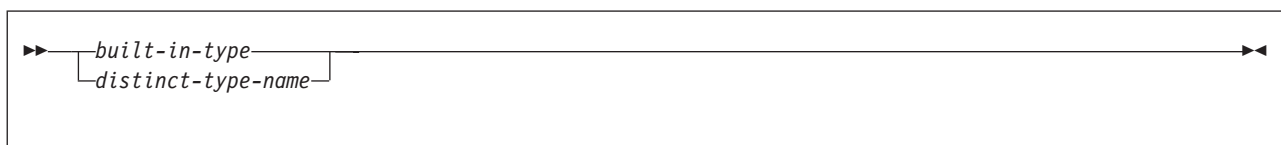
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

If the data type of the sequence is a distinct type, the privilege set must include the USAGE privilege on the distinct type.

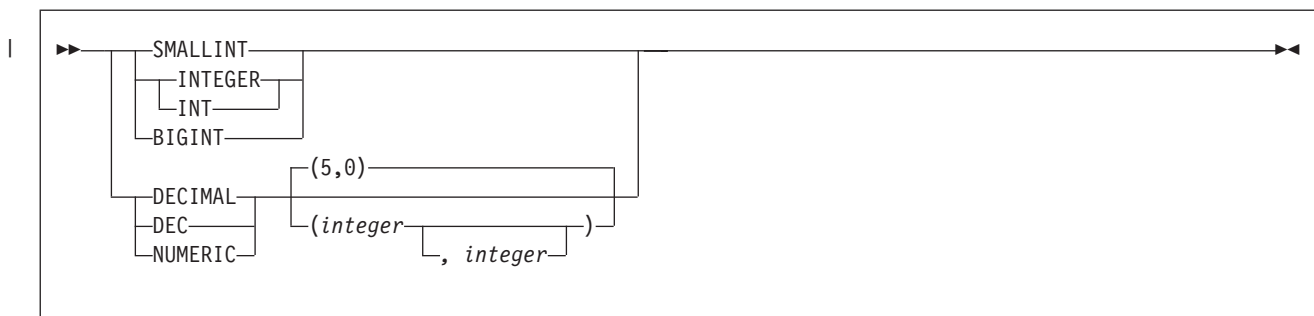
Syntax



data-type:



built-in-type:



Description

sequence-name

Names the sequence. The name, including the implicit or explicit qualifiers, must not identify an existing sequence at the current server, including the sequence names that are generated by DB2.

The schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

AS *data-type*

Specifies the data type to be used for the sequence value. The data type can be any exact numeric data type (SMALLINT, INTEGER, BIGINT, or DECIMAL with a scale of zero), or a user-defined distinct type for which the source type is an exact numeric data type with a scale of zero. The default, when AS is not specified, is INTEGER. If DECIMAL is specified, the default is DECIMAL(5,0).

START WITH *numeric-constant*

Specifies the first value for the sequence. The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point.

If the START WITH clause is not explicitly specified with a value, the default is the MINVALUE for ascending sequences and MAXVALUE for descending sequences.

This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the sequence. The value can be any positive or negative value (including 0) that could be assigned to a column of the data type that is associated with the sequence without any non-zero digits existing to the right of the decimal point. The default is 1.

If INCREMENT BY is positive, the sequence ascends. If INCREMENT BY is negative, the sequence descends. If INCREMENT is 0, the sequence is treated as an ascending sequence.

The absolute value of INCREMENT BY can be greater than the difference between MAXVALUE and MINVALUE.

MINVALUE or NO MINVALUE

Specifies the minimum value at which a descending sequence either cycles or stops generating values or an ascending sequence cycles to after reaching the maximum value. The default is NO MINVALUE.

MINVALUE *numeric-constant*

Specifies the minimum end of the range of values for the sequence. The last value that is generated for a cycle of a descending sequence will be equal to or greater than this value. MINVALUE is the value to which an ascending sequence cycles to after reaching the maximum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

For the effects of defining MINVALUE and MAXVALUE with the same value, see Defining a constant sequence.

NO MINVALUE

Specifies that the minimum end point of the range of values for the sequence has not been specified explicitly. In such a case, the default value for MINVALUE becomes one of the following:

- For an ascending sequence, the value is the START WITH value or 1 if START WITH is not specified.
- For a descending sequence, the value is the minimum value of the data type that is associated with the sequence.

MAXVALUE or NO MAXVALUE

Specifies the maximum value at which an ascending sequence either cycles or stops generating values or an descending sequence cycles to after reaching the minimum value. The default is NO MAXVALUE.

MAXVALUE *numeric-constant*

Specifies the maximum end of the range of values for the sequence. The last value that is generated for a cycle of an ascending sequence will be less than or equal to this value. MAXVALUE is the value to which a descending sequence cycles to after reaching the minimum value.

The value can be any positive or negative value that could be assigned to the a column of the data type that is associated with the sequence without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

If DECIMAL is specified in the AS clause, *numeric-constant* has a maximum value of 99 999.

For the effects of defining MAXVALUE and MINVALUE with the same value, see Defining a constant sequence.

NO MAXVALUE

Specifies the maximum end point of the range of values for the sequence has not been specified explicitly. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending sequence, the value is the maximum value of the data type that is associated with the sequence.
- For a descending sequence, the value is the START WITH value or -1 if START WITH is not specified.

CYCLE or NO CYCLE

Specifies whether or not the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition or by overshooting it. The default is NO CYCLE.

CYCLE

Specifies that the sequence continue to generate values after either the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum value, it generates its maximum value. The maximum and minimum values for the sequence defined by the MINVALUE and MAXVALUE options determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by the sequence. When a sequence is defined with CYCLE, any application

conversion tools for converting applications from other vendor platforms to DB2 should also explicitly specify MINVALUE, MAXVALUE, and START WITH values.

NO CYCLE

Specifies that the sequence cannot generate more values once the maximum or minimum value for the sequence has been reached. The NO CYCLE option (the default) can be altered to CYCLE at any time during the life of the sequence.

When the next value is being generated for a sequence if the maximum value (for an ascending sequence) or the minimum value (for a descending sequence) of the logical range of the sequence is exceeded and the NO CYCLE option is in effect, an error occurs.

CACHE or NO CACHE

Specifies whether or not to keep some preallocated values in memory for faster access. This is a performance and tuning option.

CACHE *integer-constant*

Specifies the maximum number of values of the sequence that DB2 can preallocate and keep in memory. Preallocating values in the cache reduces synchronous I/O when values are generated for the sequence. The actual number of values that DB2 caches is always the lesser of the number in effect for the CACHE option and the number of remaining values within the logical range. Thus, the CACHE value is essentially an upper limit for the size of the cache.

In the event the system is shut down (either normally or through a system failure), all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of sequence values that could be lost when the system is shut down.

The minimum value is 2. The default is CACHE 20.

In a data sharing environment, you can use the CACHE and NO ORDER options to allow multiple DB2 members to cache sequence values simultaneously.

NO CACHE

Specifies that values of the sequence are not to be preallocated. This option ensures that there is not a loss of values in the case of a system failure.

When NO CACHE is specified, the values of the sequence are not stored in the cache. In this case, every request for a new value for the sequence results in synchronous I/O.

ORDER or NO ORDER

Specifies whether the sequence numbers must be generated in order of request. The default is NO ORDER.

ORDER

Specifies that the sequence numbers are generated in order of request. Specifying ORDER might disable the caching of values. There is no guarantee that values are assigned in order across the entire server unless NO CACHE is also specified. ORDER applies only to a single-application process.

NO ORDER

Specifies that the sequence numbers do not need to be generated in order of request.

In a data sharing environment, if the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously, and the requests for next value assignments from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the same sequence, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested for next value first, then DB2B requested, and then DB2A again requested. Therefore, to guarantee that sequence numbers are generated in strict numeric order among multiple DB2 members using the same sequence concurrently, specify the ORDER option.

Notes

Owner privileges: The owner is authorized to change (ALTER privilege) or use (USAGE privilege) the sequence and grant others these privileges. See “GRANT (sequence privileges)” on page 1351. For more information about ownership of the object see “Authorization, privileges, and object ownership” on page 60.

Relationship of MINVALUE and MAXVALUE: MINVALUE must not be greater than MAXVALUE. Although MINVALUE is typically less than MAXVALUE, MINVALUE can equal MAXVALUE. If START WITH were the same value as MINVALUE and MAXVALUE, the sequence would be constant. The request for the next value in a constant sequence appears to have no effect because all of the values that are generated by the sequence are in fact the same value.

Defining sequences that cycle: When you define a sequence, you can choose to have it cycle automatically or not when the maximum or minimum value for the sequence has been reached.

- Implicitly or explicitly defining a sequence with NO CYCLE causes the sequence to not cycle automatically after the boundary is reached. However, you can use the ALTER SEQUENCE statement to cycle the sequence manually. ALTER SEQUENCE allows you to restart or extend the sequence, which causes sequence values to continue to be generated.
- Explicitly defining a sequence with CYCLE causes the sequence to cycle automatically after the boundary is reached. Sequence values continue to be generated after the sequence cycles.

When a sequence is defined to cycle automatically, the maximum or minimum value that is generated for a sequence might not be the actual MAXVALUE or MINVALUE value that is specified if the increment is a value other than 1 or -1. For example, the sequence defined with START WITH=1, INCREMENT=2, MAXVALUE=10 will generate a maximum value of 9, and will not generate the value 10.

When a sequence is defined with CYCLE, any application conversion tools (for converting applications from other vendor platforms to DB2) should also explicitly specify MINVALUE, MAXVALUE, and START WITH.

Defining a constant sequence: You can define a sequence such that it always returns the same (or a constant) value. To create a constant sequence, use either of these techniques when defining the sequence:

- Specify an INCREMENT value of zero and a START WITH value that does not exceed MAXVALUE.
- Specify the same value for START WITH, MINVALUE, and MAXVALUE, and specify CYCLE.

A constant sequence can be used as a numeric global variable. You can use ALTER SEQUENCE to adjust the values that are generated for a constant sequence.

Consumed values of a sequence: After DB2 generates a value for a sequence, that value can be said to be "consumed" regardless of whether or not that value is utilized by the application or not. The value is not reused within the current cycle. A consumed value might not be utilized when the statement that caused the value to be generated fails for some reason or is rolled back after the value was generated. Generated but unused values can constitute gaps in a sequence.

Gaps in a sequence: Consecutive values in a sequence differ by the constant INCREMENT BY value specified for the sequence. However, gaps can occur in the values that are assigned to a sequence object by DB2.

The following situations are some examples of how gaps can be introduced in the sequence values:

- A transaction has advanced the sequence and then rolls back.
- The SQL statement leading to the generation of the next value fails after the value was generated.
- The NEXT VALUE expression is used in the SELECT statement of a cursor in a DRDA environment where the client uses block-fetch and not all retrieved rows are fetched by the application.
- The sequence is altered and then the alteration is rolled back.
- The sequence (or an identity column table) is dropped and then the drop is rolled back.
- The SYSIBM.SYSSEQ table space is stopped or closed for any reason (including when DSMAX is reached)
- The DB2 subsystem is stopped or goes down

Values of such gaps are not available for the current cycle, unless the sequence is altered and restarted in a specific way to make them available.

A sequence is incremented independently of a transaction. Thus, a given transaction increments the sequence two times might see a gap in the two numbers that it receives if other transactions concurrently increment the same sequence. Most applications can tolerate these instances as these are not really gaps.

Duplicate sequence values: It is possible the duplicate values can be generated for a sequence. Duplicate values are most likely to occur when a sequence is defined with the CYCLE option, is defined as a constant sequence, or is altered. For example, the following situations could cause duplicate sequence values:

- A sequence is defined with the attributes START WITH=2, INCREMENT BY 2, MINVALUE=2, MAXVALUE=10, and CYCLE.
- The ALTER SEQUENCE statement is used to restart the sequence with a value that has already been generated.
- The ALTER SEQUENCE statement is used to reverse the ascending direction of a sequence by changing the INCREMENT BY value from a positive to a negative.

Using sequences: A sequence can be referenced using a *sequence-reference*. A sequence reference can appear in most places that an expression can appear. A sequence reference can specify whether the value to be returned is a newly generated value or the previously generated value. A NEXT VALUE sequence expression is used to generate a new value. A PREVIOUS VALUE sequence

expression is used to obtain the last assigned value of a sequence. For more information, see "Sequence reference" on page 217.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- NOMINVALUE (single key word) as a synonym for NO MINVALUE
- NOMAXVALUE (single key word) as a synonym for NO MAXVALUE
- NOCYCLE (single key word) as a synonym for NO CYCLE
- NOCACHE (single key word) as a synonym for NO CACHE
- NOORDER (single key word) as a synonym for NO ORDER

Examples

Example 1: Create a sequence names "org_seq" that starts at 1 increments by 1, does not cycle, and caches 24 values at a time:

```
CREATE SEQUENCE ORDER_SEQ
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24;
```

INCREMENT 1, NO MAXVALUE, and NO CYCLE are defaults and do not need to be specified.

Example 2: The following example shows how to create and use a sequence named "order_seq" in a table named "orders":

```
CREATE SEQUENCE ORDER_SEQ
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 20;
INSERT INTO ORDERS (ORDERNO, CUSTNO)
  VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
```

or to update the orders:

```
UPDATE ORDERS
  SET ORDERNO = NEXT VALUE FOR ORDER_SEQ
  WHERE CUSTNO = 123456;
```

Example 3: The following example shows how to use the same sequence number as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row to generate the sequence value and with a PREVIOUS VALUE expression for the other rows to refer to the sequence value most recently generated.

```
INSERT INTO ORDERS (ORDERNO, CUSTNO)
  VALUES (NEXT VALUE FOR ORDER_SEQ, 123456);
INSERT INTO LINE_ITEMS (ORDERNO, PARTNO, QUANTITY)
  VALUES (PREVIOUS VALUE FOR ORDER_SEQ, 987654, 100);
```

If NEXT VALUE is invoked in the same statement as the PREVIOUS VALUE, then regardless of their order in the statement, PREVIOUS VALUE returns the previous (unincremented) value and NEXT VALUE returns the next value.

CREATE STOGROUP

The CREATE STOGROUP statement creates a storage group at the current server. Storage from the identified volumes can later be allocated for table spaces and index spaces.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

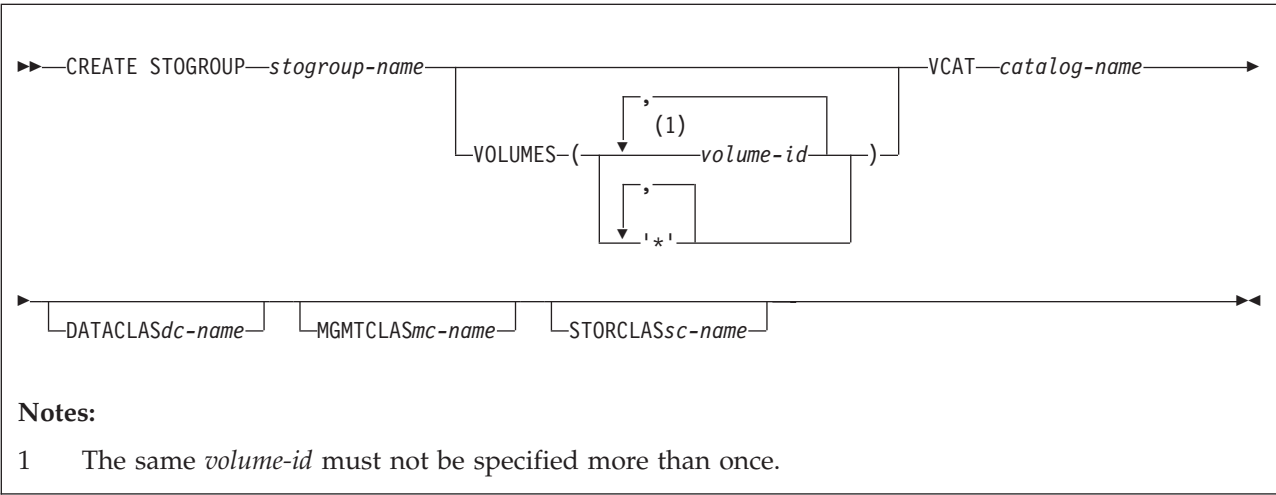
The privilege set that is defined below must include at least one of the following:

- The CREATESG privilege
- SYSADM or SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the of the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

Syntax



Description

stogroup-name

Names the storage group. The name must not identify a storage group that exists at the current server.

VOLUMES(*volume-id*,...) or **VOLUMES**('*',...)

Defines the volumes of the storage group. Each *volume-id* is a volume serial number of a storage volume. The volume serial number can have a maximum of six characters and is specified as an identifier or a string constant.

If the data set that is associated with the storage group is not managed by Storage Management Subsystem (SMS), **VOLUMES** must be specified. Asterisks are recognized only by SMS. SMS usage is recommended, rather than using DB2 to allocate data to specific volumes. Having DB2 select the volume requires non-SMS usage or assigning an SMS Storage Class with guaranteed space. However, because guaranteed space reduces the benefits of SMS allocation, it is not recommended. If one or more of the **DATACLAS**, **MGMTCLAS**, or **STORCLAS** clauses are specified, **VOLUMES** can be omitted. If the **VOLUMES** clause is omitted, the volume selection is controlled by SMS.

If you do choose to use specific volume assignments, additional manual space management must be performed. Free space must be managed for each individual volume to prevent failures during the initial allocation and extension. This process generally requires more time for space management and results in more space shortages. Guaranteed space should be used only where the space needs are relatively small and do not change.

VCAT *catalog-name*

Identifies the integrated catalog facility catalog for the storage group. You must specify an alias³⁰ if the name of the integrated catalog facility catalog is longer than 8 characters.

The designated catalog is the one in which entries are placed for the data sets created by DB2 with the aid of the storage group. These are linear VSAM data sets for associated table or index spaces or for their partitions. For each such space or partition, association is made through a **USING** clause in a **CREATE TABLESPACE**, **CREATE INDEX**, **ALTER TABLESPACE**, or **ALTER INDEX** statement. For more on the association, see the descriptions of those statements in this chapter.

Conventions for data set names are given in *DB2 Administration Guide*. *catalog-name* is the first qualifier for each data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DATACLAS *dc-name*

Identifies the name of the SMS data class to associate with the DB2 storage group. The SMS data class name must be from 1-8 characters in length. The SMS storage administrator defines the data class that can be used. **DATACLAS** must not be specified more than one time.

MGMTCLAS *mc-name*

Identifies the name of the SMS management class to associate with the DB2 storage group. The SMS management class name must be from 1-8 characters in length. The SMS storage administrator defines the management class that can be used. **MGMTCLAS** must not be specified more than one time.

STORCLAS *sc-name*

Identifies the name of the SMS storage class to associate with the DB2 storage group. The SMS storage class name must be from 1-8 characters in length. The

30. The alias of an integrated catalog facility catalog.

SMS storage administrator defines the storage class that can be used.
STORCLAS must not be specified more than one time.

Notes

Device types: When the storage group is used at run time, an error can occur if the volumes in the storage group are of different device types, or if a volume is not available to z/OS for dynamic allocation of data sets.

When a storage group is used to extend a data set, all volumes in the storage group must be of the same device type as the volumes used when the data set was defined. Otherwise, an extend failure occurs if an attempt is made to extend the data set.

Number of volumes: There is no specific limit on the number of volumes that can be defined for a storage group. However, the maximum number of volumes that can be managed for a storage group is 133.

z/OS imposes a limit on the number of volumes that can be allocated per data set (currently, 59 volumes). For the latest information on that restriction, see *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

Storage group owner: If the statement is embedded in an application program, the owner of the plan or package is the owner of the storage group. If the statement is dynamically prepared, the SQL authorization ID of the process is the owner of the storage group. The owner has the privilege of altering and dropping the storage group.

Specifying volume IDs: A new storage group must have either specific volume IDs or non-specific volume IDs. You cannot create a storage group that contains a mixture of specific and non-specific volume IDs.

Verifying the existence of volumes and classes: When processing the VOLUMES, DATACLAS, MGMTCLAS, or STORCLAS clauses, DB2 does not check the existence of the volumes or classes or determine the types of devices that are identified or if SMS is active. Later, when the storage group allocates data sets, the list of volumes is passed in the specified order to Data Facilities (DFSMSdfp). See *DB2 Administration Guide* for more information about creating DB2 storage groups.

Example

Create storage group, DSN8G910, of volumes ABC005 and DEF008. DSNCAT is the integrated catalog facility catalog name.

```
CREATE STOGROUP DSN8G910
  VOLUMES (ABC005,DEF008)
  VCAT DSNCAT;
```

CREATE SYNONYM

The CREATE SYNONYM statement defines a synonym for a table or view at the current server.

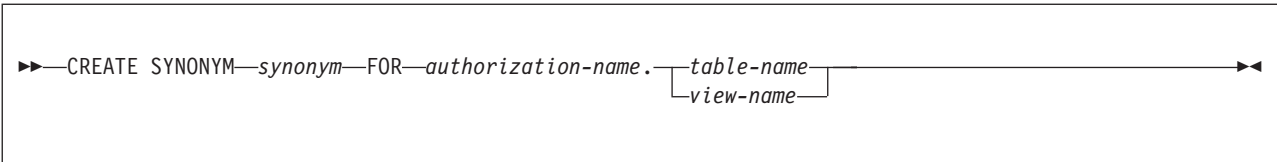
Invocation

| This statement can be embedded in an application program or issued interactively.
| It is an executable statement that can be dynamically prepared only if
| DYNAMICRULES run behavior is implicitly or explicitly specified. The statement
| cannot be processed in a trusted context that is defined with a role as the object
| owner.

Authorization

None required.

Syntax



Description

synonym

| Names the synonym. The name must not identify a synonym, table, view, or
| alias that is owned by the owner of the synonym that is being created.

FOR *authorization-name.table-name* **or** *authorization-name.view-name*

Identifies the object to which the synonym applies. The name must consist of two parts and must identify a table, view, or alias that exists at the current server. If a table is identified, it must not be an auxiliary table or a declared temporary table. If an alias is identified, it must be an alias for a table or view at the current server and the synonym is defined for that table or view. The name must not identify a table that was implicitly created for an XML column.

Notes

| **Owner privileges:** There are no specific privileges on a synonym. For more
| information about ownership of an object, see “Authorization, privileges, and
| object ownership” on page 60.

In cases where the statement is dynamically prepared, users with SYSADM authority can create synonyms for other users. This is done by changing the value of the CURRENT SQLID special register before issuing the CREATE SYNONYM statement. See “SET CURRENT SQLID” on page 1500 for details on changing the value of the CURRENT SQLID special register.

Example

Define DEPT as a synonym for the table DSN8910.DEPT.

```
CREATE SYNONYM DEPT
  FOR DSN8910.DEPT;
```

This example does not work if the current SQL authorization ID is DSN8910.

CREATE TABLE

The CREATE TABLE statement defines a table. The definition must include its name and the names and attributes of its columns. The definition can include other attributes of the table, such as its primary key and its table space.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATETAB privilege for the database explicitly specified by the IN clause. If the IN clause is not specified, the CREATETAB privilege on database DSNDDB04 is required.
- DBADM, DBCTRL, or DBMAINT authority for the database explicitly specified by the IN clause. If the IN clause is not specified, DBADM, DBCTRL, or DBMAINT authority for database DSNDDB04 is required.
- SYSADM or SYSCTRL authority

If the table space is created implicitly, the privilege set that is defined below must include at least one of the following:

- The CREATETS privilege for the database explicitly specified by the IN clause. If the IN clause is not specified, the CREATETS privilege on database DSNDDB04 is required.
- DBADM, DBCTRL, or DBMAINT authority for the database explicitly specified by the IN clause. If the IN clause is not specified, DBADM, DBCTRL, or DBMAINT authority for database DSNDDB04 is required.
- SYSADM or SYSCTRL authority

The privilege set must also have the USE privilege for the default buffer pool and default storage group of the database if the database is specified in the IN clause.

For tables that are created in an implicit database, the database authority must be held on DSNDDB04.

Additional privileges might be required in the following conditions:

- The clause IN, LIKE or FOREIGN KEY is specified.
- The data type of a column is a distinct type.
- The table space is implicitly created.
- A *fullselect* is specified.
- A column is defined as a security label column.

See the description of the appropriate clauses for details about these privileges.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified:

- A role is the owner of the table that is being created
- The privilege set is the set of privileges that are held by that role
- The schema qualifier (implicit or explicit) must be the same as the role, unless the role has the CREATEIN privilege on the schema, or SYSADM or SYSCTRL authority

Otherwise, an authorization ID is the owner of the plan or package, and the following rules apply:

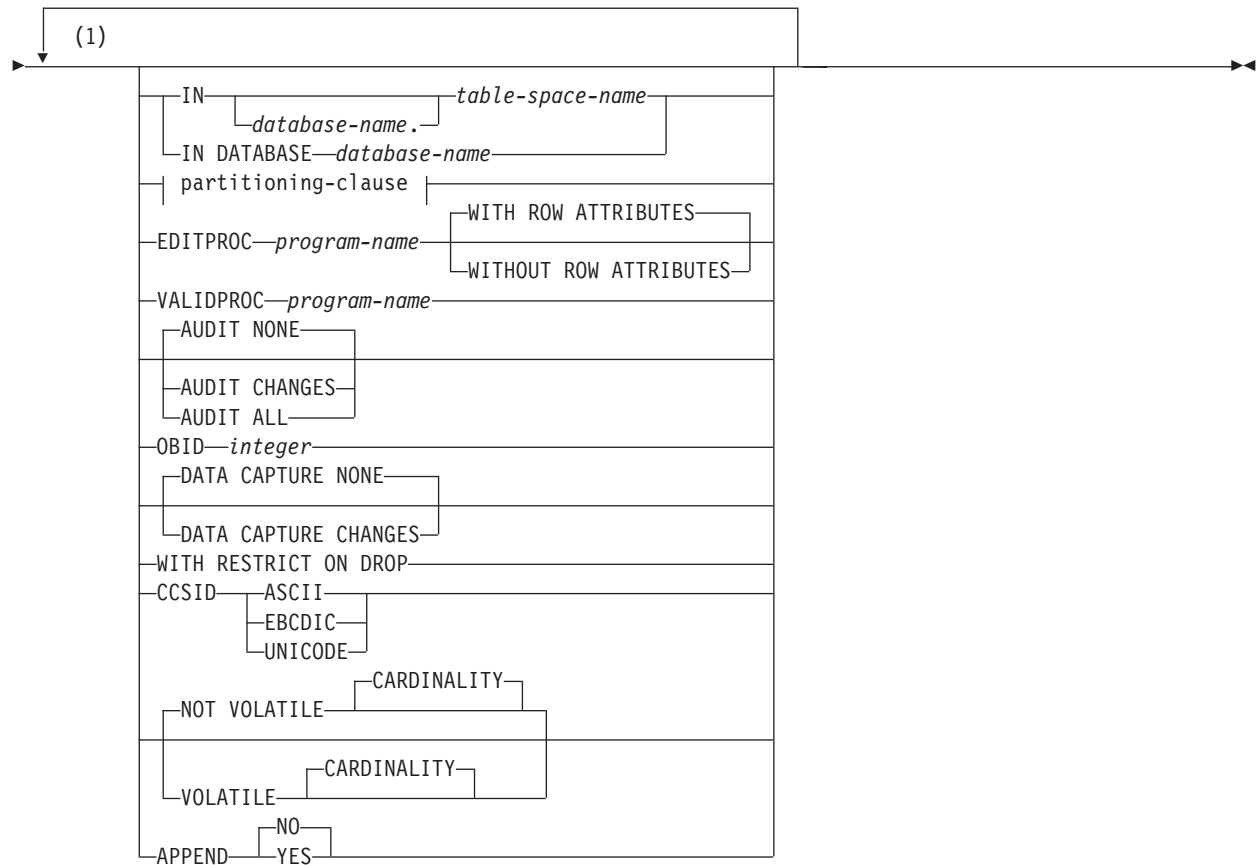
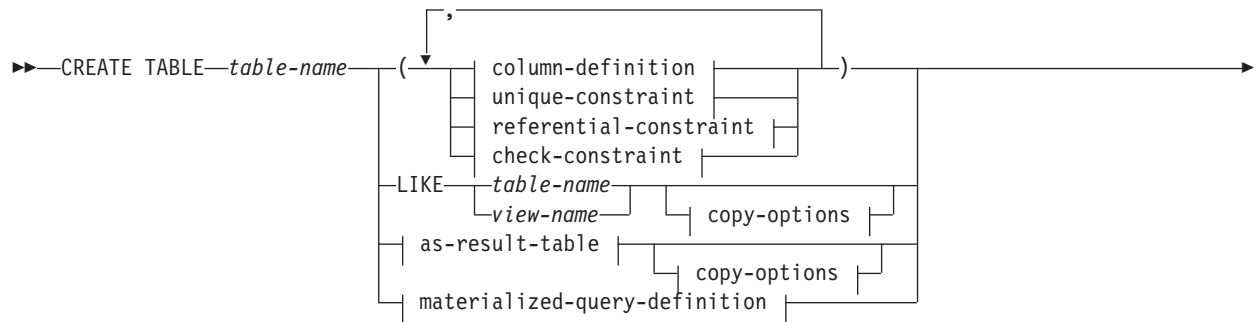
- If the privilege set lacks the CREATEIN privilege on the schema, or SYSADM or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as the authorization ID of the owner of the plan or package, and the owner of the table is that authorization ID.
- If the privilege set includes SYSADM or SYSCTRL authority, the schema qualifier (implicit or explicit) can be any schema name.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is in effect. When ROLE AS OBJECT OWNER is in effect, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process, and the owner of the table is that role. The schema qualifier (implicit or explicit) must be the same as that role, unless the role has CREATEIN privilege on the schema, or SYSADM or SYSCTRL authority.

For the case where the SQL authorization ID of the process holds the privileges, the owner of the table is the schema if the schema is explicitly specified, otherwise the owner of the table is the SQL authorization ID of the process. Additionally:

- If the privilege set lacks CREATEIN privilege on the schema, or SYSADM or SYSCTRL authority, the schema qualifier must be the same as one of the authorization IDs of the process and the privilege set that is held by that authorization ID must include all privileges that are needed to create the table.
- If the privilege set includes SYSADM or SYSCTRL authority, the schema qualifier can be any schema name. The privilege set that is held by the SQL authorization ID of the process must include all privileges that are needed to create the table.

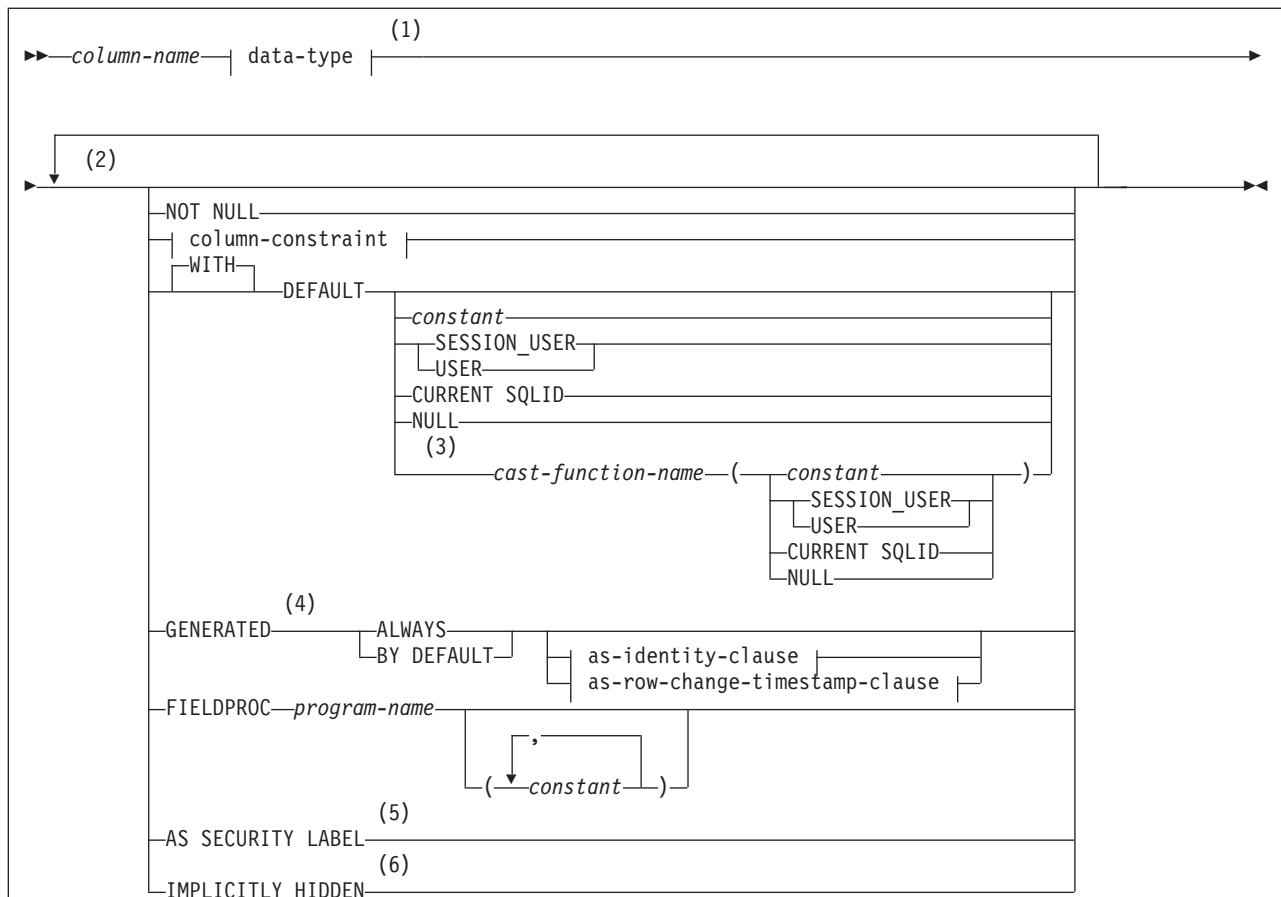
Syntax



Notes:

- 1 The same clause must not be specified more than once.

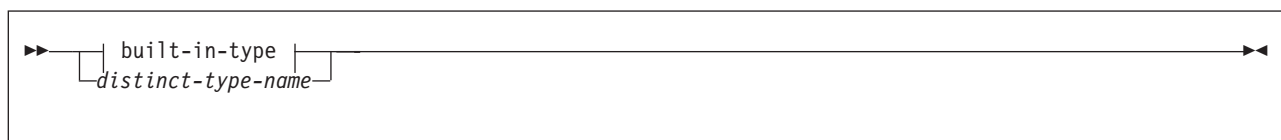
column-definition:



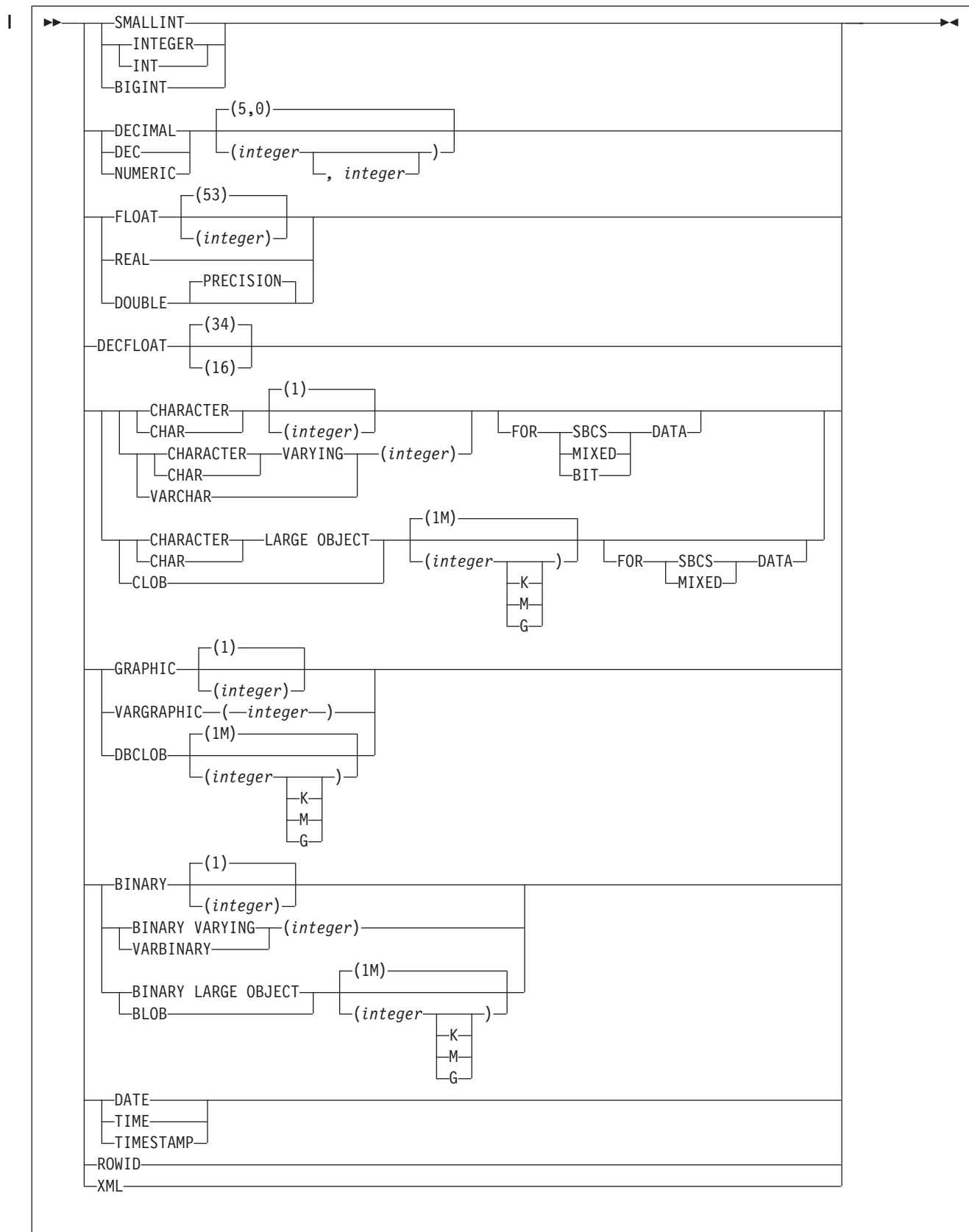
Notes:

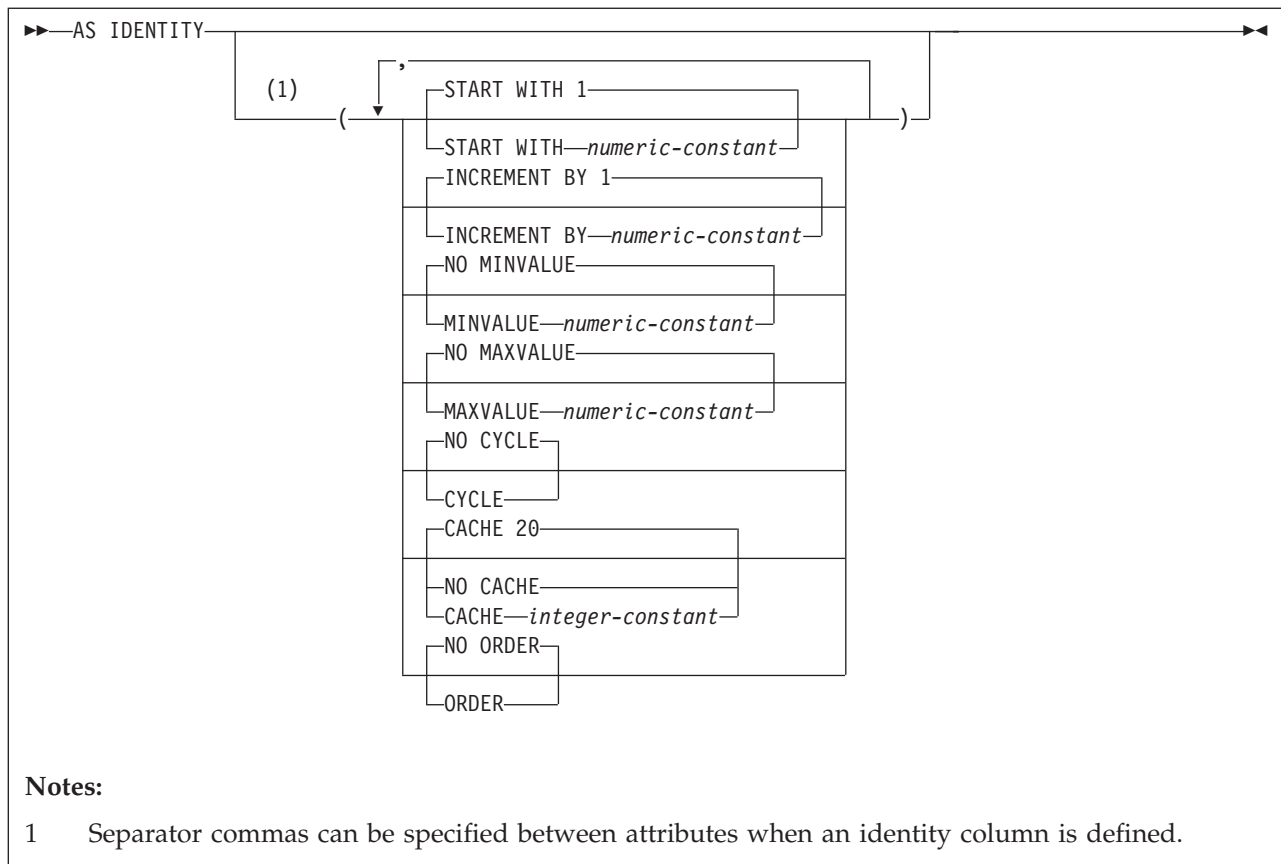
- 1 Data type is optional if *as-row-change-timestamp-clause* is specified
- 2 The same clause must not be specified more than once.
- 3 This form of the DEFAULT value can only be used with columns that are defined as a distinct type.
- 4 A column that has a ROWID data type (or a distinct type that is based on a ROWID data type) defaults to GENERATED ALWAYS.
- 5 AS SECURITY LABEL can be specified only for a CHAR(8) data type and requires that the NOT NULL and WITH DEFAULT clauses be specified.
- 6 IMPLICITLY HIDDEN must not be specified for a column defined as a ROWID, or a distinct type that is based on a ROWID.

data-type:

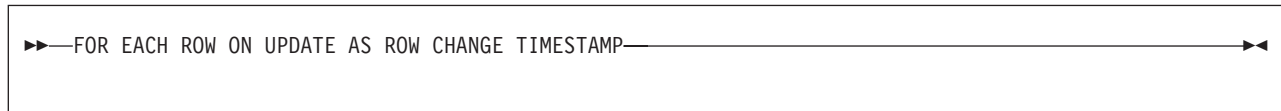


built-in-type:

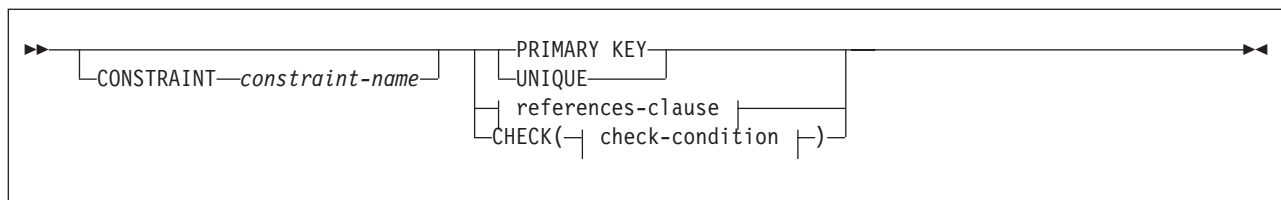




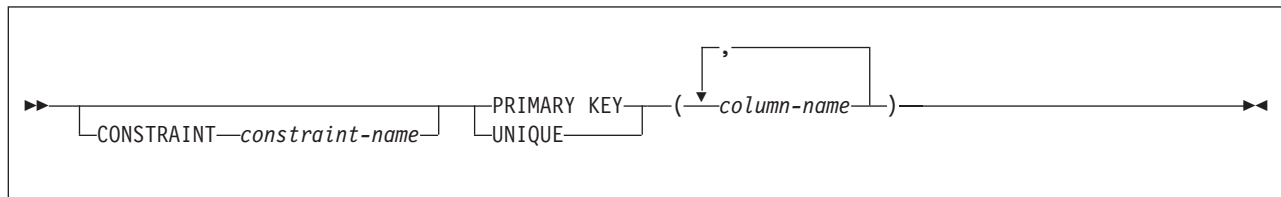
as-row-change-timestamp-clause:



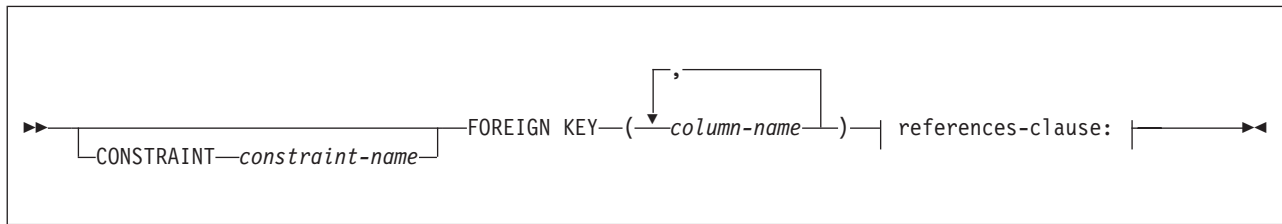
column-constraint:



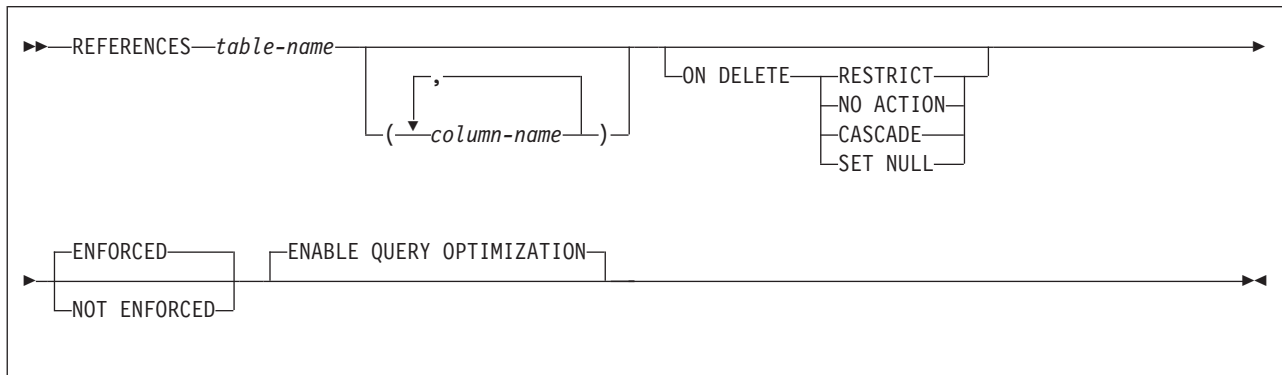
unique-constraint:



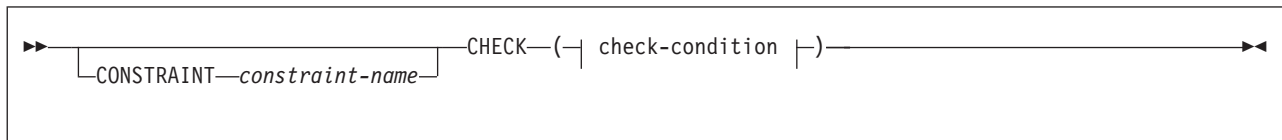
referential-constraint:



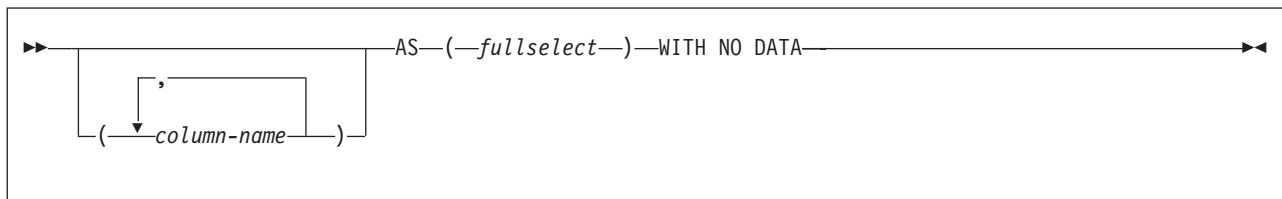
references-clause:



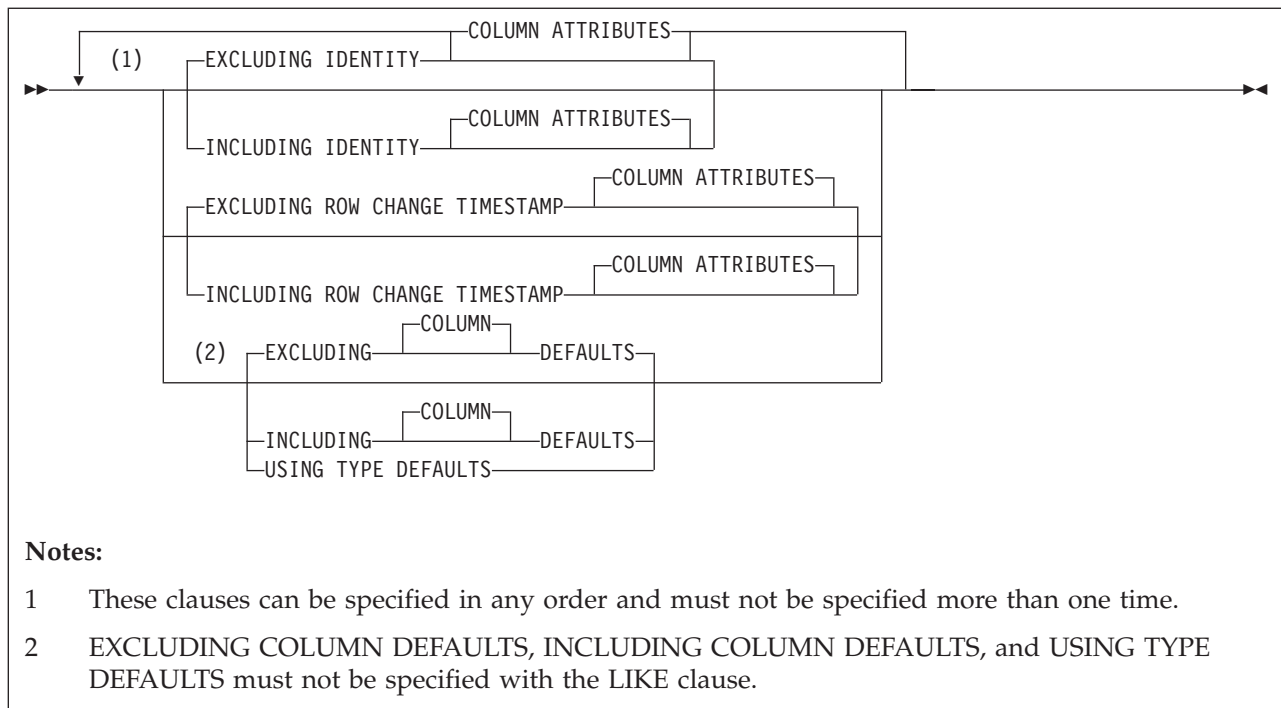
check-constraint:



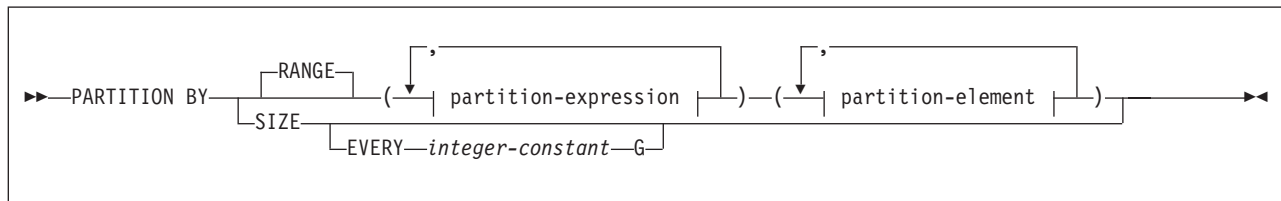
as-result-table:



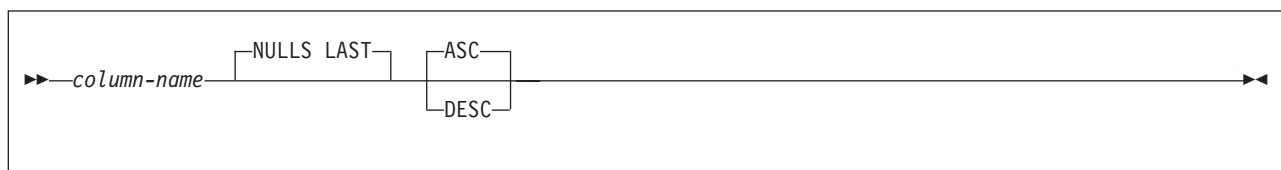
copy-options:



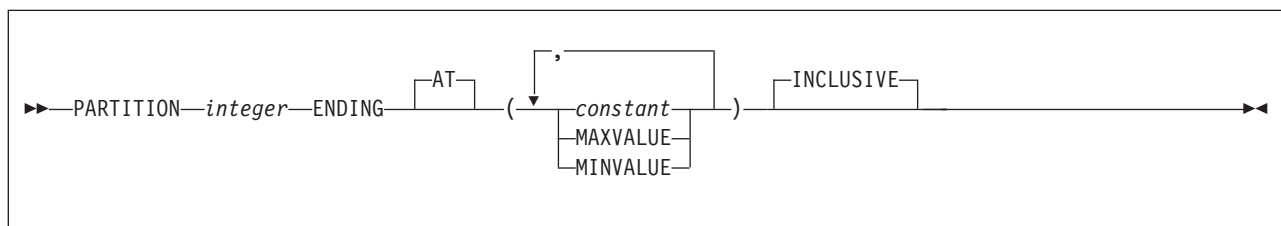
partitioning-clause:



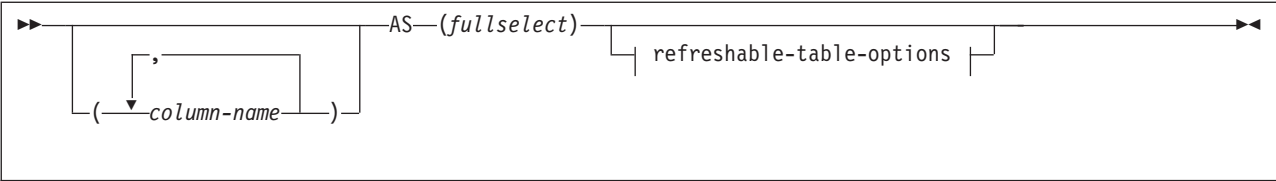
partition-expression:



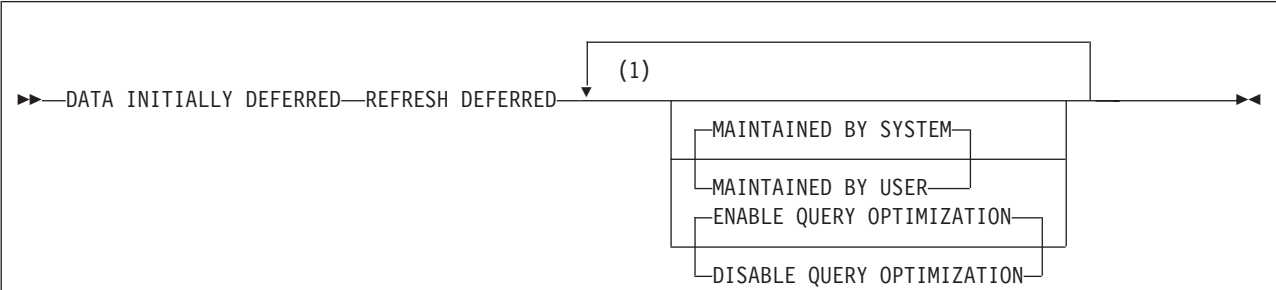
partition-element:



materialized-query-definition



refreshable-table-options:



Notes:

- 1 The same clause must not be specified more than one time.

Description

table-name

Names the table. The name, including the implicit or explicit qualifier, must not identify a table, view, alias, or synonym that exists at the current server.

If the name is qualified, the name can be a two-part or three-part name. If a three-part name is used, the first part must match the value of field DB2 LOCATION NAME on installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.)

column-name

Names a column of the table. For a dependent table, up to 749 columns can be named. For a table that is not a dependent, this number is 750. Do not qualify *column-name* and do not use the same name for more than one column of the table.

built-in-type

Specifies the data type of the column as one of the following built-in data types, and for character string data types, specifies the subtype. For more information about defining a table with a LOB column (CLOB, BLOB, or DBCLOB), see Creating a table with LOB columns.

SMALLINT

For a small integer.

INTEGER or INT

For a large integer.

BIGINT

For a big integer.

DECIMAL(integer, integer) or DEC(integer, integer)

DECIMAL(integer) or DEC(integer)

DECIMAL or DEC

For a decimal number. The first integer is the precision of the number. That is, the total number of digits, which can range from 1 to 31. The second integer is the scale of the number. That is, the number of digits to the right of the decimal point, which can range from 0 to the precision of the number.

You can use `DECIMAL(p)` for `DECIMAL(p,0)` and `DECIMAL` for `DECIMAL(5,0)`.

You can also use the word `NUMERIC` instead of `DECIMAL`. For example, `NUMERIC(8)` is equivalent to `DECIMAL(8)`. Unlike `DECIMAL`, `NUMERIC` has no allowable abbreviation.

DECFLOAT(*integer*)

For a decimal floating-point number. The value of *integer* must be either 16 or 34 and represents the number of significant digits that can be stored. If *integer* is omitted, the `DECFLOAT` column will be capable of representing 34 significant digits.

FLOAT(*integer*)

FLOAT

For a floating-point number. If *integer* is between 1 and 21 inclusive, the format is single precision floating-point. If the integer is between 22 and 53 inclusive, the format is double precision floating-point.

You can use `DOUBLE PRECISION` or `FLOAT` for `FLOAT(53)`.

REAL

For single precision floating-point.

DOUBLE or DOUBLE PRECISION

For double precision floating-point

CHARACTER(*integer*) or CHAR(*integer*)

CHARACTER or CHAR

For a fixed-length character string of length *integer*, which can range from 1 to 255. If the length specification is omitted, a length of 1 character is assumed.

VARCHAR(*integer*), CHAR VARYING(*integer*), or CHARACTER VARYING(*integer*)

For a varying-length character string of maximum length *integer*, which can range from 1 to the maximum record size minus 10 bytes. See Table 98 on page 1122 to determine the maximum record size.

FOR *subtype* DATA

Specifies a subtype for a character string column, which is a column with a data type of `CHAR`, `VARCHAR`, or `CLOB`. Do not use the `FOR subtype DATA` clause with columns of any other data type (including any distinct type). *subtype* can be one of the following:

SBCS

Column holds single-byte data.

MIXED

Column holds mixed data. Do not specify `MIXED` if the value of field `MIXED DATA` on installation panel `DSNTIPF` is `NO` unless the `CCSID UNICODE` clause is also specified, or the table is being created in a Unicode table space or database.

BIT

Column holds BIT data. Do not specify `BIT` for a `CLOB` column.

Only character strings are valid when subtype is BIT.

If you do not specify the FOR clause, the column is defined with a default subtype. For ASCII or EBCDIC data:

- The default is SBCS when the value of field MIXED DATA on installation panel DSNTIPF is NO.
- The default is MIXED when the value is YES.

For Unicode data, the default subtype is MIXED.

A security label column is always considered SBCS data, regardless of the encoding scheme of the table.

CLOB(*integer* [K|M|G]), CHAR LARGE OBJECT(*integer* [K|M|G]), or CHARACTER LARGE OBJECT(*integer* [K|M|G])

CLOB, CHAR LARGE OBJECT, or CHARACTER LARGE OBJECT

For a character large object (CLOB) string of the specified maximum length in bytes. The maximum length must be in the range of 1 to 2 147 483 647. A CLOB column has a varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see “Restrictions using LOBs” on page 86.

When *integer* is not specified, the default length is 1M. The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

integer The maximum value for *integer* is 2 147 483 647. The maximum length of the string is *integer*.

integer **K**

The maximum value for *integer* is 2 097 152. The maximum length is 1024 times *integer*.

integer **M**

The maximum value for *integer* is 2048. The maximum length is 1 048 576 times *integer*.

integer **G**

The maximum value for *integer* is 2. The maximum length is 1 073 741 824 times *integer*.

If you specify a value that evaluates to 2 gigabytes (2 147 483 648), DB2 uses a value that is one byte less, or 2 147 483 647.

GRAPHIC(*integer*)

GRAPHIC

For a fixed-length graphic string of length *integer*, which can range from 1 to 127. If the length specification is omitted, a length of 1 character is assumed.

VARGRAPHIC(*integer*)

For a varying-length graphic string of maximum length *integer*, which must range from 1 to $n/2$, where n is the maximum row size minus 2 bytes.

DBCLOB(*integer* [K|M|G])

DBCLOB

For a double-byte character large object (DBCLOB) string of the specified maximum length in double-byte characters. The maximum length must be in the range of 1 through 1 073 741 823. A DBCLOB column has a

varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see “Restrictions using LOBs” on page 86.

When *integer* is not specified, the default length is **1M**. The meaning of *integer* **K|M|G** is similar to CLOB. The difference is that the number specified is the number of double-byte characters.

BINARY(*integer*)

A fixed-length binary string of length *integer*. The *integer* can range from 1 through 255. If the length specification is omitted, a length of 1 byte is assumed.

BINARY VARYING(*integer*) or VARBINARY(*integer*)

A varying-length binary string of maximum length *integer*, which can range from 1 through 32704. The length is limited by the page size of the table space.

BLOB (*integer* **[K|M|G] or BINARY LARGE OBJECT(*integer* **[K|M|G]**)
BLOB or BINARY LARGE OBJECT**

For a binary large object (BLOB) string of the specified maximum length in bytes. The maximum length must be in the range of 1 through 2 147 483 647. A BLOB column has a varying-length. It cannot be referenced in certain contexts regardless of its maximum length. For more information, see “Restrictions using LOBs” on page 86.

When *integer* is not specified, the default length is **1M**. The meaning of *integer* **K|M|G** is the same as for CLOB.

DATE

For a date.

TIME

For a time.

TIMESTAMP

For a timestamp.

ROWID

For a row ID type.

A table can have only one ROWID column. The values in a ROWID column are unique for every row in the table and cannot be updated. You must specify NOT NULL with ROWID.

XML

For an XML document. Only well-formed XML documents can be inserted into an XML column.

distinct-type-name

Specifies the data type of the column is a distinct type (a user-defined data type). The length, precision, and scale of the column are respectively the length, precision, and scale of the source type of the distinct type. The privilege set must implicitly or explicitly include the USAGE privilege on the distinct type.

The encoding scheme of the distinct type must be the same as the encoding scheme of the table. The subtype for the distinct type, if it has the attribute, is the subtype with which the distinct type was created.

If the column is to be used in the definition of the foreign key of a referential constraint, the data type of the corresponding column of the parent key must have the same distinct type.

NOT NULL

Prevents the column from containing null values. Omission of NOT NULL implies that the column can contain null values.

column-constraint

The *column-constraint* of a *column-definition* provides a shorthand method of defining a constraint composed of a single column. Thus, if a *column-constraint* is specified in the definition of column C, the effect is the same as if that constraint were specified as a *unique-constraint*, *referential-constraint*, or *check-constraint* in which C is the only identified column.

CONSTRAINT *constraint-name*

Names the constraint. If a constraint name is not specified, a unique constraint name is generated. If the name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

PRIMARY KEY

Provides a shorthand method of defining a primary key composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

The NOT NULL clause must be specified with this clause. PRIMARY KEY cannot be specified more than one time in a column definition, and must not be specified if the UNIQUE clause is specified in the definition or if the definition is for a LOB, ROWID, DECFLOAT (including a distinct type that is based on a LOB, ROWID, or DECFLOAT data type), XML column, or a row change timestamp column.

The table is marked as unavailable until its primary index is explicitly created unless the CREATE TABLE statement is processed by the schema processor or the table space that contains the table is implicitly created. In that case, DB2 implicitly creates an index to enforce the uniqueness of the primary key and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

UNIQUE

Provides a shorthand method of defining a unique key composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE(C) clause is specified as a separate clause.

The NOT NULL clause must be specified with this clause. UNIQUE cannot be specified more than one time in a column definition and must not be specified if the PRIMARY KEY clause is specified in the column definition or if the definition is for a LOB, ROWID, row change timestamp, XML, or DECFLOAT column (including a distinct type that is based on a LOB, ROWID, or DECFLOAT data type).

The table is marked as unavailable until all the required indexes are explicitly created unless the CREATE TABLE statement is processed by the schema processor or the table space that contains the table is implicitly created. In that case, DB2 implicitly creates the indexes that are required for the unique keys and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

references-clause

The *references-clause* of a *column-definition* provides a shorthand method of defining a foreign key composed of a single column. Thus, if *references-clause* is specified in the definition of column C, the effect is the same as if the *references-clause* were specified as part of a FOREIGN KEY clause in which C is the only identified column.

Do not specify *references-clause* in the definition of a LOB, ROWID, XML, DECFLOAT, security label, or row change timestamp column; a LOB, ROWID, XML, DECFLOAT, security label, or row change timestamp column cannot be a foreign key.

CHECK (*check-condition*)

CHECK (*check-condition*) provides a shorthand method of defining a check constraint that applies to a single column. For conformance with the SQL standard, if CHECK is specified in the column definition of column C, no columns other than C should be referenced in the check condition of the check constraint. The effect is the same as if the check condition were specified as a separate clause.

DEFAULT

Specifies the default value that is assigned to the column in the absence of a value specified on an insert or update operation or LOAD. DEFAULT must not be specified more than one time in the same *column-definition*. Do not specify DEFAULT for the following types of columns because DB2 generates default values:

- An identity column (a column that is defined AS IDENTITY)
- A ROWID column (or a distinct type that is based on a ROWID)
- A row change timestamp column
- An XML column

Do not specify a value after the DEFAULT keyword for a security label column. DB2 provides the default value for a security label column.

If a value is not specified after DEFAULT, the default value depends on the data type of the column, as follows:

Data Type

Default Value

Numeric

0

Big integer

0

Fixed-length character string

Blanks

Fixed-length graphic string

Blanks

Fixed-length binary string

Hexadecimal zeros

Varying-length string

A string of length 0

Date CURRENT DATE

Time CURRENT TIME

Timestamp

CURRENT TIMESTAMP

Distinct type

The default of the source data type

A default value other than the one that is listed above can be specified in one of the following forms, except for a LOB column. The only form that can be specified for a LOB column is DEFAULT NULL. Unlike other varying-length strings, a LOB column can have the default value of only a zero-length string or null. Specify:

- WITH DEFAULT for a default value of an empty string
- DEFAULT NULL for a default value of null

Omission of NOT NULL and DEFAULT from a *column-definition*, for a column other than an identity column, is an implicit specification of DEFAULT NULL. For an identity column, it is an implicit specification of NOT NULL, and DB2 generates default values.

constant

Specifies a constant as the default value for the column. The value of the constant must conform to the rules for assigning that value to the column.

A character or graphic string constant must be short enough so that its UTF-8 representation requires no more than 1536. A hexadecimal graphic string constant (GX) cannot be specified.

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register at the time of an SQL data change statement or LOAD as the default value for the column. If SESSION_USER is specified, the data type of the column must be a character string with a length attribute greater than or equal to 8 characters when the value is expressed in CCSID 37.

CURRENT SQLID

Specifies the value of the SQL authorization ID of the process at the time of an insert or update operation or LOAD as the default value for the column. If CURRENT SQLID is specified, the data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL

Specifies null as the default value for the column. If NOT NULL is specified, DEFAULT NULL must not be specified with the same *column-definition*.

cast-function-name

The name of the cast function that matches the name of the distinct type for the column. A cast function can only be specified if the data type of the column is a distinct type.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type.

constant

Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type.

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register at the time a row is inserted as the default for the column. The source type of the distinct type of the column must be a CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the SESSION_USER special register.

CURRENT SQLID

Specifies the value of the CURRENT SQLID special register at the time a row is inserted as the default for the column. The source type of the distinct type of the column must be a CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL

Specifies the NULL value as the argument.

In a given column definition:

- DEFAULT and FIELDPROC cannot both be specified.
- NOT NULL and DEFAULT NULL cannot both be specified.

Table 96 summarizes the effect of specifying the various combinations of the NOT NULL and DEFAULT clauses on the CREATE TABLE statement *column-description* clause.

Table 96. Effect of specifying combinations of the NOT NULL and DEFAULT clauses

If NOT NULL is:	And DEFAULT is:	The effect is:
Specified ¹	Omitted	An error occurs if a value is not provided for the column on an insert or update operation or LOAD.
	Specified without an operand	The system defined nonnull default value is used.
	<i>constant</i>	The specified constant is used as the default value.
	SESSION_USER	The value of the SESSION_USER special register at the time of an insert or update operation or LOAD is used as the default value.
	CURRENT SQLID	The SQL authorization ID of the process at the time of an insert or update operation or LOAD is used as the default value.
	NULL	An error occurs during the execution of CREATE TABLE.
Omitted	Omitted	Equivalent to an implicit specification of DEFAULT NULL.
	Specified without an operand	The system defined nonnull default value is used.
	<i>constant</i>	The specified constant is used as the default value.
	SESSION_USER	The value of the SESSION_USER special register at execution time is used as the default value.
	CURRENT SQLID	The SQL authorization ID of the process is used as the default value.
	NULL	Null is used as the default value.

Note: The table does not apply to a column with a ROWID data type or to an identity column.

GENERATED

Specifies that DB2 generates values for the column. GENERATED must be specified if the column is to be considered one of the following types of columns:

- An identity column
- A row change timestamp column.
- A ROWID column

If the data type of the column is a ROWID (or a distinct type that is based on a ROWID), the default is GENERATED ALWAYS.

ALWAYS

Specifies that DB2 will always generate a value for the column when a row is inserted or updated and a default value must be generated. ALWAYS is the recommended value unless you are using data propagation.

BY DEFAULT

Specifies that DB2 will generate a value for the column when a row is inserted or updated and a default value must be generated, unless an explicit value is specified.

For a ROWID column, DB2 uses a specified value only if it is a valid row ID value that was previously generated by DB2 and the column has a unique, single-column index. Until this index is created on the ROWID column, the SQL insert or update operation and the LOAD utility cannot be used to add rows to the table. If the table space is explicitly created and the value of the CURRENT RULES special register is 'STD' when the CREATE TABLE statement is processed, or if the table space is implicitly created, DB2 implicitly creates the index on the ROWID column. The name of this index is 'I' followed by the first ten characters of the column name followed by seven randomly generated characters. If the column name is less than ten characters, DB2 adds underscore characters to the end of the name until it has ten characters. The implicitly created index has the COPY NO attribute.

For an identity column, DB2 inserts a specified value but does not verify that it is a unique value for the column unless the identity column has a unique, single-column index.

BY DEFAULT is the recommended value only when you are using data propagation.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP

Specifies that the column is a timestamp column for the table. DB2 generates a value for the column for each row as the row is inserted, and for any row in which any column is updated. The value that is generated for a row change timestamp column is a timestamp that corresponds to the insert or update time of the row. If multiple rows are inserted or updated with a single statement, the value for the row change timestamp column might be different for each row.

A table can only have one row change timestamp column.

If *data-type* is specified, it must be TIMESTAMP.

A row change timestamp column cannot have a DEFAULT clause. NOT NULL must be specified for a row change timestamp column.

AS IDENTITY

Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the

data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero, or a distinct type based on one of these types).

An identity column is implicitly NOT NULL. An identity column cannot have a WITH DEFAULT clause.

Defining a column AS IDENTITY does not necessarily ensure the uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the identity column.

START WITH *numeric-constant*

Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending identity column and the MAXVALUE for a descending identity column. This value is not necessarily the value that would be cycled to after reaching the maximum or minimum value for the identity column. The START WITH clause can be used to start the generation of values outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value (including 0) that does not exceed the value of a large integer constant, and could be assigned to the column without any non-zero digits existing to the right of the decimal point.

If this value is negative, the values for the identity column descend. If this value is 0 or positive, the values for the identity column ascend. The default is 1.

MINVALUE or NO MINVALUE

Specifies the minimum value at which a descending identity column either cycles or stops generating values or an ascending identity column cycles to after reaching the maximum value.

NO MINVALUE

Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MINVALUE becomes one of the following:

- For an ascending identity column, the value is the START WITH value or 1 if START WITH is not specified.
- For a descending identity column, the value is the minimum value of the data type of the column.

The default is NO MINVALUE.

MINVALUE *numeric-constant*

Specifies the numeric constant that is the minimum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column without non-zero digits existing to the right of the decimal point. The value must be less than or equal to the maximum value.

MAXVALUE or NO MAXVALUE

Specifies the maximum value at which an ascending identity column either cycles or stops generating values or a descending identity column cycles to after reaching the minimum value.

NO MAXVALUE

Specifies that the minimum end point of the range of values for the identity column has not be set. In such a case, the default value for MAXVALUE becomes one of the following:

- For an ascending identity column, the value is the maximum value of the data type associated with the column.
- For a descending identity column, the value is the START WITH value -1 if START WITH is not specified.

The default is NO MAXVALUE.

MAXVALUE *numeric-constant*

Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column without non-zero digits existing to the right of the decimal point. The value must be greater than or equal to the minimum value.

CYCLE or NO CYCLE

Specifies whether this identity column should continue to generate values after reaching either its maximum or minimum value. The default is NO CYCLE.

NO CYCLE

Specifies that values will not be generated for the identity column after the maximum or minimum value has been reached.

CYCLE

Specifies that values continue to be generated for the identity column after the maximum or minimum value has been reached. If this option is used, after an ascending identity column reaches the maximum value, it generates its minimum value. After a descending identity column reaches its minimum value, it generates its maximum value. The maximum and minimum values for the identity column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by DB2 for an identity column. However, if a unique index exists on the identity column and a non-unique value is generated for it, an error occurs.

CACHE *integer-constant* or NO CACHE

Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table. The default is CACHE 20.

NO CACHE

Specifies that values for the identity column and sequences are not preallocated and stored in the cache, ensuring that values will not be lost in the case of a system failure. In this case, every request for a new value for the identity column or sequence results in synchronous I/O.

In a data sharing environment, use **NO CACHE** if you need to guarantee that the identity column and sequence values are generated in the order in which they are requested.

CACHE *integer-constant*

Specifies the maximum number of values of the identity column sequence that DB2 can preallocate and keep in memory.

During a DB2 shutdown, all cached identity column values and sequence values that are yet to be assigned will be lost and will not be used. Therefore, the value that is specified for **CACHE** also represents the maximum number of identity column values and sequence values that will be lost during a DB2 shutdown.

The minimum value is 2.

In a data sharing environment, you can use the **CACHE** and **NO ORDER** options to allow multiple DB2 members to cache sequence values simultaneously.

ORDER or NO ORDER

Specifies whether the identity column values must be generated in order of request. The default is **NO ORDER**.

NO ORDER

Specifies that the values do not need to be generated in order of request.

ORDER

Specifies that the values are generated in order of request. Specifying **ORDER** might disable the caching of values. **ORDER** applies only to a single-application process.

In a data sharing environment, if the **CACHE** and **NO ORDER** options are in effect, multiple caches can be active simultaneously, and the requests for identity values from different DB2 members might not result in the assignment of values in strict numeric order. For example, if members DB2A and DB2B are using the identity column, and DB2A gets the cache values 1 to 20 and DB2B gets the cache values 21 to 40, the actual order of values assigned would be 1,21,2 if DB2A requested a value first, then DB2B requested, and then DB2A again requested. Therefore, to guarantee that identity values are generated in strict numeric order among multiple DB2 members using the same identity column, specify the **ORDER** option.

FIELDPROC *program-name*

Designates *program-name* as the field procedure exit routine for the column. Writing a field procedure exit routine is described in *DB2 Administration Guide*. A field procedure can be specified only for a column with a length attribute that is not greater than 255 bytes. **FIELDPROC** can only be specified for columns that are a built-in character string or graphic string data type that is not a LOB. The column must not be one of the following:

- a security label column
- a row change timestamp column

For more information about string comparisons with field procedures, see “Character and graphic string comparisons” on page 116.

The field procedure encodes and decodes column values: before a value is inserted in the column, it is passed to the field procedure for encoding. Before

a value from the column is used by a program, it is passed to the field procedure for decoding. A field procedure could be used, for example, to alter the sorting sequence of values entered in the column.

The field procedure is also invoked during the processing of the CREATE TABLE statement. When so invoked, the procedure provides DB2 with the column's *field description*. The field description defines the data characteristics of the encoded values. By contrast, the information you supply for the column in the CREATE TABLE statement defines the data characteristics of the decoded values.

constant

Is a parameter that is passed to the field procedure when it is invoked. A parameter list is optional. The *n*th parameter specified in the FIELDPROC clause on CREATE TABLE corresponds to the *n*th parameter of the specified field procedure. The maximum length of the parameter list is 254 bytes, including commas but excluding insignificant blanks and the delimiting parentheses.

If you omit FIELDPROC, the column has no field procedure.

AS SECURITY LABEL

Specifies that the column will contain security label values. This also indicates that the table is defined with multilevel security with row level granularity. A table can have only one security label column. To define a table with a security label column, the primary authorization ID of the statement must have a valid security label, and the RACF SECLABEL class must be active. In addition, the following conditions are also required:

- The data type of the column must be CHAR(8).
- The subtype of the column must be SBCS.
- The column must be defined with the NOT NULL and WITH DEFAULT clauses.
- The WITH DEFAULT clause must not specify a default value (DB2 determines the default value)
- No field procedures, check constraints, or referential constraints are defined on the column.
- No edit procedure for the table can be defined with row attribute sensitivity.

For information about using multilevel security, see *DB2 Administration Guide*.

IMPLICITLY HIDDEN

Specifies that the column is not visible in the result for SQL statements unless you explicitly refer to the column by name. For example, assuming that the table T1 includes a column that is defined with the IMPLICITLY HIDDEN clause, the result of a SELECT * would not include the implicitly hidden column. However, the result of a SELECT statement that explicitly refers to the name of the implicitly hidden column would include that column in the result table.

IMPLICITLY HIDDEN must not be specified for a column that is defined as a ROWID, or a distinct type that is based on a ROWID. IMPLICITLY HIDDEN must not be specified for all columns of a table.

CONSTRAINT *constraint-name*

Names the constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

PRIMARY KEY(*column-name*,...)

Defines a primary key composed of the identified columns. The clause must not be specified more than once and the identified columns must be defined as NOT NULL. Each *column-name* must be an unqualified name that identifies a column of the table except a LOB, a ROWID, a DECFLOAT (including a distinct type that is based on a LOB, ROWID, or DECFLOAT), an XML column, or a row change timestamp column and the same column must not be identified more than once. The number of identified columns must not exceed 64. In addition, the sum of the length attributes of the columns must not be greater than $2000 - 2m$, where m is the number of varying-length columns in the key.

The table is marked as unavailable until its primary index is explicitly created unless the table space is explicitly created and the CREATE TABLE statement is processed by the schema processor, or the table space is implicitly created. In that case, DB2 implicitly creates an index to enforce the uniqueness of the primary key and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

UNIQUE(*column-name*,...)

Defines a unique key composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table except a LOB, a ROWID, a DECFLOAT column (including a distinct type that is based on a LOB, ROWID, or DECFLOAT), or a row change timestamp column and the same column must not be identified more than once. Each identified column must be defined as NOT NULL. The number of identified columns must not exceed 64. In addition, the sum of the length attributes of the columns must not be greater than $2000 - 2m$, where m is the number of varying-length columns in the key.

A unique key is a duplicate if it is the same as the primary key or a previously defined unique key. The specification of a duplicate unique key is ignored with a warning.

The table is marked as unavailable until all the required indexes are explicitly created unless the table space is explicitly created and the CREATE TABLE statement is processed by the schema processor, or the table space is implicitly created. In these cases, DB2 implicitly creates the indexes that are required for the unique keys and the table definition is considered complete. (For more information about implicitly created indexes, see Implicitly created indexes.)

CONSTRAINT *constraint-name*

Names the referential constraint. If a constraint name is not specified, a unique constraint name is generated. If a name is specified, it must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

FOREIGN KEY (*column-name*,...) **references-clause**

Each specification of the FOREIGN KEY clause defines a referential constraint.

The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table except a LOB, ROWID, XML, DECFLOAT, security label, or row change timestamp column, and the same column must not be identified more than one time. The number of identified columns must not exceed 64, and the sum of their length attributes must not exceed 255 minus the number of columns that allow null values. The referential constraint is a duplicate if the FOREIGN KEY and parent table are the same as the FOREIGN KEY and

parent table of a previously defined referential constraint. The specification of a duplicate referential constraint is ignored with a warning.

REFERENCES *table-name (column-name,...)*

The table name specified after REFERENCES must identify a table that exists at the current server³¹, but it must not identify a catalog table or a declared global temporary table. In the following discussion, let T2 denote an identified table and let T1 denote the table that you are creating (T1 and T2 cannot be the same table³¹).

T2 must have a unique index and the privilege set must include the ALTER or REFERENCES privilege on the parent table, or the REFERENCES privilege on the columns of the nominated parent key.

The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The identified column cannot be a LOB, ROWID, XML, DECFLOAT, security label, or a row change timestamp column. The same column must not be identified more than one time.

The list of column names in the parent key must be identical to the list of column names in a primary key or unique key in the parent table T2. The column names must be specified in the *same order* as in the primary key or unique key. If any of the referenced columns in T2 has a non-numeric data type, T2 and T1 must use the same encoding scheme.

If a list of column names is not specified, T2 must have a primary key. Omission of a list of column names is an implicit specification of the columns of the primary key for T2.

The specified foreign key must have the same number of columns as the parent key of T2 and, except for their names, default values, null attributes and check constraints, the description of the *n*th column of the foreign key must be identical to the description of the *n*th column of the nominated parent key. If the foreign key includes a column defined as a distinct type, the corresponding column of the nominated parent key must be the same distinct type. If a column of the foreign key has a field procedure, the corresponding column of the nominated parent key must have the same field procedure and an identical field description. A field description is a description of the encoded value as it is stored in the database for a column that has been defined to have an associated field procedure.

The referential constraint specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent. A description of the referential constraint is recorded in the catalog.

ON DELETE

The delete rule of the relationship is determined by the ON DELETE clause. For more on the concepts used here, see “Referential constraints” on page 20.

SET NULL must not be specified unless some column of the foreign key allows null values. The default value for the rule depends on the value of the CURRENT RULES special register when the CREATE TABLE statement is processed. If the value of the register is 'DB2', the delete rule defaults to RESTRICT; if the value is 'STD', the delete rule defaults to NO ACTION.

31. This restriction is relaxed when the statement is processed by the schema processor and the other table is created within the same CREATE SCHEMA.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2. Then:

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

Let T3 denote a table identified in another FOREIGN KEY clause (if any) of the CREATE TABLE statement. The delete rules of the relationships involving T2 and T3 must be the same and must not be SET NULL if:

- T2 and T3 are the same table.
- T2 is a descendent of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendents of the same table and the deletion of rows from that table cascades to both T2 and T3.

ENFORCED or NOT ENFORCED

Indicates whether or not the referential constraint is enforced by DB2 during normal operations, such as insert, update, or delete.

ENFORCED

Specifies that the referential constraint is enforced by the DB2 during normal operations (such as insert, update, or delete) and that it is guaranteed to be correct. This is the default.

NOT ENFORCED

Specifies that the referential constraint is not enforced by DB2 during normal operations, such as insert, update, or delete. This option should only be used when the data that is stored in the table is verified to conform to the constraint by some other method than relying on the database manager.

ENABLE QUERY OPTIMIZATION

Specifies that the constraint can be used for query optimization. DB2 uses the information in query optimization using materialized query tables with the assumption that the constraint is correct. This is the default.

check-constraint

CONSTRAINT *constraint-name*

Names the check constraint. The constraint name must be different from the names of any referential, check, primary key, or unique key constraints previously specified on the table.

If *constraint-name* is not specified, a unique constraint name is derived from the name of the first column in the *check-condition* specified in the definition of the check constraint.

CHECK (*check-condition*)

Defines a check constraint. At any time, the *check-condition* must be true or unknown for every row of the table. A *check-condition* can evaluate to unknown if a column that is an operand of the predicate is null. A *check-condition* that evaluates to unknown does not violate the check constraint. A *check-condition* is a search condition, with the following restrictions:

- It can refer only to columns of table *table-name*; however, the columns cannot be LOB, ROWID, XML, DECFLOAT, or security label columns (including distinct types that are based on LOB, ROWID, and DECFLOAT data types).

- It can be up to 3800 bytes long, not including redundant blanks.
- It must not contain any of the following:
 - Subselects
 - Built-in or user-defined functions
 - CAST specifications
 - Cast functions other than those created when the distinct type was created
 - Host variables
 - Parameter markers
 - Special registers
 - Columns that include a field procedure
 - CASE expressions
 - ROW CHANGE expressions
 - Row expressions
 - DISTINCT predicates
 - GX constants (hexadecimal graphic string constants)
 - sequence references
 - OLAP specifications
- If a *check-condition* refers to a LOB column (including a distinct type that is based on a LOB), the reference must occur within a LIKE predicate.
- The AND and OR logical operators can be used between predicates. The NOT logical operator cannot be used.
- The first operand of every predicate must be the column name of a column in the table.
- The second operand in the *check-condition* must be either a constant or a column name of a column in the table.
 - If the second operand of a predicate is a constant, and if the constant is:
 - A floating-point number, then the column data type must be floating point.
 - A decimal number, then the column data type must be either floating point or decimal.
 - An integer number, then the column data type must not be a small integer.
 - A small integer number, then the column data type must be small integer.
 - A decimal constant, then its precision must not be larger than the precision of the column.
 - If the second operand of a predicate is a column, then both columns of the predicate must have:
 - The same data type.
 - Identical descriptions with the exception that the specification of the NOT NULL and DEFAULT clauses for the columns can be different, and that string columns with the same data type can have different length attributes.

LIKE *table-name* **or** *view-name*

Specifies that the columns of the table have exactly the same name and description as the columns of the identified table or view. The name specified after LIKE must identify a table or view that exists at the current server or a

declared temporary table. The privilege set must implicitly or explicitly include the SELECT privilege on the identified table or view. If the identified table or view contains a column with a distinct type, the USAGE privilege on the distinct type is also needed. An identified table must not be an auxiliary table or a clone table. An identified view must not include a column that is an explicitly defined ROWID column (including a distinct type that is based on a ROWID), an identity column, or a row change timestamp column. (For more information, see “Notes” on page 1071.)

The LIKE clause must not be specified in a Common Criteria environment.

The use of LIKE is an implicit definition of n columns, where n is the number of columns in the identified table (including implicitly hidden columns) or view. A column of the new table that corresponds to an implicitly hidden column in the existing table will also be defined as implicitly hidden. The implicit definition includes all attributes of the n columns as they are described in SYSCOLUMNS with the following exceptions:

- When a table is identified in the LIKE clause and a column in the table has a field procedure, the corresponding column of the new table has the same field procedure and the field description. However, the field procedure is not invoked during the execution of the CREATE TABLE statement. When a view is identified in the LIKE clause, none of the columns of the new table will have a field procedure. This is true even in the case that a column of a base table underlying the view has a field procedure defined.
- When a table is identified in the LIKE clause and a column in the table is an identity column, the corresponding column of the new table inherits only the data type of the identity column; none of the identity attributes of the column are inherited unless the INCLUDING IDENTITY clause is specified.
- When a table is identified in the LIKE clause and a column in the table is a security label column, the corresponding column of the new table inherits only the data type of the security label column; none of the security label attributes of the column are inherited.
- When a table is identified in the LIKE clause and the table contains a ROWID column (explicitly-defined or implicitly hidden), the corresponding columns of the new table inherits the ROWID columns.
- When a table is identified in the LIKE clause and the table contains a row change timestamp column, the corresponding column of the new table inherits only the data type of the row change timestamp column. The new column is not considered as a generated column.
- When a view is identified in the LIKE clause, the default value that is associated with the corresponding column of the new table depends on the column of the underlying base table for the view. If the column of the base table does not have a default, the new column does not have a default. If the column of the base table has a default, the default of the new column is:
 - Null if the column of the underlying base table allows nulls.
 - The default for the data type of the underlying base table if the underlying base table does not allow nulls.

The above defaults are chosen regardless of the current default of the base table column. The existence of an INSTEAD OF trigger does not affect the inheritance of default values.

- When a table that uses table-controlled partitioning is identified in the LIKE clause, the new table does not inherit partitioning scheme of that table. If desired, these partition boundaries can be added by specifying ALTER TABLE with the ADD PARTITION BY RANGE clause.

- The CCSID of the column is determined by the implicit or explicit CCSID clause. For more information, see the CCSID clause.

The implicit definition does not include any other attributes of the identified table or view. For example, the new table does not have a primary key or foreign key. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

AS (*fullselect*)

Specifies that the table definition is based on the column definitions from the result of a query expression. The use of AS (*fullselect*) is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *fullselect*. The columns of the new table are defined by the columns that result from the *fullselect*. Every select list element must have a unique name. The AS clause can be used in the *select-clause* to provide unique names.

The AS (*fullselect*) clause must not be specified in a Common Criteria environment.

The implicit definition includes the column name, data type, length, precision, scale, and nullability characteristic of each of the result columns of *fullselect*. The length of each column must not be 0. Other column attributes, such as DEFAULT and IDENTITY, are not inherited from the *fullselect*. A column of the new table that corresponds to an implicitly hidden column of a base table referenced in the *fullselect* is not considered hidden in the new table. A FIELDPROC is inherited for a column if the corresponding select item of the *fullselect* is a column that can be mapped to a column of a base table or a view. The new table contains a security label column if only one table in the *fullselect* contains a security label column and the primary authorization ID of the statement has a valid security label. If more than one table in the *fullselect* contains a security label column, an error occurs.

The implicit definition does not include any other attributes of the identified table or view. For example, the new table does not have a primary key or foreign key. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

The owner of the table being created must have the SELECT privilege on the tables or views referenced in the *fullselect*, or the privilege set must include SYSADM or DBADM authority for the database in which the tables of the *fullselect* reside. Having SELECT privilege means that the owner has at least one of the following authorizations:

- Ownership of the tables or views referenced in the *fullselect*
- The SELECT privilege on the tables and views referenced in the *fullselect*
- SYSADM authority
- DBADM authority for the database in which the tables of the *fullselect* reside

The rules for establishing the qualifiers for names used in the *fullselect* are the same as the rules used to establish the qualifiers for *table-name*.

The *fullselect* must not:

- Result in a column having a ROWID, BLOB, CLOB, DBCLOB, or XML data type or a distinct type based on these data types.
- Include multiple security label columns.
- Include a PREVIOUS VALUE or a NEXT VALUE expression.

- Refer to host variables or include parameter markers.
- Include an SQL data change statement in the FROM clause.
- Reference data that is encoded with different CCSID sets.
- Reference a remote object.

If the WITH NO DATA clause is not specified, the new table is considered a materialized query table.

WITH NO DATA

Specifies that the query is used only to define the attributes of the new table. The table is not populated using the results of the query and the REFRESH TABLE statement cannot be used.

copy-options

Specifies whether identity column attributes, row change timestamp attributes, and column defaults are inherited from the definition of the source of the result table.

EXCLUDING IDENTITY COLUMN ATTRIBUTES or INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies whether identity column attributes are inherited from the definition of the source of the result table.

EXCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that identity column attributes are not inherited from the definition of the source of the result table. This is the default.

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that, if available, identity column attributes (such as START WITH, INCREMENT BY, and CACHE values) are inherited from the definition of the source table. These attributes can be inherited if the element of the corresponding column in the table, view, or *fullselect* is the name of a column of a table or the name of a column of a view that directly or indirectly maps to the column name of a base table with the identity attribute. In other cases, the columns of the new temporary table do not inherit the identity attributes. The columns of the new table do not inherit the identity attributes in the following cases:

- The select list of the *fullselect* includes multiple instances of an identity column name (that is, selecting the same column more than one time).
- The select list of the *fullselect* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *fullselect* includes a set operation.

EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES or INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES

Specifies whether row change timestamp column attributes are inherited from the definition of the source of the result table.

EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES

Specifies that row change timestamp column attributes are not inherited from the source result table definition. This is the default.

INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES

Specifies that, if available, row change timestamp column attributes are inherited from the definition of the source table. These attributes can be inherited if the element of the corresponding column in the table,

view, or *fullselect* is the name of a column of a table or the name of a column of a view that directly or indirectly maps to the column name of a base table defined as a row change timestamp column. In other cases, the columns of the new temporary table do not inherit the row change timestamp column attributes. The columns of the new table do not inherit the row change timestamp attributes in the following cases:

- The select list of the *fullselect* includes multiple instances of a row change timestamp column name (that is, selecting the same column more than one time).
- The select list of the *fullselect* includes multiple row change timestamp column names (that is, it involves a join).
- The row change timestamp column is included in an expression in the select list.
- The *fullselect* includes a set operation (such as union).

EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, or USING TYPE DEFAULTS

Specifies whether column defaults are inherited from the source result table definition. EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, and USING TYPE DEFAULTS must not be specified if the LIKE clause is specified.

EXCLUDING COLUMN DEFAULTS

Specifies that the column defaults are not inherited from the definition of the source table. The default values of the column of the new table are either null or there are no default values. If the column can be null, the default is the null value. If the column cannot be null, there is no default value, and an error occurs if a value is not provided for a column on an insert or update operation, or LOAD for the new table.

INCLUDING COLUMN DEFAULTS

Specifies that column defaults for each updatable column of the definition of the source table are inherited. Columns that are not updatable do not have a default defined in the corresponding column of the created table. The existence of an INSTEAD OF trigger for a view does not affect the inheritance of default values.

USING TYPE DEFAULTS

Specifies that the default values for the table depend on data type of the columns that result from *fullselect*, as follows:

Data type

Default value

Numeric

0

Fixed-length character string

Blanks

Fixed-length graphic string

Blanks

Fixed-length binary string

Hexadecimal zeros

Varying-length string

A string of length 0

Fixed-length char or fixed-length graphic

A string of blanks

Fixed-length binary

Hexadecimal zeros

Date CURRENT DATE
Time CURRENT TIME
Timestamp
CURRENT TIMESTAMP

WITH RESTRICT ON DROP

Indicates that the table can be dropped only by using REPAIR DBD DROP. In addition, the database and table space that contain the table can be dropped only by using REPAIR DBD DROP.

IN *database-name.table-space-name* **or** **IN DATABASE** *database-name*

Identifies the database and table space in which the table is created. Both forms are optional.

If you specify both a database and a table space, the database must be described in the current server's catalog, and must not be DSNDB06 or a work file database. The table space must belong to the database that you specify and must not be an XML table space.

If you specify a database but not a table space, a table space is implicitly created in *database-name*. The name of the table space is derived from the name of the table. The qualifier of the table space is the same as the qualifier of the table. The buffer pool that is used is the default buffer pool for user data that is specified on installation panel DSNTIP1. If you specify a table space but not a database, the database that contains the table space is used.

If you specify neither a table space or a database, a database is implicitly created with the name DSNxxxxx, where xxxxx is a five digit number. A table space is also implicitly created.

If you specify a table space, it must not be one that was created implicitly, be a partitioned or a partition-by-growth table space that already contains a table, or be a LOB table space. If you specify a partitioned table space, you cannot load or use the table until its partitioned scheme is created.

You cannot specify the name of an implicitly created database. That is, you specify a database name that is eight characters, DSNxxxxx, where xxxxx is a five digit number.

To create a table space implicitly, the privilege set must have: SYSADM or SYSCTRL authority; DBADM, DBCTRL, or DBMAINT authority for the database; or the CREATETS privilege for the database. You must also have the USE privilege for the database's default buffer pool and default storage group.

If you specify a table space name, you must have SYSADM or SYSCTRL authority, DBADM authority for the database, or the USE privilege for the table space.

PARTITION BY RANGE or PARTITION BY SIZE

Specifies the partitioning scheme for the table.

PARTITION BY RANGE

Specifies the range partitioning scheme for the table (the columns that are used to partition the data). When this clause is specified, the table space is complete, and it is not necessary to create a partitioned index on the table. If this clause is used, the ENDING AT clause cannot be used on a subsequent CREATE INDEX statement for this table.

If this clause is specified, the *IN database-name.table-space-name* clause is required. This clause applies only to tables in a partitioned table space. PARTITION BY RANGE must not be specified for a table that is created in a partition-by-growth table space.

partition-expression

Specifies the key data over which the range is defined to determine the target data partition of the data.

column-name

Specifies the columns of the key. Each *column-name* must identify a column of the table. Do not specify more than 64 columns or the same column more than one time. The sum of length attributes of the columns must not be greater than 255 - *n*, where *n* is the number of columns that can contain null values. Do not specify a qualified column name. Do not specify a column for *column-name* if the column is defined as follows:

- a LOB column (or a column with a distinct type that is based on a LOB data type)
- a BINARY column (or a column with a distinct type that is based on a BINARY data type)
- a VARBINARY column (or a column with a distinct type that is based on a VARBINARY data type)
- a DECFLOAT column (or a column with a distinct type that is based on a DECFLOAT data type)
- an XML column
- a row change timestamp column

NULLS LAST

Specifies that null values are treated as positive infinity for purposes of comparison.

ASC

Puts the entries in ascending order by the column. ASC is the default.

DESC

Puts the entries in descending order by the column.

partition-element

Specifies ranges for a data partitioning key and the table space where rows of the table in the range will be stored.

PARTITION integer

integer is the physical number of a partition in the table space. A PARTITION clause must be specified for every partition of the table space. In this context, highest means highest in the sorting sequences of the columns. In a column defined as ascending (ASC), highest and lowest have their usual meanings. In a column defined as descending (DESC), the lowest actual value is highest in the sorting sequence.

ENDING AT (*constant*, MAXVALUE, or MINVALUE, ...)

Defines the limit key for a partition boundary. Specify at least one value (*constant*, MAXVALUE, or MINVALUE) after ENDING AT in each PARTITION clause. You can use as many values as there are columns in the key. The concatenation of all values is the highest value of the key for ascending and the lowest for descending.

constant

Specifies a constant value with a data type that must conform to the rules for assigning that value to the column. If a string constant is longer or shorter than required by the length

attribute of its column, the constant is either truncated or padded on the right to the required length. If the column is ascending, the padding character is X'FF'. If the column is descending, the padding character is X'00'. The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding column. A hexadecimal string constant (GX) cannot be specified.

MAXVALUE

Specifies a value greater than the maximum value for the limit key of a partition boundary (that is, all X'FF' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are ascending, a constant or the MINVALUE clause cannot be specified following MAXVALUE. After MAXVALUE is specified, all subsequent columns must be MAXVALUE.

MINVALUE

Specifies a value that is smaller than the minimum value for the limit key of a partition boundary (that is, all X'00' regardless of whether the column is ascending or descending). If all of the columns in the partitioning key are descending, a constant or the MAXVALUE clause cannot be specified following MINVALUE. After MINVALUE is specified, all subsequent columns must be MINVALUE.

The key values are subject to the following rules:

- The first value corresponds to the first column of the key, the second value to the second column, and so on. Using fewer values than there are columns in the key has the same effect as using the highest or lowest values for the omitted columns, depending on whether they are ascending or descending.
- The highest value of the key in any partition must be lower than the highest value of the key in the next partition for ascending cases.
- The values specified for the last partition are enforced. The value specified for the last partition is the highest value of the key that can be placed in the table. Any key values greater than the value specified for the last partition are out of range.
- The combination of the number of table space partitions and the corresponding limit key size cannot exceed the number of partitions * (106 + limit key size in bytes) < 65394.
- If the concatenation of all the values exceeds 255 bytes, only the first 255 bytes are considered.
- If a key includes a ROWID column or a column with a distinct type that is based on a ROWID data type, 17 bytes of the constant that is specified for the corresponding ROWID column are considered.
- If a null value is specified for the partitioning key and the key is ascending, an error is returned unless MAXVALUE is specified. If the key is descending, an error is returned unless MINVALUE is specified..

INCLUSIVE

Specifies that the specified range values are included in the data partition.

PARTITION BY SIZE

Specifies that the table is created in a partition-by-growth table space. If the IN clause is also specified, the IN clause must identify a partition-by-growth table space.

EVERY integer G

Specifies that the table is to be partitioned by growth, every *integer* G bytes. *integer* must not be greater than 64. If the IN clause identifies a table space, *integer* must be the same as the DSSIZE value that is in effect for the table space that will contain the table.

EDITPROC *program-name*

Identifies the user-written code that implements the edit procedure for the table. The edit procedure must exist at the current server. The procedure is invoked during the execution of an SQL data change statement or LOAD and all row retrieval operations on the table.

An edit routine receives an entire table row, and can transform that row in any way. Also, it receives a transformed row and must change the row back to its original form.

You must not specify an edit routine for a table with a LOB column. For information on writing an EDITPROC exit routine, see Edit procedures.

WITH ROW ATTRIBUTES

Specifies that the edit procedure parameter list contains an address for the description of a row. WITH ROW ATTRIBUTES must not be specified for a table with an identity, LOB, XML, ROWID, or SECURITY LABEL column. WITH ROW ATTRIBUTES is the default. When WITH ROW ATTRIBUTES is specified, the column names in the table must not be longer than 18 EBCDIC SBCS characters in length.

WITHOUT ROW ATTRIBUTES

Specifies that the description of the row is not provided to the edit procedure. On entry to the edit procedure, the address for the row description in the parameter list contains a value of zero.

VALIDPROC *program-name*

Designates *program-name* as the validation exit routine for the table. Writing a validation exit routine is described in *DB2 Administration Guide*.

The validation routine can inhibit a load, insert, update, or delete operation on any row of the table: before the operation takes place, the procedure is passed the row. The values that are represented by any LOB or XML columns in the table are not passed to the validation routine. On an insert or update operation, if the table has a security label column and the user does not have write-down privilege, the user's security label value is passed to the validation routine as the value of the column. After examining the row, the procedure returns a value that indicates whether the operation should proceed. A typical use is to impose restrictions on the values that can appear in various columns.

A table can have only one validation procedure at a time. In an ALTER TABLE statement, you can designate a replacement procedure or discontinue the use of a validation procedure.

If you omit VALIDPROC, the table has no validation routine.

AUDIT

Identifies the types of access to this table that causes auditing to be performed. For information about audit trace classes, see *DB2 Administration Guide*.

If a materialized query table is refreshed with the REFRESH TABLE statement, the auditing also occurs during the REFRESH TABLE operation. **AUDIT** works as usual for LOAD and SQL data change operations on a user-maintained materialized query table.

NONE

Specifies that no auditing is to be done when this table is accessed. This is the default.

CHANGES

Specifies that auditing is to be done when the table is accessed during the first insert, update, or delete operation. However, the auditing is done only if the appropriate audit trace class is active.

ALL

Specifies that auditing is to be done when the table is accessed during the first operation of any kind performed by a utility or application process. However, the auditing is done only if the appropriate audit trace class is active and the access is not performed with COPY, RECOVER, REPAIR, or any stand-alone utility.

If the table is subsequently altered with an ALTER TABLE statement, the ALTER TABLE statement is audited for successful and failed attempts in the following cases, if the appropriate audit trace class is active:

- **AUDIT** attribute is changed to **NONE**, **CHANGES**, or **ALL** on an audited or non-audited table.
- **AUDIT CHANGES** or **AUDIT ALL** is in effect.

OBID *integer*

Identifies the OBID to be used for this table. An OBID is the identifier for an object's internal descriptor. The integer must not identify an existing or previously used OBID of the database. If you omit OBID, DB2 generates a value.

The following statement retrieves the value of OBID:

```
SELECT OBID
FROM SYSIBM.SYSTABLES
WHERE CREATOR = 'ccc' AND NAME = 'nnn';
```

Here, *nnn* is the table name and *ccc* is the creator of the table.

DATA CAPTURE

Specifies whether the logging of the following actions on the table is augmented by additional information:

- SQL data change operations
- Adding columns (using the ADD COLUMN clause of the ALTER TABLE statement)
- Changing columns (using the ALTER COLUMN clause of the ALTER TABLE statement)

For guidance on intended uses of the expanded log records, see:

- The description of data propagation to IMS in *IMS DataPropagator: An Introduction*
- The instructions for using Remote Recovery Data Facility (RRDF) in *Remote Recovery Data Facility Program Description and Operations*
- The instructions for reading log records in *DB2 Administration Guide*

If a materialized query table is refreshed with the REFRESH TABLE statement, the logging of the augmented information occurs during the REFRESH TABLE

operation. DATA CAPTURE works as usual for insert, update, and delete operations on a user-maintained materialized query table.

NONE

Do not record additional information to the log. This is the default.

CHANGES

Write additional data about SQL updates to the log. Information about the values that are represented by any LOB or XML columns is not available. Do not specify DATA CAPTURE CHANGES for tables that reside in table spaces that specify NOT LOGGED.

CCSID *encoding-scheme*

Specifies the encoding scheme for string data stored in the table. If the IN clause is specified with a table space, the value must agree with the encoding scheme that is already in use for the specified table space. The specific CCSIDs for SBCS, mixed, and graphic data are determined by the table space or database specified in the IN clause. If the IN clause is not specified, the value specified is used for the table being created as well as for the table space that DB2 implicitly creates. The specific CCSIDs for SBCS, mixed, and graphic data are determined by the default CCSIDs for the server for the specified encoding scheme. The valid values are ASCII, EBCDIC, and UNICODE.

If the CCSID clause is not specified, the encoding scheme for the table depends on the IN clause:

- If the IN clause is specified, the encoding scheme already in use for the table space or database specified in the IN clause is used.
- If the IN clause is not specified, the encoding scheme of the new table is the same as the scheme for the table that is specified in the LIKE clause.

If the CCSID clause is specified for a materialized query table, the encoding scheme specified in the clause must be the same as the scheme for the result CCSID of the fullselect. The CCSID must also be the same as the CCSID of the table space for the table being created.

VOLATILE or NOT VOLATILE

Specifies how DB2 is to choose access to the table.

VOLATILE

Specifies that index access should be used on this table whenever possible for SQL operations. However, be aware that list prefetch and certain other optimization techniques are disabled when VOLATILE is used.

One instance in which use of VOLATILE might be desired is for a table whose size can vary greatly. If statistics are taken when the table is empty or has only a few rows, those statistics might not be appropriate when the table has many rows. Another instance in which use of VOLATILE might be desired is for a table that contains groups of rows, as defined by the primary key on the table. All but the last column of the primary key of such a table indicate the group to which a given row belongs. The last column of the primary key is the sequence number indicating the order in which the rows are to be read from the group. VOLATILE maximizes concurrency of operations on rows within each group, since rows are usually accessed in the same order for each operation.

NOT VOLATILE

Specifies that SQL access to this table should be based on the current statistics. NOT VOLATILE is the default.

CARDINALITY

An optional keyword that currently has no effect, but that is provided for DB2 family compatibility.

APPEND NO or APPEND YES

Specifies whether append processing is used for the table. The APPEND clause must not be specified for a table that is created in a work file table space.

NO Specifies that append processing is not used for the table. For insert and LOAD operations, DB2 will attempt to place data rows in a well clustered manner with respect to the value in the row's cluster key column.

NO is the default.

YES

Specifies that data rows are to be placed into the table by disregarding the clustering during insert and LOAD operations.

materialized-query-definition

Specifies that the column definitions of the materialized query table are based on the result of a fullselect. If *materialized-query-table-options* are specified, the REFRESH TABLE statement can be used to populate the table with the results of the fullselect.

column-name

Names the columns in the table. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the table inherit the names of the columns of the result table of the fullselect.

A list of column names must be specified if the result table of the fullselect has duplicate column names or an unnamed column. An unnamed column is a column derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

AS (*fullselect*)

Specifies that the table definition is based on the column definitions from the result of a query expression. The use of AS (*fullselect*) is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *fullselect*. The columns of the new table are defined by the columns that result from the *fullselect*. Every select list element must have a unique name. The AS clause can be used in the *select-clause* to provide unique names.

The implicit definition includes the column name, data type, length, precision, scale, and nullability characteristic of each of the result columns of *fullselect*. The length of each result column must not be 0. Other column attributes, such as DEFAULT, IDENTITY, and unique constraints, are not inherited from the *fullselect*. A column of the new table that corresponds to an implicitly hidden column of a base table referenced in the *fullselect* is not considered hidden in the new table. The generated column attributes are not inherited from the *fullselect*. That is, the new column of the materialized query table is not considered as a generated column. A FIELDPROC is inherited for a column if the corresponding select item of the *fullselect* is a column that can be directly mapped to a column of a base table or a view in the FROM clause of the *fullselect*. The materialized query table contains a security label column if only one table in the *fullselect* contains a security label column and the primary authorization ID of the

statement has a valid security label. If more than one table in the *fullselect* contains a security label column, an error occurs.

The owner of the table being created must have the SELECT privilege on the tables or views referenced in the fullselect, or the privilege set must include SYSADM or DBADM authority for the database in which the tables of the fullselect reside. Having SELECT privilege means that the owner has at least one of the following authorizations:

- Ownership of the tables or views referenced in the fullselect
- The SELECT privilege on the tables and views referenced in the fullselect
- SYSADM authority
- DBADM authority for the database in which the tables of the fullselect reside

The rules for establishing the qualifiers for names used in the fullselect are the same as the rules used to establish the qualifiers for *table-name*.

The following restrictions apply when creating materialized query tables. When *fullselect* does not satisfy the restrictions, an error occurs:

- The fullselect must not refer to host variables or include parameter markers.
- When **DISABLE QUERY OPTIMIZATION** is specified, the following additional restrictions apply:
 - The fullselect cannot contain a reference to a created global temporary table or a declared global temporary table.
 - The fullselect cannot reference another materialized query table.
- When **ENABLE QUERY OPTIMIZATION** is specified, the following additional restrictions apply:
 - The fullselect must be a subselect.
 - The outermost SELECT list of the subselect must not reference data that is encoded with different CCSID sets.
 - If the subselect references a view, the fullselect in the view definition must satisfy the restrictions for using a fullselect when the **ENABLE QUERY OPTIMIZATION** clause is specified.
 - The subselect cannot contain:
 - A FETCH FIRST clause
 - A nested table expression or view that requires temporary materialization
 - A join using the INNER JOIN syntax
 - An outer join
 - A special register
 - A scalar fullselect
 - A row change timestamp column
 - Any predicates that include a subquery
 - A row expression predicate
 - A function or expression that is not deterministic or that has external actions
 - A ROW CHANGE expression
 - lateral correlation

refreshable-table-options

Specifies the options for a refreshable materialized query table. The

ORDER BY clause is allowed, but it is used only by REFRESH. The ORDER BY clause can improve the locality of reference of data in the materialized query table.

DATA INITIALLY DEFERRED

Specifies that the data is not inserted into the materialized query table when it is created. Use the REFRESH TABLE statement to populate the materialized query table, or use the INSERT statement to insert data into a user-maintained materialized query table.

REFRESH DEFERRED

Specifies that the data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time when the REFRESH TABLE statement is processed or when it was last updated for a user-maintained materialized query table.

MAINTAINED BY SYSTEM or MAINTAINED BY USER

Specifies how the data in the materialized query table is maintained.

MAINTAINED BY SYSTEM

Specifies that the materialized query table is maintained by the system. Only the REFRESH statement is allowed on the table. This is the default.

MAINTAINED BY USER

Specifies that the materialized query table is maintained by the user, who can use the LOAD utility, an SQL data change statement, a SELECT from data change statement, or REFRESH TABLE SQL statements on the table.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION

Specifies whether this materialized query table can be used for optimization.

ENABLE QUERY OPTIMIZATION

Specifies that the materialized query table can be used for query optimization. If the fullselect specified does not satisfy the restrictions for query optimization, an error occurs.

ENABLE QUERY OPTIMIZATION is the default.

DISABLE QUERY OPTIMIZATION

Specifies that the materialized query table cannot be used for query optimization. The table can still be queried directly.

Notes®

Owner privileges: The owner of the table has all table privileges (see “GRANT (table or view privileges)” on page 1355) with the ability to grant these privileges to others. For more information about ownership of the object, see “Authorization, privileges, and object ownership” on page 60.

Table design: Designing tables is part of the process of database design. For information on design, see *Introduction to DB2 for z/OS*.

Creating a table in a segmented table space: A table cannot be created in a segmented table space if:

- The available space in the data set is less than the segment size specified for the table space, and

- The data set cannot be extended.

Creating a table with graphic and mixed data columns: You cannot create an ASCII or EBCDIC table with a GRAPHIC, VARGRAPHIC, or DBCLOB column or a CHAR, VARCHAR, or CLOB column defined as FOR MIXED DATA when the setting for installation option MIXED DATA is NO.

Creating a table with distinct type columns based on LOB, ROWID, and DECFLOAT columns: Because a distinct type is subject to the same restrictions as its source type, all the syntactic rules that apply to LOB columns (CLOB, DBCLOB, and BLOB), ROWID columns, and DECFLOAT columns apply to distinct type columns that are based on LOBs, row IDs, and DECFLOATs. For example, a table cannot have both an explicitly defined ROWID column and a column with a distinct type that is based on a row ID.

Creating a table with LOB columns: A table with a LOB column (CLOB, DBCLOB, or BLOB) must also have a ROWID column and one or more auxiliary tables. When you create the table, DB2 implicitly generates a ROWID column for you. This is called an *implicitly hidden ROWID column*, and DB2:

- Creates the column with a name of DB2_GENERATED_ROWID_FOR_LOBS nn . DB2 appends nn only if the column name already exists in the table, replacing nn with 00 and incrementing by 1 until the name is unique within the row.
- Defines the column as GENERATED ALWAYS.
- Appends the implicitly hidden ROWID column to the end of the row after all the other explicitly defined columns.

For example, assume that DB2 generated an implicitly hidden ROWID column named DB2_GENERATED_ROWID_FOR_LOBS for table MYTABLE. The result table for a SELECT * statement for table MYTABLE would not contain that ROWID column. However, the result table for SELECT COL1, DB2_GENERATED_ROWID_FOR_LOBS would include the implicitly hidden ROWID column.

The definition of the table is marked incomplete until an auxiliary table is created in a LOB table space for each LOB column in the base table and index is created on each auxiliary table. The auxiliary table stores the actual values of a LOB column. If you create a table with a LOB column in a partitioned table space, there must be one auxiliary table defined for each partition of the base table space.

Unless DB2 implicitly creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column in the base table, you need to create these objects using the CREATE TABLESPACE, CREATE AUXILIARY TABLE, and CREATE INDEX statements.

If the table space that contains the table is explicitly created and the value of the CURRENT RULES special register is 'STD' when the CREATE TABLE statement is processed, or the table space that contains the table is implicitly created, DB2 implicitly creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column in the base table.

The privilege set must include the following privileges:

- The USE privilege on the buffer pool and the storage group that is used by the XML objects

- If the base table space is explicitly created, CREATETS is also required on the database that contains the table (DSNDB04 if the database is implicitly created)

DB2 chooses the names of implicitly created objects using these conventions:

LOB table space

Name is 8 characters long, consisting of an 'L' followed by 7 random characters.

auxiliary table

Name is 18 characters long. The first five characters of the name are the first five characters of the name of the base table. The second five characters are the first five characters of the name of the LOB column. The last eight characters are randomly generated. If a base table name or a LOB column name is less than five characters, DB2 adds underscore characters to the name to pad it to a length of five characters.

index on the auxiliary table

Name is 18 characters long. The first character of the name is an 'I'. The next ten characters are the first ten characters of the name of the auxiliary table. The last seven characters are randomly generated. The index has the COPY NO attribute.

The other attributes of these implicitly created objects are those that would have been created by their respective CREATE statements with all optional clauses omitted, with the following exceptions:

- The database name is the database name of the base table.
- If the LOB table space is implicitly created, the buffer pool is determined by the DEFAULT BUFFER POOL FOR USER LOB DATA fields of install panel DSNTIP1. The appropriate USE privilege is required on that buffer pool.

Utility REPORT TABLESPACESET identifies the LOB table spaces that DB2 implicitly created.

Creating a table with an XML column: When a table is created with an XML column, an XML table space, XML table, and a node ID index and document ID index are implicitly created.

The privilege set must include the following privileges:

- The USE privilege on the buffer pool and the storage group that is used by the XML objects
- If the base table space is explicitly created, CREATETS is also required on the database that contains the table (DSNDB04 if the database is implicitly created)

The buffer pool for the XML table space is determined by the DEFAULT BUFFER POOL FOR USER XML DATA fields of install panel DSNTIP1. The appropriate USE privilege is required on that buffer pool.

The XML table space will have a larger DSSIZE than the base table space if the base table space is partitioned by range. If the base table space is partitioned by growth, the default DSSIZE of 4GB will be used for the XML table space. The DSSIZE for an XML table space that is associated with a partitioned by range base table space is determined as follows.

Table 97. Default DSSIZE for XML table spaces, given base table space DSSIZE and page size

Base table space DSSIZE	4KB base page size	8KB base page size	16KB base page size	32KB base page size
1GB - 4GB	4GB	4GB	4GB	4GB
8GB	32GB	16GB	16GB	16GB
16GB	64GB	32GB	16GB	16GB
32GB	64GB	64GB	32GB	16GB
64GB	64GB	64GB	64GB	32GB

For example: for a base table space that has a DSSIZE of 8GB and a page size of 8KB, the XML table space will have a DSSIZE of 16GB.

Naming convention for implicitly created XML objects: Implicitly created XML table spaces names will be *Xyyyynnnn*, where *yyy* is the first three UTF-8 bytes of the base table name (if the name is shorter than 3, *yyy* is padded with #). *nnnn* is a numeric string that will start at 0000 and be incremented by 1 until a unique number is found.

Implicitly created XML table names will be *Xyyyyyyyyyyyyyyyyynnn*, where *yyyyyyyyyyyyyyyyyy* is the first 18 UTF-8 bytes of the base table name or of the entire name if it is less than 18. *nnn* will only be appended if the name already exists in the table. If the name already exists, *nnn* will be replaced with 000 and will be incremented by 1 until the name is unique.

Implicitly created document ID index names will be *I_DOCIDyyyyyyyyyyyyynnn*, where *yyyyyyyyyyyyyyyyyy* is the first 18 UTF-8 bytes of the base table name or the entire name if it is less than 18. *nnn* will only be appended if the index already exists in the table. If the index already exists, *nnn* will be replaced with 000 and will be incremented by 1 until the name is unique.

Implicitly created node ID index names will be *I_NODEIDyyyyyyyyyyyyynnn*, where *yyyyyyyyyyyyyyyyyy* is the first 18 UTF-8 bytes of the XML table name or the entire name if it is less than 18. *nnn* will only be appended if the index already exists in the table. If the index already exists, *nnn* will be replaced with 000 and will be incremented by 1 until the name is unique.

Creating a table with an identity column: When a table has an identity column, DB2 can automatically generate sequential numeric values for the column as rows are inserted into the table. Thus, identity columns are ideal for primary keys. Identity columns and ROWID columns are similar in that both types of columns contain values that DB2 generates. ROWID columns are used in large object (LOB) table spaces and can be useful in direct-row access. ROWID columns contain values of the ROWID data type, which returns a 40 byte VARCHAR value that is not regularly ascending or descending. ROWID data values are therefore not well suited to many application uses, such as generating employee numbers or product numbers. For data that is not LOB data and that does not require direct-row access, identity columns are usually a better approach, because identity columns contain existing numeric data types and can be used in a wide variety of uses for which ROWID values would not be suitable.

When a table is recovered to a point-in-time, it is possible that a large gap in the generated values for the identity column might result. For example, assume a table

has an identity column that has an incremental value of 1 and that the last generated value at time T1 was 100 and DB2 subsequently generates values up to 1000. Now, assume that the table space is recovered back to time T1. The generated value of the identity column for the next row that is inserted after the recovery completes will be 1001, leaving a gap from 100 to 1001 in the values of the identity column.

If you want to ensure that an identity column has unique values, create a unique index on the column.

Creating a table with a LONG VARCHAR or LONG VARGRAPHIC column:

Although the syntax LONG VARCHAR and LONG VARGRAPHIC is allowed for compatibility with previous releases of DB2, its use is not encouraged. VARCHAR(*integer*) and VARGRAPHIC(*integer*) is the recommended syntax, because after the CREATE TABLE statement is processed, DB2 considers a LONG VARCHAR column to be VARCHAR and a LONG VARGRAPHIC column to be VARGRAPHIC.

When a column is defined using the LONG VARCHAR or LONG VARGRAPHIC syntax, DB2 determines the length attribute of the column. You can use the following information, which is provided for existing applications that require the use of the LONG VARCHAR or LONG VARGRAPHIC syntax, to calculate the byte count and the character count of the column.

To calculate the byte count, use this formula:

$$2 * (\text{INTEGER}((\text{INTEGER}((m - i - k) / j)) / 2))$$

Where:

- m* Is the maximum row size (8 less than the maximum record size)
- i* Is the sum of the byte counts of all columns in the table that are not LONG VARCHAR or LONG VARGRAPHIC
- j* is the number of LONG VARCHAR and LONG VARGRAPHIC columns in the table
- k* k is the number of LONG VARCHAR and LONG VARGRAPHIC columns that allow nulls

To find the character count:

1. Find the byte count.
2. Subtract 2.
3. If the data type is LONG VARGRAPHIC, divide the result by 2. If the result is not an integer, drop the fractional part.

If the IN DATABASE clause is specified: If you specify IN DATABASE (either explicitly or by default), but do not specify a table space, a table space is implicitly created in *database-name*. The name of the table space is derived from the table name. The qualifier of the table space is the same as the qualifier of the table.

If range-partitioning is not specified, the implicitly created table space is a partition-by-growth table space with MAXPARTITIONS 256, SEGSIZE 4, and DSSIZE 4G.

If range-partitioning is specified, the table space will be segmented with a SEGSIZE of 4, LOCKSIZE ROW, and LOCKMAX SYSTEM.

If the IN clause is not specified: If you do not specify the IN clause, DB2 will implicitly create a table space as described previously, but DB2 will also choose a database as follows:

- DB2 chooses a name of the form DSNnnnnnn, where nnnnnn is between 00001 and 10000, inclusive.
 - If DSNnnnnnn already exists and is an implicitly created database, DB2 creates the table in that database.
 - If DSNnnnnnn does not exist, DB2 creates a database with the name DSNnnnnnn.
- If DSNnnnnnn cannot be created because of a deadlock, timeout, or resource unavailable condition, DB2 increments nnnnnn by one and tries the resultant database name. If DB2 reaches 10000, and DSN10000 is not available, DB2 sets nnnnnn to 00001 and tries the resultant database name. If DB2 tries 10000 database names without success, an error occurs.

Implicitly created table spaces: If the table space is implicitly created, all of the following required system objects will also be implicitly created:

- The enforcing primary key index
- The enforcing unique key index
- Any necessary LOB table spaces, auxiliary table spaces, and auxiliary indexes
- The ROWID index (if the ROWID column is defined as GENERATED BY DEFAULT)

Implicitly created indexes: When the PRIMARY KEY or UNIQUE clause is used in the CREATE TABLE statement and the CREATE TABLE statement is processed by the schema processor or the table space that contains the table is implicitly created, DB2 implicitly creates the unique indexes used to enforce the uniqueness of the primary or unique keys.

When a ROWID column is defined as GENERATED BY DEFAULT in the CREATE TABLE statement, and the CREATE TABLE statement is processed by SET CURRENT RULES = 'STD' or the table space that contains the table is implicitly created, DB2 implicitly creates the unique indexes used to enforce the uniqueness of the ROWID column.

The privilege set must include the USE privilege of the buffer pool.

Each index is created as if the following CREATE INDEX statement were issued:
CREATE UNIQUE INDEX xxx ON table-name (column1,...)

Where:

- xxx is the name of the index that DB2 generates.
- table-name is the name of the table specified in the CREATE TABLE statement.
- (column1,...) is the list of column names that were specified in the UNIQUE or PRIMARY KEY clause of the CREATE TABLE statement, or the column is a ROWID column that is defined as GENERATED BY DEFAULT.

For more information about the schema processor, see *DB2 Administration Guide*.

In addition, if the table space that contains the table is implicitly created, DB2 will check the DEFINE DATA SET subsystem parameter to determine whether to define the underlying data set for the index space of the implicitly created index on the base table.

If DEFINE DATA SET is NO, the index is created as if the following CREATE INDEX statement is issued:

```
CREATE UNIQUE INDEX xxx ON table-name (column1,...) DEFINE NO
```

Maximum record size: The maximum record size of a table depends on the page size of the table space and whether the EDITPROC clause is specified, as shown in Table 98. The initial page size of the table space is the size of its buffer, which is determined by the BUFFERPOOL clause that was explicitly or implicitly specified when the table space was created. When the record size reaches 90 percent of the maximum record size for the page size of the table space, the next largest page size is automatically used.

Table 98. Maximum record size, in bytes

EDITPROC	Page Size = 4KB	Page Size = 8KB	Page Size = 16KB	Page Size = 32KB
NO	4056	8138	16330	32714
YES	4046	8128	16320	32704

The maximum record size corresponds to the maximum length of a VARCHAR column if that column is the only column in the table.

If the table space that contains the table is implicitly created, the proper buffer pool size is chosen according to the actual record size. If the record size reaches 90% of the maximum record size for the page size of the table space, the next largest page size will be used. Table 99 shows what 90% of the maximum record size:

Table 99. 90% of Maximum record size, in bytes

Page Size = 4KB	Page Size = 8KB	Page Size = 16KB	Page Size = 32KB
3640	7310	14680	29432

Byte counts: The sum of the byte counts of the columns must not exceed the maximum row size of the table. The maximum row size is eight less than the maximum record size.

For columns that do not allow null values, Table 100 gives the byte counts of columns by data type. For columns that allow null values, the byte count is one more than shown in the table.

Table 100. Byte counts of columns by data type

Data Type	Byte Count
INTEGER	4
SMALLINT	2
BIGINT	8

Table 100. Byte counts of columns by data type (continued)

Data Type	Byte Count
FLOAT(<i>n</i>)	If <i>n</i> is between 1 and 21, the byte count is 4. If <i>n</i> is between 22 and 53, the byte count is 8.
DECIMAL	INTEGER(<i>p</i> /2)+1, where <i>p</i> is the precision
DECFLOAT(16)	9
DECFLOAT(34)	17
CHAR(<i>n</i>)	<i>n</i>
VARCHAR(<i>n</i>)	<i>n</i> +2
CLOB	6
GRAPHIC(<i>n</i>)	2 <i>n</i>
VARGRAPHIC(<i>n</i>)	2 <i>n</i> +2
DBCLOB	6
BINARY(<i>n</i>)	<i>n</i>
VARBINARY(<i>n</i>)	<i>n</i> +2
BLOB	6
DATE	4
TIME	3
TIMESTAMP	10
ROWID	19
distinct type	The length of the source data type upon which the distinct type was based

Creating a materialized query table: If the fullselect in the CREATE TABLE statement contains a SELECT *, the select list of the subselect is determined at the time the materialized query table is created. In addition, any references to user-defined functions are resolved at the same time. The isolation level at the time when the CREATE TABLE statement is executed is the isolation level for the materialized query table. After a materialized query table is created, the REFRESH_TIME column of the row for the table in the SYSIBM.SYSVIEWS catalog table contains the default timestamp.

The owner of a materialized query table has all the table privileges with the grant option on the table irrespective of whether the owner has the necessary privileges on the base tables, views, functions, and sequences.

No unique constraints or unique indexes can be created for materialized query tables. Thus, a materialized query table cannot be a parent table in a referential constraint.

When you are creating user-maintained materialized query tables, you should create the materialized query table with query optimization disabled and then enable the table for query optimization after it is populated. Otherwise, DB2 might rewrite queries to use the empty materialized query table, and you will not get accurate results.

Considerations for implicitly hidden columns: A column that is defined as implicitly hidden is not part of the result table of a query that specifies * in a SELECT list. However, an implicitly hidden column can be explicitly referenced in

a query. For example, an implicitly hidden column can be referenced in the SELECT list or in a predicate in a query. Additionally, an implicitly hidden column can be explicitly referenced in a COMMENT, CREATE INDEX statement, ALTER TABLE statement, INSERT statement, MERGE statement, UPDATE statement, or RENAME statement. An implicitly hidden column can be referenced in a referential constraint. A REFERENCES clause that does not contain a column list refers implicitly to the primary key of the parent table. It is possible that the primary key of the parent table includes a column defined as implicitly hidden. Such a referential constraint is allowed.

If the SELECT list of the fullselect of a materialized query definition explicitly refers to an implicitly hidden column, that column will be part of the materialized query table.

If the SELECT list of the full select of a view definition (CREATE VIEW statement) explicitly refers to an implicitly hidden column, that column will be part of the view, however the view column is not considered 'hidden'.

Restrictions on field procedures, edit procedures, and validation exit procedures: Field procedures, edit procedures, and validation exit procedures cannot be used on tables that have column names that are larger than 18 EBCDIC bytes. If you have tables that have field procedures or validation exit procedures and you add a column where the column name is larger than 18 bytes, the field procedures and validation exit procedures for the table will be invalidated.

Consider using triggers to replace the functionality on field procedures, edit procedures, and validation exit procedures on tables where the column names are larger than 18 EBCDIC bytes.

Restrictions on SQL data change statements in the same unit of work as CREATE TABLE: SQL data change statements cannot follow, in the same unit of work, CREATE TABLE statements that specify the PARTITION BY clause.

Creating a table while a utility runs: You cannot use CREATE TABLE while a DB2 utility has control of the table space implicitly or explicitly specified by the IN clause.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following clauses:

- NOCACHE (single clause) as a synonym for NO CACHE
- NOCYCLE (single clause) as a synonym for NO CYCLE
- NOMINVALUE (single clause) as a synonym for NO MINVALUE
- NOMAXVALUE (single clause) as a synonym for NO MAXVALUE
- NOORDER (single clause) as a synonym for NO ORDER
- PART *integer* VALUES can be specified as an alternative to PARTITION *integer* ENDING AT.
- VALUES as a synonym for ENDING AT
- DEFINITION ONLY as a synonym for WITH NO DATA
- SUMMARY between CREATE and TABLE

Examples

Example 1: Create a table named DSN8910.DEPT in the table space DSN8S91D of the database DSN8D91A. Name the five columns DEPTNO, DEPTNAME,

MGRNO, ADMRDEPT, and LOCATION, allowing only MGRNO and LOCATION to contain nulls, and designating DEPTNO as the only column in the primary key. All five columns hold character string data. Assuming a value of NO for the field MIXED DATA on installation panel DSNTIPF, all five columns have the subtype SBCS.

```
CREATE TABLE DSN8910.DEPT
(DEPTNO CHAR(3) NOT NULL,
DEPTNAME VARCHAR(36) NOT NULL,
MGRNO CHAR(6) ,
ADMRDEPT CHAR(3) NOT NULL,
LOCATION CHAR(16) ,
PRIMARY KEY(DEPTNO)
)
IN DSN8D91A.DSN8S91D;
```

Example 2: Create a table named DSN8910.PROJ in an implicitly created table space of the database DSN8D91A. Assign the table a validation procedure named DSN8EAPR.

```
CREATE TABLE DSN8910.PROJ
(PROJNO CHAR(6) NOT NULL,
PROJNAME VARCHAR(24) NOT NULL,
DEPTNO CHAR(3) NOT NULL,
RESPEMP CHAR(6) NOT NULL,
PRSTAFF DECIMAL(5,2) ,
PRSTDATE DATE ,
PRENDATE DATE ,
MAJPROJ CHAR(6) NOT NULL)
IN DATABASE DSN8D91A
VALIDPROC DSN8EAPR;
```

Example 3: Assume that table PROJECT has a non-primary unique key that consists of columns DEPTNO and RESPEMP (the department number and employee responsible for a project). Create a project activity table named ACTIVITY with a foreign key on that unique key.

```
CREATE TABLE ACTIVITY
(PROJNO CHAR(6) NOT NULL,
ACTNO SMALLINT NOT NULL,
ACTDEPT CHAR(3) NOT NULL,
ACTOWNER CHAR(6) NOT NULL,
ACSTAFF DECIMAL(5,2) ,
ACSTDATE DATE NOT NULL,
ACENDATE DATE ,
FOREIGN KEY (ACTDEPT,ACTOWNER)
REFERENCES PROJECT (DEPTNO,RESPEMP) ON DELETE RESTRICT)
IN DSN8D91A.DSN8S91D;
```

Example 4: Create an employee photo and resume table EMP_PHOTO_RESUME that complements the sample employee table. The table contains a photo and resume for each employee. Put the table in table space DSN8D91A.DSN8S91E. Let DB2 always generate the values for the ROWID column.

```
CREATE TABLE DSN8910.EMP_PHOTO_RESUME
(EMPNO CHAR(6) NOT NULL,
EMP_ROWID ROWID NOT NULL GENERATED ALWAYS,
EMP_PHOTO BLOB(110K),
RESUME CLOB(5K),
PRIMARY KEY (EMPNO))
IN DSN8D91A.DSN8S91E
CCSID EBCDIC;
```

Example 5: Create an EMPLOYEE table with an identity column named EMPNO. Define the identity column so that DB2 will always generate the values for the

column. Use the default value, which is 1, for the first value that should be assigned and for the incremental difference between the subsequently generated consecutive numbers.

```
CREATE TABLE EMPLOYEE
  (EMPNO      INTEGER GENERATED ALWAYS AS IDENTITY,
   ID         SMALLINT,
   NAME       CHAR(30),
   SALARY     DECIMAL(5,2),
   DEPTNO     SMALLINT)
IN DSN8D91A.DSN8S91D;
```

Example 6: Assume a very large transaction table named TRANS contains one row for each transaction processed by a company. The table is defined with many columns. Create a materialized query table for the TRANS table that contain daily summary data for the date and amount of a transaction.

```
CREATE TABLE STRANS AS
  (SELECT YEAR AS SYEAR, MONTH AS SMONTH, DAY AS SDAY, SUM(AMOUNT) AS SSUM
   FROM TRANS
   GROUP BY YEAR, MONTH, DAY)
DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

Example 7: The following example creates a table in a partition-by-growth table space and includes the APPEND option:

```
CREATE TABLE TS01TB
  (C1 SMALLINT,
   C2 DECIMAL(9,2),
   C3 CHAR(4))
APPEND YES
IN TS01DB.TS01TS;
```

Example 8: The following example creates a table in a partition-by-growth table space where the table space is implicitly created:

```
CREATE TABLE TS02TB
  (C1 SMALLINT,
   C2 DECIMAL(9,2),
   C3 CHAR(4))
PARTITION BY SIZE EVERY 4G
IN DATABASE DSND804;
```

Example 9: Create a table, EMP_INFO, that contains a phone number and address for each employee. Include a row change timestamp column in the table to track the modification of employee information:

```
CREATE TABLE EMP_INFO
  (EMPNO CHAR(6) NOT NULL,
   EMP_INFOCHANGE NOT NULL
   GENERATED ALWAYS FOR EACH ROW ON UPDATE
   AS ROW CHANGE TIMESTAMP,
   EMP_ADDRESS VARCHAR(300),
   EMP_PHONENO CHAR(4),
   PRIMARY KEY (EMPNO));
```

Example 10: Create a table, TB01, that uses a range partitioning scheme with a segment size of 4 and 4 partitions.

```
CREATE TABLE TB01 (
  ACCT_NUM      INTEGER,
  CUST_LAST_NM  CHAR(15),
  LAST_ACTIVITY_DT VARCHAR(25),
  COL2          CHAR(10),
  COL3          CHAR(25),
  COL4          CHAR(25),
  COL5          CHAR(25),
```



```
|          COL6          CHAR(55),  
|          STATE          CHAR(55))  
| IN DBB.TS01  
|  
| PARTITION BY (ACCT_NUM)  
| (PARTITION 1 ENDING AT (199),  
| PARTITION 2 ENDING AT (299),  
| PARTITION 3 ENDING AT (399),  
| PARTITION 4 ENDING AT (MAXVALUE));  
|  
|
```

CREATE TABLESPACE

The CREATE TABLESPACE statement defines a segmented, partitioned, or universal table space at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATETS privilege for the database
- DBADM, DBCTRL, or DBMAINT authority for the database
- SYSADM or SYSCTRL authority

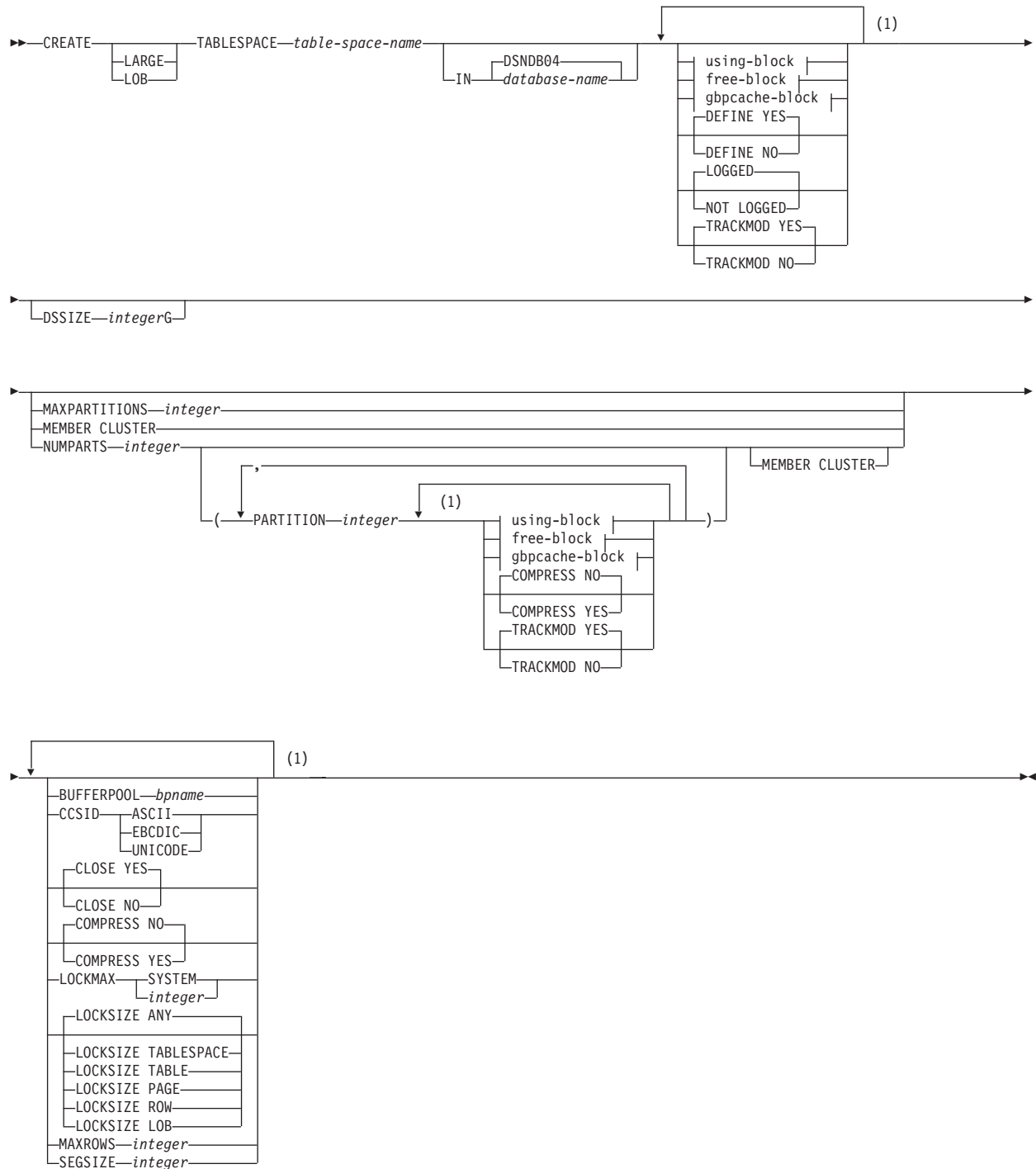
If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Additional privileges might be required, as explained in the description of the BUFFERPOOL and USING STOGROUP clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

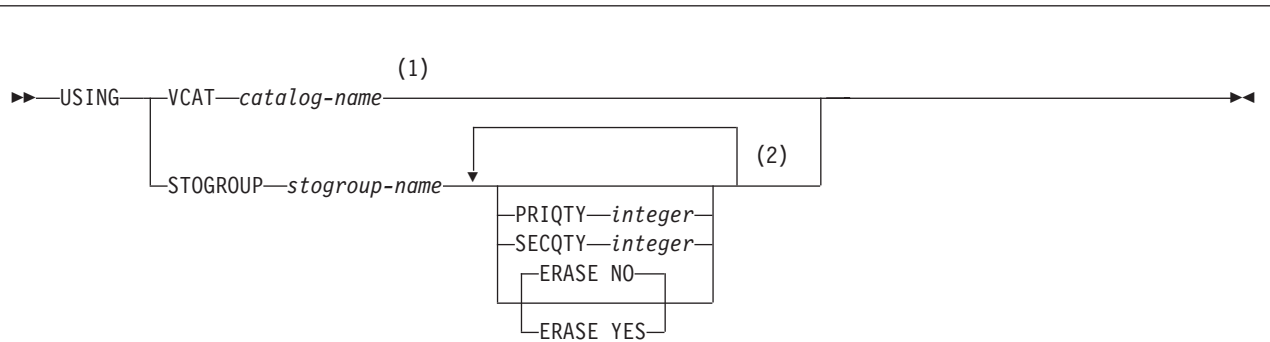
Syntax



Notes:

- 1 The same clause must not be specified more than once.

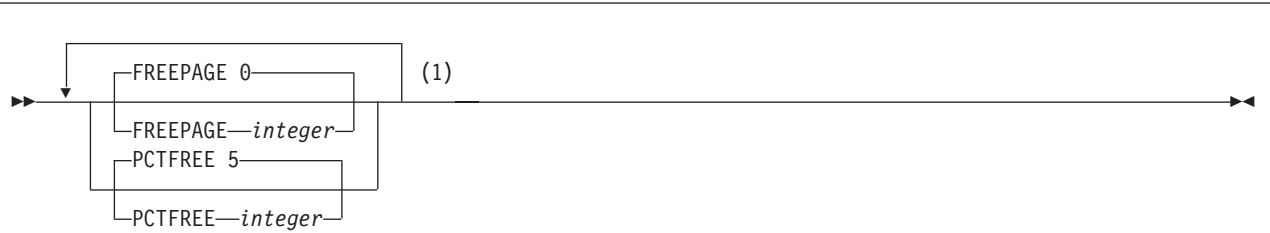
using-block:



Notes:

- 1 USING VCAT must not be specified if MAXPARTITIONS is also specified.
- 2 The same clause must not be specified more than once.

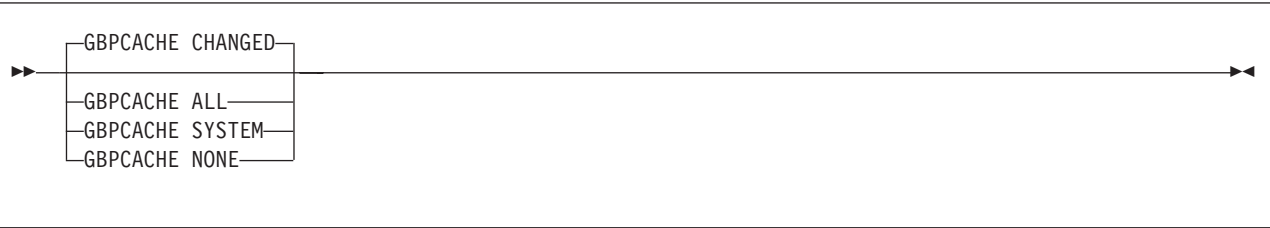
free-block:



Notes:

- 1 The same clause must not be specified more than once.

gbpcache-block:



Description

LARGE

Identifies that each partition of a partitioned table space has a maximum partition size of 4 GB, which enables the table space to contain more than 64 GB of data. The preferred method to specify a maximum partition size of 4 GB and larger is the DSSIZE clause. The LARGE clause is for compatibility of releases of DB2 for z/OS prior to Version 6. Do not specify LARGE if LOB or DSSIZE is specified. Do not specify LARGE for a table space in a work file database.

LOB

Identifies the table space as LOB table space. A LOB table space is used to hold LOB values.

The LOB table space must be in the same database as its associated base table space. Do not specify LOB for a table space in a work file database.

table-space-name

Names the table space. The name, qualified with the *database-name* implicitly or explicitly specified by the IN clause, must not identify a table space, index space, or LOB table space that exists at the current server.

A table space that is for declared temporary tables must be in the work file database. PUBLIC implicitly receives the USE privilege (without GRANT authority) on any table space created in the work file database. This implicit privilege is not recorded in the DB2 catalog, and it cannot be revoked.

IN *database-name*

Specifies the database in which the table space is created. *database-name* must identify a database that exists at the current server and must not specify the following:

- DSNDB06 for any type of table space
- A work file database for a LOB table space
- A TEMP database
- An implicitly created database

If the table space is for declared temporary tables or static scrollable cursors, the name of the work file database must be specified.

DSNDB04 is the default.

The components of the USING clause are discussed below, first for nonpartitioned table spaces and then for partitioned table spaces. If you omit USING, the default storage group of the database must exist.

USING clause for nonpartitioned table spaces:

For nonpartitioned table spaces, the USING clause indicates whether the data set for the table space is defined by you or by DB2. If DB2 is to define the data set, the clause also gives space allocation parameters and an erase rule.

If you omit USING, DB2 defines the data sets using the default storage group of the database and the defaults for PRIQTY, SECQTY, and ERASE.

VCAT *catalog-name*

Specifies that the first data set for the table space is managed by the user, and following data sets, if needed, are also managed by the user.

The data sets defined for the table space are linear VSAM data sets cataloged in an integrated catalog facility catalog identified by *catalog-name*. An alias³² must be used if the catalog name is longer than eight characters.

Conventions for table space data set names are given in *DB2 Administration Guide*. *catalog-name* is the first qualifier for each data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

32. The alias of an integrated catalog facility catalog.

VCAT must not be specified if MAXPARTITIONS is also specified.

STOGROUP *stogroup-name*

Specifies that DB2 will define and manage the data sets for the table space. Each data set will be defined on a volume of the identified storage group. The values specified (or the defaults) for PRIQTY and SECQTY determine the primary and secondary allocations for the data set. The storage group supplies the name of a volume for the data set and the first-level qualifier for the data set name. The first-level qualifier is also the name of, or an alias³² for, the integrated catalog facility catalog on which the data set is to be cataloged. The naming conventions for the data set are the same as if the data set is managed by the user. As was mentioned above for VCAT, the first-level qualifier could cause naming conflicts if the local DB2 can share integrated catalog facility catalogs with other DB2 subsystems.

stogroup-name must identify a storage group that exists at the current server. SYSADM or SYSCTRL authority, or the USE privilege on the storage group, is required.

The description of the storage group must include at least one volume serial number, or it must indicate that the choice of volumes is left to Storage Management Subsystem (SMS). If volume serial numbers appear in the description, each must identify a volume that is accessible to z/OS for dynamic allocation of the data set, and all identified volumes must be of the same device type.

The integrated catalog facility catalog used for the storage group must *not* contain an entry for the first data set of the table space. If the integrated catalog facility catalog is password protected, the description of the storage group must include a valid password.

PRIQTY *integer*

Specifies the minimum primary space allocation for a DB2-managed data set. When you specify PRIQTY with a value other than -1, the primary space allocation is at least *n* kilobytes, where *n* is the value of *integer* with the following exceptions:

- For 4KB page sizes, if *integer* is less than 12, *n* is 12.
- For 8KB page sizes, if *integer* is less than 24, *n* is 24.
- For 16KB page sizes, if *integer* is less than 48, *n* is 48.
- For 32KB page sizes, if *integer* is less than 96, *n* is 96.
- For any page size, if *integer* is greater than 4194304, *n* is 4194304.

For LOB table spaces, the exceptions are:

- For 4KB page sizes, if *integer* is less than 200, *n* is 200.
- For 8KB page sizes, if *integer* is less than 400, *n* is 400.
- For 16KB page sizes, if *integer* is less than 800, *n* is 800.
- For 32KB page sizes, if *integer* is less than 1600, *n* is 1600.
- For any page size, if *integer* is greater than 4194304, *n* is 4194304.

If the DB2 subsystem is under DFP 1.5, the maximum value allowed for PRIQTY is 64GB (67108864 kilobytes). Otherwise, the existing maximum value of 4GB (4194304 kilobytes) applies.

If you do not specify PRIQTY or specify PRIQTY -1, DB2 uses a default value for the primary space allocation; for information on how DB2 determines the default value, see Rules for primary and secondary space allocation.

If you specify PRIQTY and do not specify a value of -1, DB2 specifies the primary space allocation to access method services using the smallest multiple of p KB not less than n , where p is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. The amount of storage space requested must be available on some volume in the storage group based on VSAM space allocation restrictions. Otherwise, the primary space allocation will fail. To more closely estimate the actual amount of storage, see the description of the DEFINE CLUSTER command in *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

Executing this statement causes only one data set to be created. However, you might have more data than this one data set can hold. DB2 automatically defines more data sets when they are needed. Regardless of the value in PRIQTY, when a data set reaches its maximum size, DB2 creates a new one. To enable a data set to reach its maximum size without running out of extents, it is recommended that you allow DB2 to automatically choose the value of the secondary space allocations for extents.

If you do choose to explicitly specify SECQTY, to avoid wasting space, use the following formula to make sure that PRIQTY and its associated secondary extent values do not exceed the maximum size of the data set:

$$\text{PRIQTY} + (\text{number of extents} * \text{SECQTY}) \leq \text{DSSIZE (implicit or explicit)}$$

SECQTY integer

Specifies the minimum secondary space allocation for a DB2-managed data set. If you do not specify SECQTY, DB2 uses a formula to determine a value. For information on the actual value that is used for secondary space allocation, whether you specify a value or not, see Rules for primary and secondary space allocation.

If you specify SECQTY and do not specify a value of -1, DB2 specifies the secondary space allocation to access method services using the smallest multiple of p KB not less than n , where p is the page size of the table space. The allocated space can be greater than the amount of space requested by DB2. For example, it could be the smallest number of tracks that will accommodate the request. To more closely estimate the actual amount of storage, see the description of the DEFINE CLUSTER command in .

ERASE

Indicates whether the DB2-managed data sets for the table space or partition are to be erased when they are deleted during the execution of a utility or an SQL statement that drops the table space.

NO Does not erase the data sets. Operations involving data set deletion will perform better than ERASE YES. However, the data is still accessible, though not through DB2. This is the default.

YES

Erases the data sets. As a security measure, DB2 overwrites all data in the data sets with zeros before they are deleted.

USING clause for partitioned table spaces:

If the table space is partitioned, there is a USING clause for each partition;

either one you give explicitly or one provided by default. Except as explained below, the meaning of the clause and the rules that apply to it are the same as for a nonpartitioned table space.

The USING clause for a particular partition is the first of these choices that can be found:

- A USING clause in the PARTITION clause for the partition
- A USING clause that is not in any PARTITION clause
- An implicit USING STOGROUP clause that identifies the default storage group of the database and accepts the defaults for PRIQTY, SECQTY, and ERASE

VCAT *catalog-name*

Indicates that the data set for the partition is managed by the user using the naming conventions set forth in *DB2 Administration Guide*. As was true for the nonpartitioned case, *catalog-name* identifies the catalog for the data set and supplies the first-level qualifier for the data set name.

One or more DB2 subsystems could share integrated catalog facility catalogs with the current server. To avoid the chance of having one of those subsystems attempt to assign the same name to different data sets, select a value for *catalog-name* that is not used by the other DB2 subsystems.

DB2 assumes one and only one data set for each partition.

STOGROUP *stogroup-name*

Indicates that DB2 will create a data set for the partition with the aid of a storage group named *stogroup-name*. The data set is defined during the execution of this statement. DB2 assumes one and only one data set for each partition.

The *stogroup-name* must identify a storage group that exists at the current server and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the storage group. The integrated catalog facility catalog used for the storage group must *not* contain an entry for that data set.

When USING STOGROUP is specified for a partition, the defaults for PRIQTY, SECQTY, and ERASE are the values specified in the USING STOGROUP clause that is not in any PARTITION clause. If that USING STOGROUP clause is not specified, the defaults are those specified in the description of PRIQTY, SECQTY, and ERASE.

FREEPAGE *integer*

Specifies how often to leave a page of free space when the table space or partition is loaded or reorganized. You must specify an integer in the range 0 to 255. If you specify 0, no pages are left as free space. Otherwise, one free page is left after every *n* pages, where *n* is the specified integer. However, if the table space is segmented and the integer you specify is not less than the segment size, *n* is one less than the segment size.

If the table space is segmented, the number of pages left free must be less than the SEGSIZE value. If the number of pages to be left free is greater than or equal to the SEGSIZE value, then the number of pages is adjusted downward to one less than the SEGSIZE value.

The default is FREEPAGE 0, leaving no free pages. Do not specify FREEPAGE for a LOB table space or a table space in a work file database.

For XML table spaces, this change has no effect until data in the table space is reorganized.

PCTFREE *integer*

Indicates what percentage of each page to leave as free space when the table is loaded or reorganized. *integer* can range from 0 to 99. The first record on each page is loaded without restriction. When additional records are loaded, at least *integer* percent of free space is left on each page.

The default is PCTFREE 5. Do not specify PCTFREE for a LOB table space or a table space in a work file database.

For XML table spaces, this change has no effect until data in the table space is reorganized.

If the table space is partitioned, the values of FREEPAGE and PCTFREE for a particular partition are given by the first of these choices that apply:

- The values of FREEPAGE and PCTFREE given in the PARTITION clause for that partition
- The values given in a *free-block* that is not in any PARTITION clause
- The default values are FREEPAGE 0 and PCTFREE 5.

GBPCACHE

In a data sharing environment, specifies what pages of the table space or partition are written to the group buffer pool in a data sharing environment. In a non-data-sharing environment, you can specify GBPCACHE for a table space other than one in a work file database, but it is ignored. Do not specify GBPCACHE for a table space in a work file database in either environment (data sharing or non-data-sharing).

CHANGED

When there is inter-DB2 R/W interest on the table space or partition, updated pages are written to the group buffer pool. When there is no inter-DB2 R/W interest, the group buffer pool is not used. Inter-DB2 R/W interest exists when more than one member in the data sharing group has the table space or partition open, and at least one member has it open for update. GBPCACHE CHANGED is the default.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), CHANGED is ignored and no pages are cached to the group buffer pool.

ALL

Indicates that pages are to be cached in the group buffer pool as they are read in from DASD.

Exception: In the case of a single updating DB2 when no other DB2s have any interest in the page set, no pages are cached in the group buffer pool.

If the table space is in a group buffer pool that is defined to be used only for cross-invalidation (GBPCACHE NO), ALL is ignored and no pages are cached to the group buffer pool.

SYSTEM

Indicates that only changed system pages within the LOB table space are to be cached to the group buffer pool. A system page is a space map page or any other page that does not contain actual data values.

This is the default for LOB table spaces. You can use SYSTEM only for a LOB table space.

NONE

Indicates that no pages are to be cached to the group buffer pool. DB2 uses the group buffer pool only for cross-invalidation.

If you specify NONE, the table space or partition must not be in recover pending status and must be in the stopped state when the CREATE TABLESPACE statement is executed.

If the table space is partitioned, the value of GBPCACHE for a particular partition is given by the first of these choices that applies:

1. The value of GBPCACHE given in the PARTITION clause for that partition. Do not use more than one *gbpcache-block* in any PARTITION clause.
2. The value given in a *gbpcache-block* that is not in any PARTITION clause.
3. The default value CHANGED.

DEFINE

Specifies when the underlying data sets for the table space are physically created.

YES

The data sets are created when the table space is created (the CREATE TABLESPACE statement is executed). YES is the default.

If MAXPARTITIONS is also specified, only the first partition is created when the table space is created. Additional partitions are created as needed.

NO The data sets are not created until data is inserted into the table space. DEFINE NO is applicable only for DB2-managed data sets (USING STOGROUP is specified). DEFINE NO is ignored for user-managed data sets (USING VCAT is specified). DB2 uses the SPACE column in catalog table SYSTABLEPART to record the status of the data sets (undefined or allocated). DEFINE NO is also ignored for a LOB table space.

Do not specify DEFINE NO for a table space in a work file database. DEFINE NO is not recommended if you intend to use any tools outside of DB2 to manipulate data, such as to load data, because data sets might then exist when DB2 does not expect them to exist. When DB2 encounters this inconsistent state, applications will receive an error.

LOGGED or NOT LOGGED

Specifies whether changes that are made to the data in the specified table space are recorded in the log.

LOGGED

Specifies that changes that are made to the data in the specified table space are recorded in the log. This applies to all tables that are created in the specified table space and to all indexes of those tables. XML table spaces and their indexes inherit the logging attribute from the associated base table space. Auxiliary indexes also inherit the logging attribute from the associated base table space.

LOGGED cannot be specified for table spaces in DSNDB06 (the DB2 catalog) or in a work file database.

LOGGED is the default.

NOT LOGGED

Specifies that changes that are made to data in the specified table space are not recorded in the log. This parameter applies to all tables that are created in the specified table space and to all indexes of those tables. XML table

spaces and their indexes inherit the logging attribute from the associated base table space. Auxiliary indexes inherit the logging attribute from the associated base table space.

NOT LOGGED prevents undo and redo information from being recorded in the log; however, control information for the specified table space will continue to be recorded in the log.

NOT LOGGED cannot be specified for table spaces in the following databases:

- DSNDB06 (the DB2 catalog)
- a work file database

TRACKMOD

Specifies whether DB2 tracks modified pages in the space map pages of the table space or partition. Do not specify TRACKMOD for a LOB table space. Do not specify TRACKMOD for a table space in a work file database.

YES

DB2 tracks changed pages in the space map pages to improve the performance of incremental image copy.

NO DB2 does not track changed pages in the space map pages. It uses the LRSN value in each page to determine whether a page has been changed.

If the table space is partitioned, the value of TRACKMOD for a particular partition is given by the first of these choices that applies:

1. The value of TRACKMOD given in the PARTITION clause for that partition.
2. The value given in a *trackmod-block* that is not in any PARTITION clause.
3. The default value YES.

DSSIZE integerG

Specifies the maximum size for each partition, or for LOB table spaces, each data set. If you specify DSSIZE, you must also specify the NUMPARTS, MAXPARTITIONS, or LOB clause.

The following values are valid:

1G	1 gigabyte
2G	2 gigabytes
4G	4 gigabytes
8G	8 gigabytes
16G	16 gigabytes
32G	32 gigabytes
64G	64 gigabytes

To specify a value greater than 4G, the following conditions must be true:

- DB2 is running with DFSMS Version 1 Release 5.
- The data sets for the table space are associated with a DFSMS data class that has been specified with extended format and extended addressability.

If NUMPARTS is also specified, the maximum size of each partition depends on the value of NUMPARTS, as shown in the following table. Otherwise, the maximum size of each partition is 4G.

Table 101. Maximum partition size depending on value of Numparts (assuming that LARGE is not specified)

Value of Numparts	Maximum partition size (default for DSSIZE)
1 to 16	4GB (4G)
17 to 32	2GB (2G)
33 to 64	1GB (1G)
65 to 256	4GB (4G)

If Numparts is greater than 256, the maximum partition size (and the default for DSSIZE) depends on the page size of the table space, as shown in the following table. If DSSIZE is explicitly specified, the maximum partition size can be 64GB when Numparts 34 is specified (i.e. if DSSIZE 64GB is specified, the maximum Numparts can be 256 for a 4K page size partitioned table). For a 8K page size table, the maximum number of Numparts can be 512 if DSSIZE is set to 64GB.

Table 102. Maximum partition size depending on page size (assuming that LARGE is not specified)

Page size	Maximum partition size (default for DSSIZE)
4K	4GB (4G)
8K	8GB (8G)
16K	16GB (16G)
32K	32GB (32G)

The partition size shown is not necessarily the actual number of bytes used or allocated for any one partition; it is the largest number that can be logically addressed. Each partition occupies one data set.

For LOB table spaces, if DSSIZE is not specified, the default for the maximum size of each data set is 4GB. The maximum number of data sets is 254.

MAXPARTITIONS *integer*

Specifies that the table space is a partition-by-growth table space. The data set for the first partition is allocated unless the DEFINE NO clause is specified for the partition. The data sets for additional partitions are not allocated until they are needed.

integer specifies the maximum number of partitions to which the table space can grow. *integer* must be in the range of 1 to 4096, depending on the corresponding value of the DSSIZE clause. The following table shows the maximum value for MAXPARTITIONS in relation to the page size or DSSIZE value for the table space.

Table 103. Maximum value for MAXPARTITIONS given the page size or DSSIZE value for the table space

DSSIZE value	4K page size	8K page size	16K page size	32K page size
1G - 4G (1 GB to 4 GB)	4096	4096	4096	4096
8G (8 GB)	2048	4096	4096	4096
16G (16 GB)	1024	2048	4096	4096

Table 103. Maximum value for MAXPARTITIONS given the page size or DSSIZE value for the table space (continued)

DSSIZE value	4K page size	8K page size	16K page size	32K page size
32G (32 GB)	512	1024	2048	4096
64G (64 GB)	256	512	1024	2048

MAXPARTITIONS should not be specified for a table space that is in a work file database.

MEMBER CLUSTER

Specifies that data inserted by an insert operation is not clustered by the implicit clustering index (the first index) or the explicit clustering index. Instead, DB2 chooses where to locate the data in the table space based on available space.

Do not specify MEMBER CLUSTER for segmented table spaces.

Do not specify MEMBER CLUSTER for a LOB table space, a universal table space, or a table space in a work file database.

NUMPARTS *integer*

Indicates that the table space is partitioned. If SEGSIZE is also specified, the table space is a range-partitioned universal table space.

integer is the number of partitions, and can range from 1 to 4096 inclusive. NUMPARTS must be specified if DSSIZE is specified and LOB is omitted, or if LARGE is specified.

The maximum size of each partition depends on the value that is specified for DSSIZE or LARGE. If DSSIZE or LARGE is not specified, the number of partitions that are specified determines the maximum size of each partition. For a summary of the values for the maximum size of each partition, see Table 101 on page 1138 and Table 102 on page 1138.

The maximum number of partitions that a table space can have depends on the page size and DSSIZE. The total table space size depends on how many partitions it has and DSSIZE. Page size affects table size because it affects how many partitions a table space can have. The following table shows the maximum number of partitions for DSSIZE and page size and total table space size for both EA-enabled (extended addressability) and non-EA-enabled data sets. (Specifying a DSSIZE greater than 4GB requires EA-enabled data sets.)

Table 104. Table space size given page size and partitions

Type of RID	Page size	DSSIZE	Maximum number of partitions	Total table space size
5-byte EA	4KB	1GB	4096	4TB
5-byte EA	4KB	2GB	4096	8TB
5-byte EA	4KB	4GB	4096	16TB
5-byte EA	4KB	8GB	2048	16TB
5-byte EA	4KB	16GB	1024	16TB
5-byte EA	4KB	32GB	512	16TB
5-byte EA	4KB	64GB	256	16TB
5-byte EA	8KB	1GB	4096	4TB
5-byte EA	8KB	2GB	4096	8TB

Table 104. Table space size given page size and partitions (continued)

Type of RID	Page size	DSSIZE	Maximum number of partitions	Total table space size
5-byte EA	8KB	4GB	4096	16TB
5-byte EA	8KB	8GB	4096	32TB
5-byte EA	8KB	16GB	2048	32TB
5-byte EA	8KB	32GB	1024	32TB
5-byte EA	8KB	64GB	512	32TB
5-byte EA	16KB	1GB	4096	4TB
5-byte EA	16KB	2GB	4096	8TB
5-byte EA	16KB	4GB	4096	16TB
5-byte EA	16KB	8GB	4096	32TB
5-byte EA	16KB	16GB	4096	64TB
5-byte EA	16KB	32GB	2048	64TB
5-byte EA	16KB	64GB	1024	64TB
5-byte EA	32KB	1GB	4096	4TB
5-byte EA	32KB	2GB	4096	8TB
5-byte EA	32KB	4GB	4096	16TB
5-byte EA	32KB	8GB	4096	32TB
5-byte EA	32KB	16GB	4096	64TB
5-byte EA	32KB	32GB	4096	128TB
5-byte EA	32KB	64GB	2048	128TB
5-byte (non-EA) LARGE	4KB	(4GB)	4096	16TB

If you omit NUMPARTS, the table space is segmented with a SEGSIZE of 4, LOCKSIZE ANY (unless it is explicitly specified), is not partitioned, and initially occupies one data set.

Do not specify NUMPARTS for a LOB table space, a table space in a work file database, or a partition-by-growth table space.

PARTITION *integer*

Specifies to which partition the following *using-block* or *free-block* applies. *integer* can range from 1 to the number of partitions given by NUMPARTS.

You can code the PARTITION clause (and any *using-block* or *free-block* that follows it) as many times as needed. If you use the same partition number more than once, only the last specification for that partition is used.

BUFFERPOOL *bpname*

Identifies the buffer pool to be used for the table space and determines the page size of the table space. For 4KB, 8KB, 16KB and 32KB page buffer pools, the page sizes are 4 KB, 8 KB, 16 KB, and 32 KB, respectively. The *bpname* must identify an activated buffer pool, and the privilege set must include SYSADM or SYSCTRL authority, or the USE privilege on the buffer pool. If the table space is to be created in a work file database, you cannot specify 8KB and 16KB buffer pools.

If you do not specify the BUFFERPOOL clause, the default buffer pool of the database is used unless the table space that is being created is a LOB table space. If you do not specify the BUFFERPOOL clause and the table space that is being created is a LOB table space, the default buffer pool is the buffer pool that is specified in the DEFAULT BUFFER POOL FOR USER LOB DATA field on installation panel DSNTIP1.

See “Naming conventions” on page 51 for more details about *bpname*. See *DB2 Command Reference* for a description of active and inactive buffer pools.

LOCKSIZE

Specifies the size of locks used within the table space and, in some cases, also the threshold at which lock escalation occurs. Do not use this clause for a table space in a work file database.

ANY

Specifies that DB2 can use any lock size.

In most cases, DB2 uses LOCKSIZE PAGE LOCKMAX SYSTEM for non-LOB table spaces and LOCKSIZE LOB LOCKMAX SYSTEM for LOB table spaces. However, when the number of locks acquired for the table space exceeds the maximum number of locks allowed for a table space (an installation parameter), the page or LOB locks are released and locking is set at the next higher level. If the table space is segmented, the next higher level is the table. If the table space is nonsegmented, the next higher level is the table space.

If the table space is implicitly created, DB2 uses LOCKSIZE ROW.

TABLESPACE

Specifies table space locks.

TABLE

Specifies table locks. Use TABLE only for a segmented table space. Do not use TABLE for a universal table space.

PAGE

Specifies page locks. Do not use PAGE for a LOB table space.

ROW

Specifies row locks. Do not use ROW for a LOB table space.

LOB

Specifies LOB locks. Use LOB only for a LOB table space.

LOCKMAX

Specifies the maximum number of page, row, or LOB locks an application process can hold simultaneously in the table space. If a program requests more than that number, locks are escalated. The page, row, or LOB locks are released and the intent lock on the table space or segmented table is promoted to S or X mode. If you specify LOCKMAX for a table space in a work file database, DB2 ignores the value because these types of locks are not used.

integer

Specifies the number of locks allowed before escalating, in the range 0 to 2 147 483 647.

Zero (0) indicates that the number of locks on the table or table space are not counted and escalation does not occur.

SYSTEM

Indicates that the value of LOCKS PER TABLE(SPACE), on installation

panel DSNTIPJ, specifies the maximum number of page, row, or LOB locks a program can hold simultaneously in the table or table space.

The following table summarizes the results of specifying a LOCKSIZE value while omitting LOCKMAX.

LOCKSIZE	Resultant LOCKMAX
ANY	SYSTEM
TABLESPACE, TABLE, PAGE, 0 ROW, or LOB	

If the lock size is TABLESPACE or TABLE, LOCKMAX must be omitted, or its operand must be 0.

For an application that uses Sysplex query parallelism, a lock count is maintained on each member.

CLOSE

When the limit on the number of open data sets is reached, specifies the priority in which data sets are closed.

YES

Eligible for closing before CLOSE NO data sets. This is the default unless the table space is in a work file database.

NO Eligible for closing after all eligible CLOSE YES data sets are closed.

For a table space in a work file database, DB2 uses CLOSE NO regardless of the value specified.

COMPRESS

Specifies whether data compression applies to the rows of the table space or partition. Do not specify COMPRESS for a LOB table space or a table space in a work file database.

For partitioned table spaces, the COMPRESS attribute for each partition is the value from the first of the following conditions that apply:

- The value specified in the COMPRESS clause in the PARTITION clause for the partition
- The value specified in the COMPRESS clause that is not in any PARTITION clause
- An implicit COMPRESS NO by default.

See *DB2 Performance Monitoring and Tuning Guide* for more information about data compression.

YES

Specifies data compression. The rows are not compressed until the LOAD or REORG utility is run on the table in the table space or partition. If COMPRESS YES is specified, the table space is created with basic row format.

NO Specifies no data compression for the table space or partition.

CCSID *encoding-scheme*

Specifies the encoding scheme for tables stored in the table space.

If you do not specify a CCSID when it is allowed, the default is the encoding scheme of the database in which the table space resides, except for table spaces

in database DSNDB04; for table spaces in DSNDB04, the default is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

ASCII Specifies that the data is to be encoded using ASCII CCSIDs. If the database in which the table space is to reside is already defined as ASCII, the ASCII CCSIDs associated with that database are used. Otherwise, the default ASCII CCSIDs of the server are used.

EBCDIC

Specifies that the data is to be encoded using EBCDIC CCSIDs.

UNICODE

Specifies that the data is to be encoded using the UNICODE CCSIDs of the server.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or Unicode data is used.

All data stored within a table space must use the same encoding scheme unless the table space is in a work file database.

Do not specify CCSID for a LOB table space or a table space in a work file database. The encoding scheme for a LOB table space is inherited from the base table space. A table space in a work file database does not have an associated encoding scheme because the table space can contain created and declared temporary tables with a mixture of encoding schemes.

MAXROWS *integer*

Specifies the maximum number of rows that DB2 will consider placing on each data page. The integer can range from 1 through 255. This value is considered for insert operations, LOAD, and REORG. For LOAD and REORG (which do not apply for a table space in the work file database), the PCTFREE specification is considered before MAXROWS; therefore, fewer rows might be stored than the value you specify for MAXROWS.

If you do not specify MAXROWS, the default number of rows is 255.

Do not use MAXROWS for a LOB table space or a table space in a work file database.

SEGSIZE *integer*

Specifies that the table space will be a segmented or a universal table space depending on the specification of the MAXPARTITIONS or NUMPARTS clauses. *integer* specifies the number of pages that are to be assigned to each segment of the table space. *integer* must be a multiple of 4 between 4 and 64 (inclusive). Do not specify SEGSIZE for a LOB table space. The type of table space is determined by the SEGSIZE, MAXPARTITIONS, and NUMPARTS clauses as shown in the following table:

Table 105. Type of table space depending on value of SEGSIZE, MAXPARTITIONS, and NUMPARTS clauses

SEGSIZE clause	MAXPARTITIONS clause	NUMPARTS clause	Type of table space
specified	specified	not specified	partition-by-growth table space
specified	not specified	not specified	segmented table space
specified	not specified	specified	range-partitioned universal table space

Table 105. Type of table space depending on value of SEGSIZE, MAXPARTITIONS, and Numparts clauses (continued)

SEGSIZE clause	MAXPARTITIONS clause	Numparts clause	Type of table space
not specified	specified	not specified	partition-by-growth table space with an implicit specification of SEGSIZE 4
not specified	not specified	not specified	segmented table space with an implicit specification of SEGSIZE 4
not specified	not specified	specified	partitioned table space

If you specify SEGSIZE with Numparts, the table space will be created as a range-partitioned universal table space. If SEGSIZE is specified with MAXPARTITIONS, the table space will be created as a partition-by-growth universal table space. See *DB2 Administration Guide* for information about universal table spaces.

Notes

Segmented table spaces: If neither LOB, Numparts, nor SEGSIZE are specified, the table space that is created is a segmented table space. See *DB2 Administration Guide* for a discussion of types of table spaces.

Universal table spaces: If Numparts and SEGSIZE are specified, the table space that is created is a range-partitioned universal table space. If MAXPARTITIONS or MAXPARTITIONS and SEGSIZE are specified, the table space that is created is a partition-by-growth universal table space. See *DB2 Administration Guide* for information about universal table spaces. If a range-partitioned universal table space contains an XML column, the corresponding XML table space will be range-partitioned universal as well.

Table spaces in a work file database: The following restrictions apply to table spaces created in a work file database:

- They can be created for another member only if both the executing DB2 subsystem and the other member can access the work file data sets. That is required whether the data sets are user-managed or in a DB2 storage group.
- They cannot use 8 KB or 16 KB page sizes. (The buffer pool in which you define the table space determines the page size. For example, a table space that is defined in a 4 KB buffer pool has 4 KB page sizes.)
- When you create a table space in a work file database, the following clauses are not allowed:

CCSID	LARGE	MAXPARTITIONS
COMPRESS	LOB	MEMBER CLUSTER
DEFINE NO	LOCKPART	NOT LOGGED
FREEPAGE	LOCKSIZE	Numparts
GBPCACHE	LOGGED	PCTFREE
	MAXROWS	TRACKMOD

Table spaces for declared temporary tables: Declared temporary tables and sensitive static scrollable cursors must reside in segmented table spaces in the work file database. At least one table space with a 32KB page size must exist in the work file database before a declared temporary table can be defined and used or before sensitive static scrollable cursors are opened.

Table spaces in the work file database are shared by work files, created and declared global temporary tables and sensitive static scrollable cursor result tables. You cannot specify which table space is to be used for any specific object.

When you create table spaces in the work file database, it is recommended that you give each table space the same segment size, with the same minimum primary and secondary space allocation values for the data sets, to maximize the use of all the table spaces for all objects in all application processes.

Creating LOB table spaces: When you create a LOB table space, the following clauses are not allowed:

CCSID	LOCKSIZE PAGE	PCTFREE
COMPRESS	LOCKSIZE ROW	SEGSIZE
FREEPAGE	MAXPARTITIONS	TRACKMOD
LOCKSIZE TABLE	NUMPARTS	

Recommended GBPCACHE setting for LOB table spaces: For LOB table spaces, use the GBPCACHE CHANGED option instead of the GBPCACHE SYSTEM option. Due to the usage patterns of LOBs, the use of GBPCACHE CHANGED can help avoid excessive and synchronous writes to disk and the group buffer pool.

Altering the logging attribute of a table space: See “Notes” on page 852 of “ALTER TABLESPACE” on page 841 for information about altering the logging attributes of a table space.

Table space row formats: Depending on the value of the SPRMRRF subsystem parameter, newly created table spaces will be in either re-ordered row format or basic row format. When the value of the SPRMRRF parameter is ENABLE, table spaces will be created in re-ordered row format. When the value of the SPRMRRF parameter is DISABLE, newly created table spaces will be created in basic row format. This includes universal table spaces, except for XML table spaces, which are always created in re-ordered row format, regardless of the value of the SPRMRRF parameter.

Making a partitioned table space larger: Depending on the needs of your application, you might need to increase the size of a partitioned table space to hold more data by either adding more partitions or by increasing the size of the existing partitions:

- To add more partitions, use the ALTER TABLE statement with the ADD PARTITION clause.
- To increase the size of the partitions, use the following steps:
 1. Unload the data rows from the table space, if necessary.
 2. Drop the table space. The table and any indexes, views, or synonyms dependent on the table are dropped, and authorizations for the table and views are revoked.

3. Recreate the table space (specifying an appropriate value for the DSSIZE clause), the table (create the partitioning scheme for the table using either table-controlled or index-controlled partitioning), and the necessary indexes.
4. Recreate views and synonyms. Reestablish appropriate authorizations.
5. Load data into the new table.
6. Rebind any plans and packages that have changed.

Redistributing data between existing partitions: If you need to redistribute the data between the existing partitions to make better use of the space within the existing table, you can use either of these two methods:

- Use the ALTER TABLE statement with the ALTER PARTITION clause. You can alter the partitions to specify new partition boundaries to explicitly specify how to redistribute the data. Any affected partitions are set to REORG-pending status.
- Use the REORG utility with the REBALANCE keyword. REBALANCE specifies that the data is evenly redistributed across the partitions that are reorganized. See *DB2 Utility Guide and Reference* for information about using the REORG utility.

Rules for primary and secondary space allocation: You can specify the primary and secondary space allocation for table spaces and indexes or allow DB2 to choose them. Having DB2 choose the values, especially the secondary space quantity, increases the possibility of reaching the maximum data set size before running out of extents.

In the following rules that describe how allocation works, these terms are used:

PRIQTY, SECQTY

The keywords for CREATE TABLESPACE, ALTER TABLESPACE, CREATE INDEX, and ALTER INDEX.

specified-priqty

The user-specified value for PRIQTY.

specified-secqty

The user-specified value for SECQTY.

actual-priqty

The actual primary space allocation, in kilobytes.

actual-priqty-cylinders

The actual primary space allocation, in cylinders.

actual-secqty

The actual secondary space allocation, in kilobytes.

actual-secqty-cylinders

The actual secondary space allocation, in cylinders.

calculated-extent-cylinders

A value that is calculated by DB2 using a *sliding scale*. A sliding scale means that the first secondary extent allocations are smaller than later secondary allocations. For example, Figure 19 on page 1147 shows the sliding scale of secondary extent allocations that DB2 uses for a 64-GB data set. The size of each secondary extent is larger for each secondary extent that is allocated up to the 127th extent. For the 127th secondary extent and any subsequent extents, the secondary size allocation is 559 cylinders.

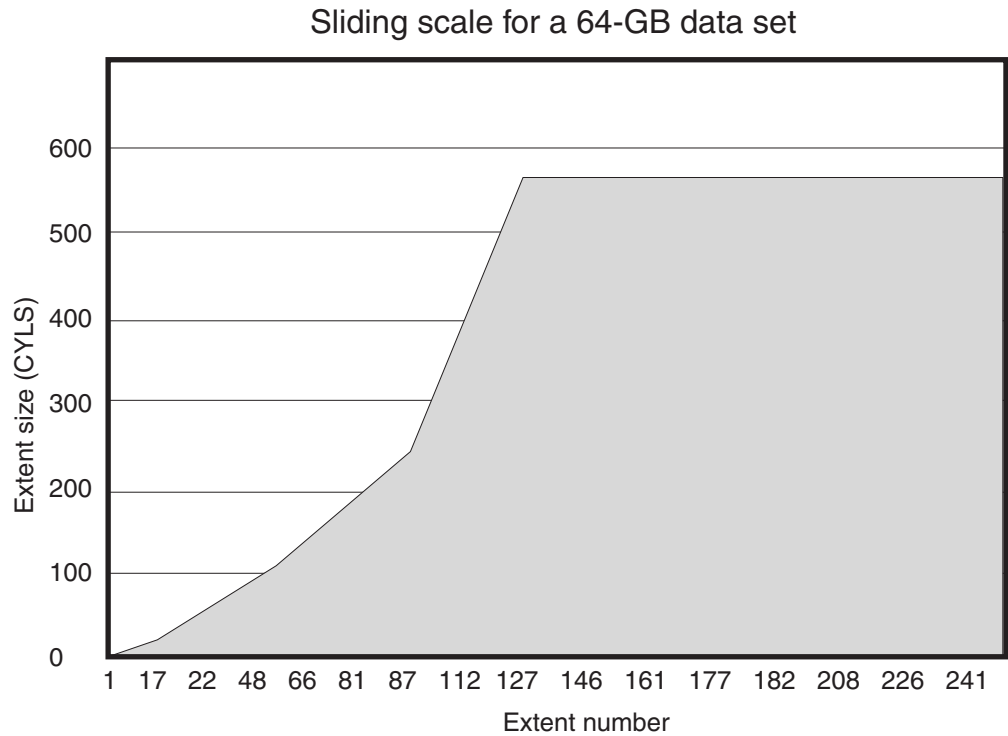


Figure 19. Sliding scale allocation of secondary extents for a 64 GB data set

The rules are:

- **Rule 1 (for primary space allocation)**

If PRIQTY is specified and *specified-priqty* is not equal to -1, *actual-priqty* is at least *specified-priqty* KB.

If PRIQTY is not specified or *specified-priqty* is equal to -1, *actual-priqty* is determined as follows:

- For a table space, if the TSQTY subsystem parameter value is specified and is greater than 0, *actual-priqty* is at least the value of TSQTY.

If the TSQTY subsystem parameter is not specified or is 0, *actual-priqty* is one cylinder for a non-LOB table space. *actual-priqty* is 10 cylinders for a LOB table space.

- For an index, if the IXQTY subsystem parameter value is specified and is greater than 0, *actual-priqty* is at least the value of IXQTY.

If the IXQTY subsystem parameter is not specified or is 0, *actual-priqty* is one cylinder.

- **Rule 2 (for secondary space allocation)**

If SECQTY is not specified, the following formulas determine *actual-secqty*:

- If the maximum size of a data set in the table space or index is less than 32 GB, the formula is:

actual-secqty-cylinders=
MAX(0.1*actual-priqty-cylinders, MIN(calculated-extent-cylinders, 127))

- If the maximum size of a data set in the table space or index is 32 GB or greater, the formula is:

actual-secqty-cylinders=
MAX(0.1*actual-priqty-cylinders, MIN(calculated-extent-cylinders, 559))

- **Rule 3 (for secondary space allocation)**

If SECQTY is 0, *actual-secqty* is 0.

- **Rule 4 (for secondary space allocation)**

This is the only rule that depends on the value of subsystem parameter MGEXTSZ (field OPTIMIZE EXTENT on installation panel DSNTIP7).

If MGEXTSZ is YES:

- If SECQTY is specified and *specified-secqty* is not equal to -1 or 0, the following formulas determine *actual-secqty*:
 - If the maximum size of a data set in the table space or index is less 32 GB, the formula is:

$$\text{actual-secqty-cylinders} = \text{MAX}(\text{MIN}(\text{calculated-extent-cylinders}, 127), \text{specified-secqty-cylinders})$$
 - If the maximum size of a data set in the table space or index is 32 GB or greater, the formula is:

$$\text{actual-secqty-cylinders} = \text{MAX}(\text{MIN}(\text{calculated-extent-cylinders}, 559), \text{specified-secqty-cylinders})$$

If MGEXTSZ is NO:

- For a table space, if SECQTY is *n*, the secondary space allocation is at least *n* kilobytes, with the following exceptions:
 - If SECQTY is greater than 4194304, *n* is 4194304 kilobytes.
 - For LOB table spaces:
 - For 4KB page sizes, if *integer* is greater than 0 and less than 200, *n* is 200.
 - For 8KB page sizes, if *integer* is greater than 0 and less than 400, *n* is 400.
 - For 16KB page sizes, if *integer* is greater than 0 and less than 800, *n* is 800.
 - For 32KB page sizes, if *integer* is greater than 0 and less than 1600, *n* is 1600.
 - For any page size, if *integer* is greater than 4194304, *n* is 4194304.
- For an index, if SECQTY is *integer*, the secondary space allocation is at least *n* kilobytes, where *n* is:

12	If SECQTY and PRIQTY are omitted
4194304	If <i>integer</i> is greater than 4194304
<i>integer</i>	If <i>integer</i> is not greater than 4194304

Alternative syntax and synonyms: For compatibility with previous releases of DB2, the following keywords are supported:

- You can specify the LOCKPART clause, but it has no effect. Starting with Version 8, DB2 treats all table spaces as if they were defined as LOCKPART YES. LOCKPART YES specifies the use of selective partition locking. When all the conditions for selective partition locking are met, DB2 locks only the partitions that are accessed. When the conditions for selective partition locking are not met, DB2 locks every partition of the table space.
 If you specify LOCKSIZE TABLESPACE, the table space must be defined with LOCKPART NO. LOCKSIZE TABLESPACE and LOCKPART YES are mutually exclusive.
- When creating a partitioned table space, you can specify PART as a synonym for PARTITION.
- When specifying the logging attributes for a table space, you can specify LOG YES as a synonym for LOGGED, and you can specify LOG NO as a synonym for NOT LOGGED.

Although these keywords are supported as alternatives, they are not the preferred syntax.

Examples

Example 1: Create table space DSN8S91D in database DSN8D91A. Let DB2 define the data sets, using storage group DSN8G910. The primary space allocation is 52 kilobytes; the secondary, 20 kilobytes. The data sets need not be erased before they are deleted.

Locking on tables in the space is to take place at the page level. Associate the table space with buffer pool BP1. The data sets can be closed when no one is using the table space.

```
CREATE TABLESPACE DSN8S91D
  IN DSN8D91A
  USING STOGROUP DSN8G910
  PRIQTY 52
  SECQTY 20
  ERASE NO
  LOCKSIZE PAGE
  BUFFERPOOL BP1
  CLOSE YES;
```

For the above example, the underlying data sets for the table space will be created immediately, which is the default (DEFINE YES). If you want to defer the creation of the data sets until data is first inserted into the table space, you would specify DEFINE NO instead of accepting the default behavior.

Example 2: Assume that a large query database application uses a table space to record historical sales data for marketing statistics. Create large table space SALESHX in database DSN8D91A for the application. Create it with 82 partitions, specifying that the data in partitions 80 through 82 is to be compressed.

Let DB2 define the data sets for all the partitions in the table space, using storage group DSN8G910. For each data set, the primary space allocation is 4000 kilobytes, and the secondary space allocation is 130 kilobytes. Except for the data set for partition 82, the data sets do not need to be erased before they are deleted.

Locking on the table is to take place at the page level. There can only be one table in a partitioned table space. Associate the table space with buffer pool BP1. The data sets cannot be closed when no one is using the table space. If there are no CLOSE YES data sets to close, DB2 might close the CLOSE NO data sets when the DSMAX is reached.

```
CREATE TABLESPACE SALESHX
  IN DSN8D91A
  USING STOGROUP DSN8G910
  PRIQTY 4000
  SECQTY 130
  ERASE NO
  Numparts 82
  (PARTITION 80
   COMPRESS YES,
   PARTITION 81
   COMPRESS YES,
   PARTITION 82
   COMPRESS YES
   ERASE YES)
  LOCKSIZE PAGE
  BUFFERPOOL BP1
  CLOSE NO;
```


Example 3: Assume that a column named EMP_PHOTO with a data type of BLOB(110K) has been added to the sample employee table for each employee's photo. Create LOB table space PHOTOLTS in database DSN8D91A for the auxiliary table that will hold the BLOB data.

Let DB2 define the data sets for the table space, using storage group DSN8G910. For each data set, the primary space allocation is 3200 kilobytes, and the secondary space allocation is 1600 kilobytes. The data sets do not need to be erased before they are deleted. (Because ERASE NO is the default, the clause does not have to be explicitly specified to get the desired behavior.)

```
CREATE LOB TABLESPACE PHOTOLTS
  IN DSN8D91A
  USING STOGROUP DSN8G910
    PRIQTY 3200
    SECQTY 1600
  LOCKSIZE LOB
  BUFFERPOOL BP16K0
  GBPCACHE SYSTEM
  NOT LOGGED
  CLOSE NO;
```

Example 4: The following example creates a range-partitioned universal table space, TS1, in database DSN8D91A using storage group DSN8G910. The table space has 16 pages per segment and has 55 partitions. It specifies LOCKSIZE ANY.

```
CREATE TABLESPACE TS1
  IN DSN8D91A
  USING STOGROUP DSN8G910
  NUMPARTS 55
  SEGSIZE 16
  LOCKSIZE ANY;
```

Example 5: The following example creates a range-partitioned universal table space, TS2, in database DSN8D91A using storage group DSN8G910. The table space has 64 pages per segment and has seven defer-defined partitions, where every other partition is compressed.

```
CREATE TABLESPACE TS2
  IN DSN8D91A
  USING STOGROUP DSN8G910
  NUMPARTS 7
  (
    PARTITION 1 COMPRESS YES,
    PARTITION 3 COMPRESS YES,
    PARTITION 5 COMPRESS YES,
    PARTITION 7 COMPRESS YES
  )
  SEGSIZE 64
  DEFINE NO;
```

Example 6: The following example creates a partition-by-growth table space that has a maximum size of 2 GB for each partition, four pages per segment with a maximum of 24 partitions for the table space.

```
CREATE TABLESPACE TS01TS IN TS01DB USING STOGROUP SG1
  DSSIZE 2G
  MAXPARTITIONS 24
  LOCKSIZE ANY
  SEGSIZE 4;
```

CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger in a schema and builds a trigger package at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

In defining a trigger on a table, the privilege set that is defined below must include SYSADM authority or each of the following:

- The SELECT privilege on the table on which the trigger is defined if any transition variables or transition tables are specified
- The SELECT privilege on any table or view to which the search condition of triggered action refers
- The necessary privileges to invoke the triggered SQL statements in the triggered action
- The authorization to define a trigger on the table, which must include at least one of the following:
 - The TRIGGER privilege on the table on which the trigger is defined
 - The ALTER privilege on the table on which the trigger is defined
 - DBADM authority on the database that contains the table
 - SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

In defining a trigger on a view, the privilege set that is defined below must include SYSADM authority or each of the following:

- The SELECT privilege on any table or view to which the search condition of triggered action refers
- The necessary privileges to invoke the triggered SQL statements in the triggered action
- The authorization to define a trigger on the view, which must include at least one of the following:
 - Ownership of the view on which the trigger is defined
 - SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the owner is a role, the implicit schema match does not apply and this role needs to include one of the previously listed conditions.

If the statement is dynamically prepared and is not running in a trusted context for which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the

Description

trigger-name

Names the trigger. The name, including the implicit or explicit schema name, must not identify a trigger that exists at the current server.

The name is also used to create the trigger package; therefore, the name must also not identify a package that is already described in the catalog. The schema name becomes the collection-id of the trigger package. Although *trigger-name* can be specified as an ordinary or delimited identifier, the name should conform to the rules for an ordinary identifier. Refer to The implicitly created trigger package for additional information.

The schema name must not begin with 'SYS' unless the name is 'SYSADM', or the schema name is 'SYSTOOLS' and the user who executes the CREATE statement has SYSADM or SYSCTRL privilege.

NO CASCADE BEFORE

Specifies that the trigger is a before trigger. DB2 executes the triggered action before it applies any changes caused by an insert, delete, or update operation on the subject table. It also specifies that the triggered action does not activate other triggers because the triggered action of a before trigger cannot contain any updates.

NO CASCADE BEFORE must not be specified when *view-name* is also specified. FOR EACH ROW must be specified for a BEFORE trigger.

AFTER

Specifies that the trigger is an after trigger. DB2 executes the triggered action after it applies any changes caused by an insert, delete, or update operation on the subject table. AFTER must not be specified if *view-name* is also specified.

INSTEAD OF

Specifies that the trigger is an instead of trigger. The associated triggered action replaces the action against the subject view. Only one INSTEAD OF trigger is allowed for each type of operation on a given subject view. DB2 executes the triggered-action instead of the insert, update, or delete operation on the subject view.

INSTEAD OF must not be specified when *table-name* is also specified. The WHEN clause can not be specified for an INSTEAD OF trigger. FOR EACH STATEMENT must not be specified for an INSTEAD OF trigger.

INSERT

Specifies that the trigger is an insert trigger. DB2 executes the triggered action whenever there is an insert operation on the subject table. However, if the insert trigger is defined on any explain table, and the insert operation was caused by DB2 adding a row to the table, the triggered action is not to be executed.

DELETE

Specifies that the trigger is a delete trigger. DB2 executes the triggered action whenever there is a delete operation on the subject table.

UPDATE

Specifies that the trigger is an update trigger. DB2 executes the triggered action whenever there is an update operation on the subject table.

If you do not specify a list of column names, an update operation on any column of the subject table, including columns that are subsequently added with the ALTER TABLE statement, activates the triggered action.

OF *column-name*,...

Each *column-name* that you specify must be a column of the subject table and must appear in the list only once. An update operation on any of the listed columns activates the triggered action.

UPDATE OF *column-name* cannot be specified for an INSTEAD OF trigger.

ON *table-name*

Identifies the subject table of the BEFORE or AFTER trigger definition. The name must identify a base table that exists at the current server. It must not identify a materialized query table, a clone table, a temporary table, an auxiliary table, an alias, a synonym, a real-time statistics table, or a catalog table.

ON *view-name*

Identifies the subject view of the INSTEAD OF trigger definition. The name must identify a view that exists at the current server.

view-name must not specify a view where any of the following conditions are true:

- The view is defined with the WITH CASCADED CHECK option (a symmetric view)
- The view on which a symmetric view has been defined
- The view references data that is encoded with different encoding schemes or CCSID values
- The view has a column that is a rowid, LOB, or XML (or a distinct type that is defined as one of these types)
- The view has a column that is based on an underlying column that is defined as an identity column, security label column, or a row change timestamp column
- The view has a column that is defined (directly or indirectly) as an expression
- The view has a column that is based on a column of a result table that involves a set operator
- The view has columns that have field procedures
- All of the underlying tables of the view are catalog tables or created global temporary tables
- The view has other views that are dependent on it

REFERENCING

Specifies the correlation names for the transition variables and the table names for the transition tables. For the rows in the subject table that are modified by the triggering SQL operation (insert, delete, or update), a correlation name identifies the columns of a specific row. *table-identifiers* identify the complete set of affected rows. Transition variables with XML types cannot be referenced inside of a trigger. If the column of a transition table is referenced, the data type of the column cannot be XML.

Each row that is affected by the triggering SQL operation is available to the triggered action by qualifying column names with *correlation-names* that are specified as follows:

OLD AS *correlation-name*

Specifies the correlation name that identifies the values in the row prior to the triggering SQL operation.

NEW AS *correlation-name*

Specifies the correlation name that identifies the values in the row as modified by the triggering SQL operation and by any SET statement in a before trigger that has already been executed.

The complete set of rows that are affected by the triggering operation is available to the triggered action by using *table-identifiers* that are specified as follows:

OLD_TABLE AS *table-identifier*

Specifies the name of a temporary table that identifies the values in the complete set of rows that are modified rows by the triggering SQL operation prior to any actual changes.

NEW_TABLE AS *table-identifier*

Specifies the name of a temporary table that identifies the values in the complete set of rows as modified by the triggering SQL operation and by any SET statement in a before trigger that has already been executed.

Only one OLD and one NEW *correlation-name* can be specified for a trigger. Only one OLD_TABLE and one NEW_TABLE *table-identifier* can be specified for a trigger. All of the *correlation-names* and *table-identifiers* must be unique from one another.

Table 106 on page 1156 summarizes the allowable combinations of transition variables and transition tables that you can specify for the various trigger types. The OLD *correlation-name* and the OLD_TABLE *table-identifier* are valid only if the triggering event is either a delete operation or an update operation. For a delete operation, the OLD *correlation-name* captures the values of the columns in the deleted row, and the OLD_TABLE *table-identifier* captures the values in the set of deleted rows. For an update operation, the OLD *correlation-name* captures the values of the columns of a row before the update operation, and the OLD_TABLE *table-identifier* captures the values in the set of updated rows.

The NEW *correlation-name* and the NEW_TABLE *table-identifier* are valid only if the triggering event is either an insert operation or an update operation. For both operations, the NEW *correlation-name* captures the values of the columns in the inserted or updated row and the NEW_TABLE *table-identifier* captures the values in the set of inserted or updated rows. For BEFORE triggers, the values of the updated rows include the changes from any SET statements in the triggered action of BEFORE triggers.

The OLD and NEW *correlation-name* variables cannot be modified in an AFTER or INSTEAD OF trigger.

Table 106. Allowable combinations of attributes in a trigger definition

Granularity	Activation time	Triggering SQL operation	Transition variables allowed ¹	Transition tables allowed ¹
FOR EACH ROW	BEFORE	DELETE	OLD	None
		INSERT	NEW	None
		UPDATE	OLD, NEW	None
	AFTER	DELETE	OLD	OLD_TABLE
		INSERT	NEW	NEW_TABLE
		UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
	INSTEAD OF	DELETE	OLD	OLD_TABLE
		INSERT	NEW	NEW_TABLE
		UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
FOR EACH STATEMENT	AFTER	DELETE	None	OLD_TABLE
		INSERT	None	NEW_TABLE
		UPDATE	None	OLD_TABLE, NEW_TABLE

Note:

1. If a transition table or variable is referenced where it is not allowed, an error is returned.

A transition variable that has a character data type inherits the subtype and CCSID of the column of the subject table. During the execution of the triggered action, the transition variables are treated like host variables. Therefore, character conversion might occur. However, unlike a host variable, a transition variable can have the bit data attribute, and character conversion never occurs for bit data. A transition variable is considered to be bit data if the column of the table to which it corresponds is bit data.

You cannot modify a transition table; transition tables are read-only. Although a transition table does not inherit any edit or validation procedures from the subject table, it does inherit the encoding scheme and field procedures of the subject table.

The scope of each *correlation-name* and each *table-identifier* is the entire trigger definition.

FOR EACH ROW or FOR EACH STATEMENT

Specifies the conditions for which DB2 executes the triggered action.

FOR EACH ROW

Specifies that DB2 executes the triggered action for each row of the subject table that the triggering SQL operation modifies. If the triggering SQL operation does not modify any rows, the triggered action is not executed.

FOR EACH STATEMENT

Specifies that DB2 executes the triggered action only one time for the triggering operation. Even if the triggering operation does not modify or delete any rows, the triggered action is executed one time.

FOR EACH STATEMENT must not be specified for a BEFORE or INSTEAD OF trigger.

MODE DB2SQL

Specifies the mode of the trigger. MODE DB2SQL triggers are activated after all of the row operations have occurred.

triggered-action

Specifies the action to be performed when the trigger is activated. The *triggered-action* is composed of one or more SQL statements and an optional condition that controls whether the statements are executed.

WHEN (*search-condition*)

Specifies a condition that evaluates to true, false, or unknown. The triggered SQL statements are executed only if the *search-condition* evaluates to true. If the WHEN clause is omitted, the associated SQL statements are always executed. The condition for a before trigger must not include a subselect that references the subject table.

The WHEN clause must not be specified for an INSTEAD OF trigger.

SQL-trigger-body

Specifies the SQL statements that are to be executed for the triggered action.

triggered-SQL-statement

Specifies a single SQL statement that is to be executed for the triggered action.

BEGIN ATOMIC *triggered-SQL-statement*,... **END**

Specifies a list of SQL statements that are to be executed for the triggered action. The statements are executed in the order in which they are specified.

SQL processor programs, such as SPUFI, the command line processor, and DSNTEP2, might not correctly parse SQL statements in the triggered action that are ended with semicolons. These processor programs accept multiple SQL statements, each separated with a terminator character, as input. Processor programs that use a semicolon as the SQL statement terminator can truncate a CREATE TRIGGER statement with embedded semicolons and pass only a portion of it to DB2. Therefore, you might need to change the SQL terminator character for these processor programs. For information on changing the terminator character for SPUFI and DSNTEP2, see *DB2 Application Programming and SQL Guide*.

Only certain SQL statements can be specified in the *SQL-trigger-body*. Table 107 shows the list of allowable SQL statements, which differs depending on whether the trigger is being defined as BEFORE, AFTER, or INSTEAD OF. An 'X' in the table indicates that the statement is valid.

Table 107. Allowable SQL statements

SQL statement	Trigger activation time		
	BEFORE	AFTER	INSTEAD OF
CALL	X	X	X
DELETE (searched)		X	X
fullselect	X	X	X
INSERT		X	X
MERGE		X	X
REFRESH TABLE		X	X

Table 107. Allowable SQL statements (continued)

SQL statement	Trigger activation time		
	BEFORE	AFTER	INSTEAD OF
SET transition variable	X		
SIGNAL	X	X	X
UPDATE (searched)		X	X
VALUES	X	X	X

The statements in the triggered action have these restrictions:

- They must not refer to host variables, parameter markers, undefined transition variables, or declared temporary tables.
- They must only refer to a table or view that is at the current server.
- They must only invoke a stored procedure or user-defined function that is at the current server. An invoked routine can, however, access a server other than the current server.
- They must not contain a fullselect that refers to the subject table if the trigger is defined as BEFORE.

The triggered action can refer to the values in the set of affected rows. This action is supported through the use of transition variables and transition tables.

Transition variables use the names of the columns in the subject table qualified by a specified name that identifies whether the reference is to the old value (before the update) or the new value (after the update). A transition variable can be referenced in *search-condition* or *triggered-SQL-statement* of the triggered action wherever a host variable is allowed in the statement if it were issued outside the body of a trigger.

Transition tables can be referenced in the triggered action of an after trigger. Transition tables are read-only. Transition tables also use the name of the columns of the subject table but have a name specified that allows the complete set of affected rows to be treated as a table. The name of the transition table can be referenced in *triggered-SQL-statement* of the triggered action whenever a table name is allowed in the statement if it were issued outside the body of a trigger. The name of the transition table can be specified in *search-condition* or *triggered-SQL-statement* of the triggered action whenever a column name is allowed in the statement if it were issued outside the body of a trigger.

In addition, a transition table can be passed as a parameter to a user-defined function or procedure specifying the TABLE keyword before the name of the transition table. When the function or procedure is invoked, a table locator is passed for the transition table.

A transition variable or transition table is not affected after being returned from a procedure invoked from within a triggered action regardless of whether the corresponding parameter was defined in the CREATE PROCEDURE statement as IN, INOUT, or OUT.

Notes

Owner privileges:

When an INSTEAD OF trigger is defined, the associated privilege (INSERT, UPDATE, or DELETE on the view) is given to the owner of the view. The

owner is granted the privilege with the ability to grant that privilege to others. For more information about ownership of an object, see “Authorization, privileges, and object ownership” on page 60.

Execution authorization:

The user executing the triggering SQL operation does not need authority to execute a triggered-SQL-statement. A triggered-SQL-statement will execute using the authority of the owner of the trigger.

Activating a trigger:

Only insert, delete, or update operations can activate a trigger. The activation of a trigger might cause trigger cascading. *Trigger cascading* is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions might also cause updates as a result of the original modification, which can result in the activation of additional triggers. With trigger cascading, a significant chain of triggers might be activated, causing a significant change to the database as a result of a single insert, delete, or update operation.

Loading a table with the LOAD utility does not activate any triggers that are defined for the table if SHRLEVEL NONE is specified as part of the LOAD utility or if the default SHRLEVEL option for LOAD is taken. If SHRLEVEL CHANGE is specified as part of the LOAD utility, triggers are activated when loading a table with the LOAD utility.

Adding triggers to enforce constraints:

Adding a trigger on a table that already has rows in it will not cause the triggered action to be executed. Thus, if the trigger is designed to enforce constraints on the data in the table, the data in the existing rows might not satisfy those constraints.

Multiple triggers:

Multiple triggers that have the same triggering SQL operation and activation time can be defined on a table. The triggers are activated in the order in which they were created. For example, the trigger that was created first is executed first; the trigger that was created second is executed second.

Read-only views:

The addition of an INSTEAD OF trigger for a view affects the read only characteristic of the view. If a read-only view has a dependency relationship with an INSTEAD OF trigger, the type of operation that is defined for the INSTEAD OF trigger defines whether the view is deletable, insertable, or updatable.

The creation of an INSTEAD OF trigger causes dependent packages, plans, and statements in the dynamic statement cache to be marked invalid if the view definition is not read-only.

Invalidation of plans and packages:

A trigger package becomes invalid if an object or privilege on which it depends is dropped or revoked. The next time the trigger is activated, DB2 attempts to rebind the invalid trigger package. If the automatic rebind is unsuccessful, the trigger package remains invalid.

You cannot create another package from the trigger package, such as with the BIND COPY command. The only way to drop a trigger package is to

drop the trigger or the subject table. Dropping the trigger drops the trigger package; dropping the subject table drops the trigger and the trigger package.

DB2 creates the trigger package with the following initial attributes (some of these attributes can be modified using the REBIND TRIGGER PACKAGE command):

- ACTION(ADD)
- CURRENTDATA(NO)
- DBPROTOCOL(DRDA)
- DEGREE(1)
- DYNAMICRULES(BIND)
- ENABLE(*)
- ENCODING(0)
- EXPLAIN(NO)
- FLAG(I)
- ISOLATION(CS)
- REOPT(NONE) and NODEFER(PREPARE)
- OWNER(authorization ID) or ROLE of appropriate
- QUERYOPT(1)
- PATH(path)
- RELEASE(COMMIT)
- ROUNDING(value from the CURRENT DECFLOAT ROUNDING MODE special register)
- SQLERROR(NOPACKAGE)
- QUALIFIER(authorization ID)
- VALIDATE(BIND)

The values of OWNER, QUALIFIER, and PATH are set depending on whether the CREATE TRIGGER statement is embedded in a program or issued interactively. If the statement is embedded in a program, OWNER and QUALIFIER are the owner and qualifier of the package or plan. PATH is the value from the PATH bind option. If the statement is issued interactively, both OWNER and QUALIFIER are the SQL authorization ID. PATH is the value in the CURRENT PATH special register.

Transition variable values and INSTEAD OF triggers:

The initial values for new transition variables or new transition table columns that are visible in an INSTEAD OF INSERT trigger are set as follows:

- If a value is explicitly specified for a column in the insert operation, the corresponding new transition variable is that explicitly specified value.
- If a value is not explicitly specified for a column in the insert operation or the DEFAULT clause is specified, the corresponding new transition variable is:
 - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger)
 - otherwise, the null value

The initial values for new transition variables that are visible in an INSTEAD OF UPDATE trigger are set as follows:

- If a value is explicitly specified for a column in the update operation, the corresponding new transition variable is that explicitly specified value
- If the DEFAULT clause is explicitly specified for a column in the update operation, the corresponding new transition variable is:
 - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger)
 - otherwise, the null value
- Otherwise, the corresponding new transition variable is the existing value of the column in the row.

Considerations for a MERGE statement:

The MERGE statement can execute insert and update operations. The applicable INSERT or UPDATE triggers are activated for the MERGE statement when an insert or update operation is executed.

Considerations for implicitly hidden columns:

In the body of a trigger, a trigger transition variable that corresponds to an implicitly hidden column can be referenced. A trigger transition table, that corresponds to a table with an implicitly hidden column, includes that column as part of the transition table. Likewise, a trigger transition variable will exist for the column that is defined as implicitly hidden. A trigger transition variable that corresponds to an implicitly hidden column can be referenced in the body of a trigger.

Considerations for the special plan, statement, and function tables for EXPLAIN:

You can create a trigger on PLAN_TABLE, DSN_STATEMENT_TABLE, or DSN_FUNCTION_TABLE. However, insert triggers that are defined on these tables are not activated when DB2 adds rows to the tables.

Adding columns to a subject table or a table referenced in the *triggered action*:

If a column is added to the subject table after triggers have been defined, the following rules apply:

- If the trigger is an update trigger that was defined without an explicit list of column names, an update to the new column activates the trigger.
- If the subject table is referenced in the *triggered-action*, the new column is not accessible to the SQL statements until the trigger package is rebound.
- The OLD_TABLE and the NEW_TABLE transition tables contain the new column, but the column cannot be referenced unless the trigger is recreated. If the transition tables are passed to a user-defined function or a stored procedure, the user-defined function or stored procedure must be recreated with the new definition of the table (that is, the function or procedure must be dropped and recreated), and the package for the user-defined function or stored procedure must be rebound.

If a column is added to any table that is referenced in the *triggered-action*, the new column is not accessible to the SQL statements until the trigger package is rebound.

Altering the attributes of a column that the *triggered action* references:

If a column is altered in the table on which the trigger is defined (the subject table), the alter is processed, and the dependent trigger packages are invalidated.

Renaming the table for which the trigger is defined, or tables referenced in the *triggered action*:

You cannot rename a table for which a trigger is defined (the subject table).

Except for the subject table, you can rename any table to which the SQL statements in the triggered action refer. After renaming such a table, drop the trigger and then re-create the trigger so that it refers to the renamed table.

Dependencies when dropping objects and revoking privileges:

The following dependencies apply to a trigger:

- Dropping the subject table (the table on which the trigger is defined) causes the trigger and its package to also be dropped.
- Dropping any table, view, alias, or index that is referenced or used within the SQL statements in the triggered action causes the trigger and its package to be invalidated. Dropping a referenced synonym has no effect.
- Dropping a user-defined function that is referenced by the SQL statements in the triggered action is not allowed. An error occurs.
- Dropping a sequence that is referenced by the SQL statements in the triggered action is not allowed. An error occurs.
- Revoking a privilege on which the trigger depends causes the trigger and its package to be invalidated.

Errors executing triggers:

Severe errors that occur during the execution of triggered SQL statements are returned with SQLCODE -901, -906, -911, and -913 and the corresponding SQLSTATE. Non-severe errors raised by a triggered SQL statement that is a SIGNAL statement or that contains a RAISE_ERROR function are returned with SQLCODE -438 and the SQLSTATE that is specified in the SIGNAL statement or the RAISE_ERROR condition. Other non-severe errors are returned with SQLCODE -723 and SQLSTATE 09000. Warnings are not returned.

Special registers:

The values of the special registers are saved before a trigger is activated and are restored on return from the trigger.

The following table describes how special registers are set on entry to the trigger body. Some of the special registers are applicable only to dynamic SQL. Although dynamic SQL statements are not allowed directly in the triggered SQL statements, they are allowed in a user-defined function or stored procedure that is invoked by the triggered SQL statements.

Table 108. Rules for the values of special registers in triggers

Special register	The value is
CURRENT DATE CURRENT TIME CURRENT TIMESTAMP	Inherited from the triggering SQL operation (delete, insert, update). All triggered SQL statements, including the SQL statements in a user-defined function or a stored procedure invoked by the trigger, inherit these values.
CURRENT PACKAGESET	Set to the schema name of the trigger
CURRENT TIMEZONE	Set to the TIMEZONE parameter

Table 108. Rules for the values of special registers in triggers (continued)

Special register	The value is
<ul style="list-style-type: none"> • CURRENT CLIENT_ACCTNG • CURRENT CLIENT_APPLNAME • CURRENT CLIENT_USERID • CURRENT CLIENT_WRKSTNNAME • 	Inherited from the triggering SQL operation (delete, insert, update)
<ul style="list-style-type: none"> • CURRENT APPLICATION • CURRENT ENCODING SCHEME • 	
<ul style="list-style-type: none"> • CURRENT DECFLOAT ROUNDING • CURRENT MODE • CURRENT DEBUG MODE • CURRENT DEGREE • CURRENT LC_CTYPE • 	
<ul style="list-style-type: none"> • CURRENT MAINTAINED • CURRENT TABLE TYPES • CURRENT MEMBER • CURRENT OPTIMIZATION HINT • CURRENT PATH • CURRENT PACKAGE PATH • CURRENT PRECISION • CURRENT REFRESH AGE • CURRENT RULES • CURRENT SERVER • CURRENT SQLID • SESSION_USER 	

Result sets for stored procedures:

If a trigger invokes a stored procedure that returns result sets, the application that activated the trigger cannot access those result sets.

Transaction isolation:

All of the statements in the *SQL-trigger-body* run under the isolation level in effect for the trigger.

Limiting processor time:

The DB2 resource limit facility allows you to specify the maximum amount of processor time for a dynamic, manipulative SQL statement such as SELECT or SQL data change statements. The execution of a trigger is counted as part of the triggering SQL statement.

The implicitly created trigger package:

When you create a trigger, DB2 automatically creates a trigger package with the same name as the trigger name. The collection name of the trigger package is the schema name of the trigger, and the version identifier is the empty string. Multiple versions of a trigger package are not allowed.

Use the REBIND command to explicitly rebind the trigger package. To specify the name of a trigger package for the bind commands, the name must conform to the rules for an ordinary identifier.

Defining triggers on tables that contain XML columns:

Although a trigger can be defined on a table that contains an XML column,

| an XML column cannot be referenced with a trigger transition variable in
| the trigger body. An *SQL-procedure-statement* cannot reference a transition
| variable that is an XML data type.

Errors binding triggers:

When a CREATE TRIGGER statement is bound, the SQL statements within the triggered action might not be fully parsed. Syntax errors in those statements might not be caught until the CREATE TRIGGER statement is executed.

Alternative syntax and synonyms:

To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- OLD TABLE as a synonym for OLD_TABLE
- NEW TABLE as a synonym for NEW_TABLE

Examples

Example 1: Create two triggers that track the number of employees that a company manages. The subject table is the EMPLOYEE table, and the triggers increment and decrement a column with the total number of employees in the COMPANY_STATS table. The tables have these columns:

EMPLOYEE table: ID, NAME, ADDRESS, and POSITION

COMPANY_STATS table: NBEMP, NBPRODUCT, and REVENUE

This example shows the use of transition variables in a row trigger to maintain summary data in another table.

Create the first trigger, NEW_HIRE, so that it increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table, increase the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END
```

Create the second trigger, FORM_EMP, so that it decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE, decrease the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER FORM_EMP
  AFTER DELETE ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1;
  END
```

Example 2: Create a trigger, REORDER, that invokes user-defined function ISSUE_SHIP_REQUEST to issue a shipping request whenever a parts record is updated and the on-hand quantity for the affected part is less than 10% of its maximum stocked quantity. User-defined function ISSUE_SHIP_REQUEST orders a quantity of the part that is equal to the part's maximum stocked quantity minus its on-hand quantity; the function also ensures that the request is sent to the appropriate supplier.

The parts records are in the PARTS table. Although the table has more columns, the trigger is activated only when columns ON_HAND and MAX_STOCKED are updated.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS NROW
  FOR EACH ROW MODE DB2SQL
  WHEN (NROW.ON_HAND < 0.10 * NROW.MAX_STOCKED)
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(NROW.MAX_STOCKED - NROW.ON_HAND, NROW.PARTNO));
  END
```

Example 3: Repeat the scenario in *Example 2* except use a fullselect instead of a VALUES statement to invoke the user-defined function. This example also shows how to define the trigger as a statement trigger instead of a row trigger. For each row in the transition table that evaluates to true for the WHERE clause, a shipping request is issued for the part.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW TABLE AS NTABLE
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    SELECT ISSUE_SHIP_REQUEST(MAX_STOCKED - ON_HAND, PARTNO)
      FROM NTABLE
     WHERE (ON_HAND < 0.10 * MAX_STOCKED);
  END
```

Example 4: Assume that table EMPLOYEE contains column SALARY. Create a trigger, SAL_ADJ, that prevents an update to an employee's salary that exceeds 20% and signals such an error. Have the error that is returned with an SQLSTATE of '75001' and a description. This example shows that the SIGNAL statement is useful for restricting changes that violate business rules.

```
CREATE TRIGGER SAL_ADJ
  AFTER UPDATE OF SALARY ON EMPLOYEE
  REFERENCING OLD AS OLD_EMP
               NEW AS NEW_EMP
  FOR EACH ROW MODE DB2SQL
  WHEN (NEW_EMP.SALARY > (OLD_EMP.SALARY * 1.20))
  BEGIN ATOMIC
    SIGNAL SQLSTATE '75001' ('Invalid Salary Increase - Exceeds 20');
  END
```

Example 5: Assume that the following statements create a table, WEATHER (which stores temperature values in Fahrenheit), and a view, CELSIUS_WEATHER for users who prefer to work in Celsius instead of Fahrenheit:

```
CREATE TABLE WEATHER
  (CITY VARCHAR(25),
   TEMPF DECIMAL(5,2));
CREATE VIEW CELSIUS_WEATHER (CITY, TEMPC) AS
  SELECT CITY, (TEMPF-32)/1.8
     FROM WEATHER;
```

The following INSTEAD OF trigger is used on the CELSIUS_WEATHER view to convert Celsius values to Fahrenheit values and then insert the Fahrenheit value into the WEATHER table:

```
CREATE TRIGGER CW_INSERT INSTEAD OF INSERT
  ON CELSIUS_WEATHER
  REFERENCING NEW AS NEWCW
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
```

```
|          INSERT INTO WEATHER VALUES  
|          (NEWCW.CITY,  
|            1.8*NEWCW.TEMPC+32)  
|        END;  
  
|
```

CREATE TRUSTED CONTEXT

The CREATE TRUSTED CONTEXT statement defines a trusted context at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

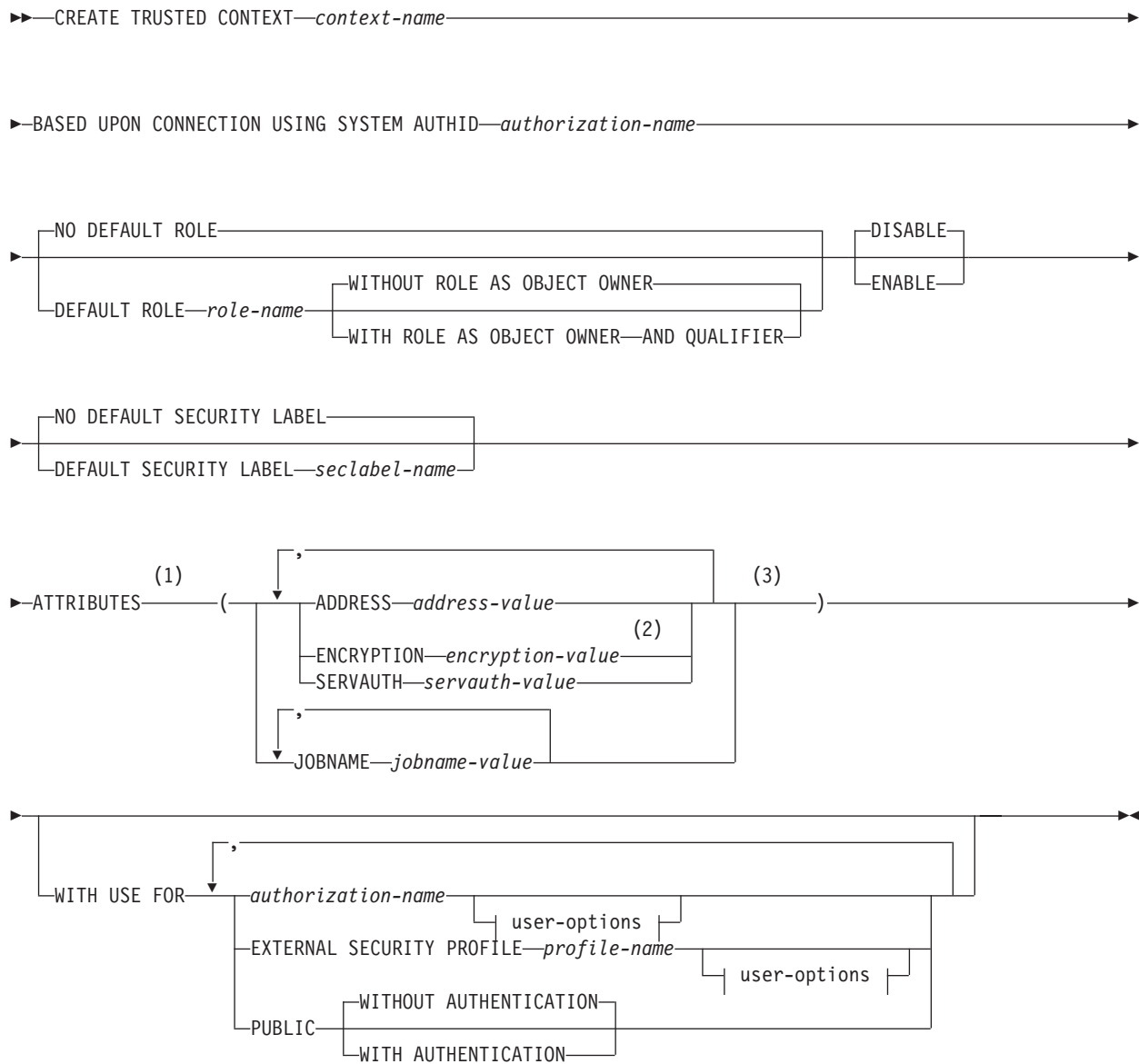
Authorization

The privilege set that is defined below must include SYSADM authority.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process unless the process is within a trusted context and the ROLE AS OBJECT OWNER clause is specified. In that case, the privileges set is the privileges that are held by the role that is associated with the primary authorization ID of the process.

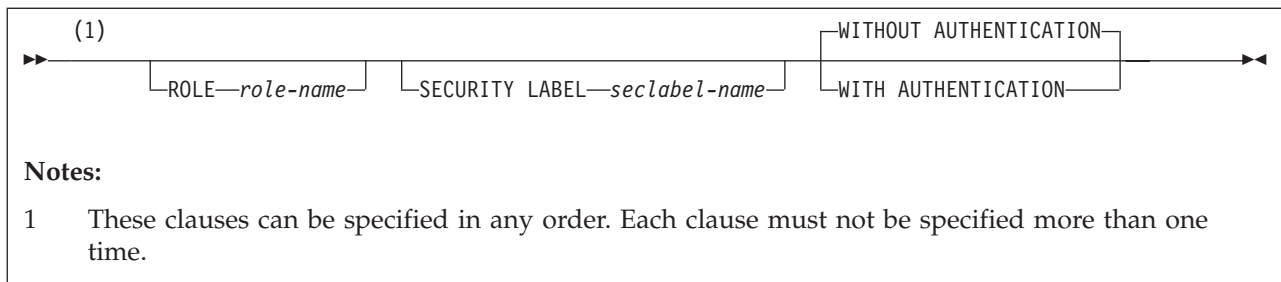
Syntax



Notes:

- 1 This clause and the clauses that follow can be specified in any order. Each clause must not be specified more than one time.
- 2 ENCRYPTION must not be specified more than one time.
- 3 Each pair of attribute name and corresponding value must be unique.

user-options:



Description

context-name

Names the trusted context. The name must not identify a trusted context that exists at the current server.

BASED UPON CONNECTION USING SYSTEM AUTHID *authorization-name*

Specifies that the context is a connection that is established by the authorization ID that is specified by *authorization-name*. The system authorization ID is the primary authorization ID. For a remote connection, it is derived from the system user ID that is provided by an external entity, such as a middleware server. For a local connection, the system authorization ID is derived depending on the sources, as specified in Table 109.

Table 109. System authorization ID for a local connection

Source of local connection	System authorization ID
Started task (RRSAF)	USER parameter on JOB statement or RACF USER.
TSO	TSO logon ID
BATCH	USER parameter on JOB statement

authorization-name must not be associated with an existing trusted context.

NO DEFAULT ROLE or DEFAULT ROLE *role-name*

Specifies whether a default role is associated with a trusted connection that is based on the specified trusted context.

NO DEFAULT ROLE

Specifies that the trusted context does not have a default role. The authorization ID of the process is the owner of any object that is created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create that object.

NO DEFAULT ROLE is the default.

DEFAULT ROLE *role-name*

Specifies that *role-name* is the role for the trusted context. *role-name* must identify a role that exists at the current server. This role is used with the user in a trusted connection that is based on the specified trusted context when the user does not have a user-specified role that is defined as part of the definition of this trusted context.

WITHOUT ROLE AS OBJECT OWNER or WITH ROLE AS OBJECT OWNER AND QUALIFIER

Specifies whether a role is used as the owner of objects that are created using a trusted connection that is based on the specified trusted context.

WITHOUT ROLE AS OBJECT OWNER

Specifies that a role is not used as the owner of the objects that are created

using a trusted connection that is based on the specified trusted context. The authorization ID of the process is the owner of any object that is the created using a trusted connection that is based on this trusted context. That authorization ID must possess all of the privileges that are necessary to create the object.

WITHOUT ROLE AS OBJECT OWNER is the default.

WITH ROLE AS OBJECT OWNER AND QUALIFIER

Specifies that the context assigned role is the owner of the objects that are created using a trusted connection that is based on this trusted context and that role must possess all of the privileges that are necessary to create the object. The context assigned role is the role that is defined for the user within this trusted context, if one is defined. Otherwise, the role is the default role that is associated with the trusted context. The role is also used as the grantor for any GRANT statements that are issued, and the revoker for any REVOKE statement that are issued using a trusted connection that is based on this trusted context.

AND QUALIFIER

Specifies that *role-name* will be used as the default for the CURRENT SCHEMA special register. The *role-name* will also be included in the SQL PATH (in place of CURRENT SQLID).

When **WITH ROLE AS OBJECT OWNER AND QUALIFIER** is not specified, there is no change to the default for the CURRENT SCHEMA special register and the SQL PATH.

DISABLE or ENABLE

Specifies whether the trusted context is created in the enabled or disabled state.

DISABLE

Specified that the trusted context is disabled when it is created. A trusted context that is disabled is not considered when a trusted connection is established. DISABLE is the default.

ENABLE

Specifies that the trusted context is enabled when it is created.

NO DEFAULT SECURITY LABEL or DEFAULT SECURITY LABEL *seclabel-name*

Specifies whether the trusted connection has a default security label.

NO DEFAULT SECURITY LABEL

Specifies that the trusted context does not have a default security label.

DEFAULT SECURITY LABEL *seclabel-name*

Specifies that *seclabel-name* is the default security label for the trusted context and is the security label that is used for multilevel security verification. *seclabel-name* must identify one of the RACF SECLABEL values that is defined for the SYSTEM AUTHID. This security label is used for a trusted connection that is based on the specified trusted context when the user does not have a specific security label defined as part of the definition of this trusted context. In this case, *seclabel-name* must also identify one of the RACF SECLABEL values that is defined for the user.

ATTRIBUTES

Specifies a list of one or more connection trust attributes that are used to define the trusted context.

ADDRESS *address-value*

Specifies the actual communication address that is used by the connection

to communicate with the database manager. The protocol supported is only for TCP/IP. The ADDRESS attribute can be specified multiple times, but each *address-value* must be unique.

When establishing a trusted connection, if multiple values are defined for the ADDRESS attribute for a trusted context, a candidate connection is considered to match this attribute if the address that is used by a connection matches any of the defined values for the ADDRESS attribute of the trusted context.

address-value specifies a string constant that contains the value that is associated with the ADDRESS trust attribute. *address-value* must be an IPv4 address, an IPv6 address, or a secure domain name with a length no greater than 254 bytes. No validation of *address-value* is done at the time the CREATE TRUSTED CONTEXT statement is processed. *address-value* must be left justified within the string constant.

- An IPv4 address is represented as a dotted decimal address. An example of an IPv4 address is 9.112.46.111
- An IPv6 address is represented as a colon hexadecimal address. An example of an IPv6 address is 2001:0DB8:0000:0000:0800:200C:417A. This address can also be express in a compressed form as 2001:DB8::8:800:200C:417A.
- A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is www.ibm.com. The gethostbyname socket call is used to resolve the domain name.

ENCRYPTION *encryption-value*

Specifies the minimum level of encryption of the data stream (network encryption).

encryption-value specifies a string constant that contains the value that is associated with the ENCRYPTION trust attribute. *encryption-value* must be left justified within the string constant. ENCRYPTION must not be specified more than one time in the statement. *encryption-value* must be one of the following:

- NONE, which specifies that no specific level of encryption is required.
- LOW, which specifies that a minimum of light encryption is required. LOW corresponds to 64-bit DRDA encryption.
- HIGH, which specifies that strong encryption is required. HIGH corresponds to SSL encryption.

The following table summarizes when a trusted context can be used depending on the encryption that is used by the existing connection. If the trusted context cannot be used for the connection, a warning is returned.

Table 110. Summary of when trusted context can be used by an existing connection

Encryption that is used by the existing connection	Value of the ENCRYPTION clause for the trusted context	Can the trusted context be used for the connection?
No encryption	NONE	Yes
No encryption	LOW	No
No encryption	HIGH	No
Low encryption (64-bit)	NONE	Yes
Low encryption (64-bit)	LOW	Yes
Low encryption (64-bit)	HIGH	No

Table 110. Summary of when trusted context can be used by an existing connection (continued)

Encryption that is used by the existing connection	Value of the ENCRYPTION clause for the trusted context	Can the trusted context be used for the connection?
High encryption (128-bit)	NONE	Yes
High encryption (128-bit)	LOW	Yes
High encryption (128-bit)	HIGH	Yes

JOBNAME *jobname-value*

Specifies the z/OS job name or started task name (depending on the source of the address space) for local applications. The JOBNAME attribute can be specified multiple times, but each *jobname-value* must be unique.

jobname-value specifies a string constant that contains the value that is associated with the JOBNAME trust attribute. *jobname-value* is an EBCDIC 8 byte value that specifies the job name or the started task name. The value must be left justified within the string constant. The last character in the name can be a wildcard character (*) if the first character is an alphabetic character. If the job name ends with a wildcard, any job names that begin with the specified characters are considered for establishing the trusted connection.

The following table lists possible values for the job name depending on the source of the address space.

Table 111. Job name for local connection

Source of the address space	Job name
RRSAF	Job name or started task name
TSO	TSO logon ID
BATCH	Job name on JOB statement

SERVAUTH *servauth-value*

Specifies the name of a resource in the RACF SERVAUTH class. This resource is the network access security zone name that contains the IP address of the connection that is used to communicate with DB2. The SERVAUTH attribute can be specified multiple times but each *servauth-value* must be unique.

servauth-value specifies a string constant that contains the value that is associated with the SERVAUTH trust attribute. *servauth-value* is an EBCDIC 64 byte RACF SERVAUTH CLASS resource name. *servauth-value* must be left justified in the string constant. No validation of *servauth-value* is done at the time the CREATE TRUSTED CONTEXT statement is processed.

WITH USE FOR

Specifies who can use a trusted connection that is based on the specified trusted context.

authorization-name

Specifies that the trusted connection can be used by the specified *authorization-name*. This is the DB2 primary authorization ID. The *authorization-name* must not be specified more than one time in the WITH USE FOR clause.

ROLE *role-name*

Specifies that *role-name* is the role that is used when a trusted

connection is used by the specified *authorization-name*. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the user overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*

Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by the specified *authorization-name*. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the user overrides any default security label that is associated with the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION

Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION

Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION

Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user. If a trusted connection is established locally, the authentication token is the password that is provided by the CONNECT statement with the USER and USING clauses. If the trusted connection is established from a remote client, the authentication token can be one of the following tokens:

- password
- RACF Passticket
- Kerberos token

EXTERNAL SECURITY PROFILE *profile-name*

Specifies that the trusted connection can be used by the DB2 primary authorization IDs that are permitted to use the specified *profile-name* in RACF. The *profile-name* must not be specified more than one time in the **WITH USE FOR** clause.

ROLE *role-name*

Specifies that *role-name* is the role that is used when a trusted connection is used by any authorization ID permitted to use the specified *profile-name* in RACF. The *role-name* must identify a role that exists at the current server. The role that is explicitly specified for the profile overrides any default role that is associated with the trusted context.

SECURITY LABEL *seclabel-name*

Specifies that *seclabel-name* is the security label to use for multilevel security verification when the trusted connection is used by any authorization ID that is permitted to use the specified *profile-name* in RACF. The *seclabel-name* must be one of the RACF SECLABEL values that is defined for the user. The security label that is explicitly specified for the profile overrides any default security label that is associated with the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION

Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION

Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION

Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user. If a trusted connection is established locally, the authentication token is the password that is provided by the CONNECT statement with the USER and USING clauses. If the trusted connection is established from a remote client, the authentication token can be one of the following tokens:

- password
- RACF Passticket
- Kerberos token

PUBLIC

Specifies that a trusted connection that is based on the specified trusted context can be used by any user. All users that are using a trusted connection that is defined with PUBLIC use the privileges that are associated with the default role for the associated trusted context. If the default role is not defined for the trusted context, there is no role associated with the users that use a trusted connection that is based on the specified trusted context.

If the default security label for the trusted context is defined, all users that are using the trusted context must have the security label defined as one of the RACF SECLABEL values for the user. The default security label is used for multilevel security verification with all users that are using the trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION

Specifies whether use of the trusted connection requires authentication of the user.

WITHOUT AUTHENTICATION

Specifies that use of a trusted connection by the user does not require authentication. WITHOUT AUTHENTICATION is the default.

WITH AUTHENTICATION

Specifies that use of a trusted connection requires the authentication token with the authorization ID to authenticate the user. If a trusted connection is established locally, the authentication token is the password that is provided by the CONNECT statement with the USER and USING clauses. If the trusted connection is established from a remote client, the authentication token can be one of the following tokens:

- password
- RACF Passticket
- Kerberos token

Notes

Owner privileges: There are no specific privileges on a trusted context.

Order of precedence for users of a trusted connection: The specifications for a user are determined in the following order of precedence:

- *authorization-name*
- **EXTERNAL SECURITY PROFILE** *profile-name*
- **PUBLIC**

For example, assume that a trusted context is defined with use for JOE WITH AUTHENTICATION, EXTERNAL SECURITY PROFILE SPROFILE WITHOUT AUTHENTICATION, and PUBLIC WITH AUTHENTICATION. Users JOE and SAM are permitted to use the RACF PROFILE SPROFILE. If the trusted connection is used by JOE, authentication is required. If the trusted connection is used by SAM, authentication is not required. However, if user SALLY uses the trusted connection, authentication is required.

User-clause SYSTEM AUTHID considerations: If the *authorization-name* that is specified in the SYSTEM AUTHID clause is the same as the *authorization-name* that is specified in the user-clause *authorization-name*, the role or the security label that is specified for *authorization-name* takes precedence over the default value. The value that is specified for the *profile-name*, is permitted to use the profile. If the authorization name that is specified in the SYSTEM AUTHID clause is permitted to use one of the profile names and is not defined in *authorization-name*, the role or the security label that is specified for that *profile-name* takes precedence over the default value.

If authentication is required for SYSTEM AUTHID, either by specification of the AUTHENTICATION clause in the *user-clause* or by setting the value of the TCP/IP Already Verified subsystem parameter to NO, the authentication requirement takes precedence when establishing a remote trusted connection. For example, if *authorization-name* is the same as the authorization name that is specified for SYSTEM AUTHID and the WITHOUT AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to NO, an authentication token is required for SYSTEM AUTHID when the remote trusted connection is established. If *authorization-name* is the SYSTEM AUTHID and the WITH AUTHENTICATION clause is specified, but the TCP/IP Already Verified subsystem parameter is set to YES, an authentication token is still required for SYSTEM AUTHID.

Specifying a role in the definition of a trusted context: The definition of a trusted context can designate a role for a specific authorization ID, and a default role for use for an authorization ID for which a specific role has not been specified in the definition of the trusted context. This role can be used with a trusted connection that is based on the trusted context, but it does not make the role available outside of a trusted connection that is based on the trusted context. When an SQL statement that is not a CREATE, GRANT, or REVOKE statement is issued using a trusted connection, the privileges that are held by a role that is in effect for the authorization ID within the definition of the associated trusted context are considered in addition to other privileges that are directly held by the authorization ID of the statement. The CREATE, GRANT, and REVOKE statements only consider the privileges of the role that is in effect for the trusted connection, or the authorization ID of the statement if a role is not in effect for the trusted connection. If ROLE AS OBJECT OWNER is in effect for a trusted connection, the

role that is in effect for the authorization ID for the trusted connection becomes the owner of any object that is created while using the trusted connection.

When a newly created trusted context takes effect: The newly created trusted context takes effect after the CREATE TRUSTED CONTEXT statement is committed. If the CREATE TRUSTED CONTEXT statement results in an error or is rolled back, no trusted context is created.

Examples

Example 1: The following statement creates a trusted context called CTX1, which is based on a connection and can only be used by users JOE and SAM. Authentication information is required for JOE to use the trusted connection. The trusted context specifies a default role called CTXROLE. However, when JOE uses the trusted connection, the default role is overridden by the user role, ROLE1. When SAM uses the trusted connection, SAM uses the default role. CTX1 is enabled when it is created.

```
CREATE TRUSTED CONTEXT CTX1
  BASED UPON CONNECTION USING SYSTEM AUTHID ADMF001
  ATTRIBUTES (ADDRESS '9.30.131.203',
              ENCRYPTION 'LOW')
  DEFAULT ROLE CTXROLE
  ENABLE
  WITH USE FOR SAM, JOE ROLE ROLE1 WITH AUTHENTICATION;
```

Example 2: The following statement creates a trusted context, CTX2, for a started task, WASPROD. CTX2 is based on a connection, can be used by user SALLY, specifies a default role CTXROLE, and is enabled when it is created. SALLY uses the default role that is associated with the trusted context.

```
CREATE TRUSTED CONTEXT CTX2
  BASED UPON CONNECTION USING SYSTEM AUTHID ADMF002
  ATTRIBUTES (JOBNAME 'WASPROD')
  DEFAULT ROLE CTXROLE WITH ROLE AS OBJECT OWNER AND QUALIFIER
  ENABLE
  WITH USE FOR SALLY;
```

CREATE TYPE

The CREATE TYPE statement defines a distinct type, which is a data type that a user defines. A distinct type must be based on one of the built-in data types.

Successful execution of the statement also generates:

- A function to cast between the distinct type and its source type
- A function to cast between the source type and its distinct type
- As appropriate, support for the use of comparison operators with the distinct type

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege on the schema
- SYSADM or SYSCTRL authority

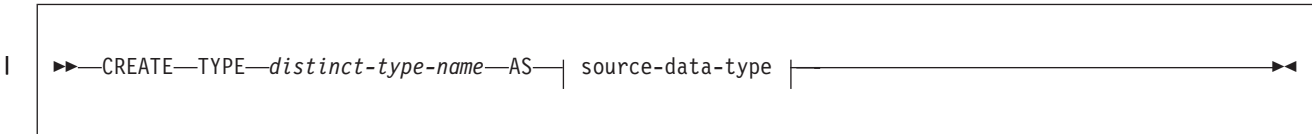
The authorization ID that matches the schema name implicitly has the CREATEIN privilege on the schema.

| **Privilege set:** If the statement is embedded in an application program, the
| privilege set is the privileges that are held by the owner of the plan or package. If
| the owner is a role, the implicit schema match does not apply and this role needs
| to include one of the previously listed conditions.

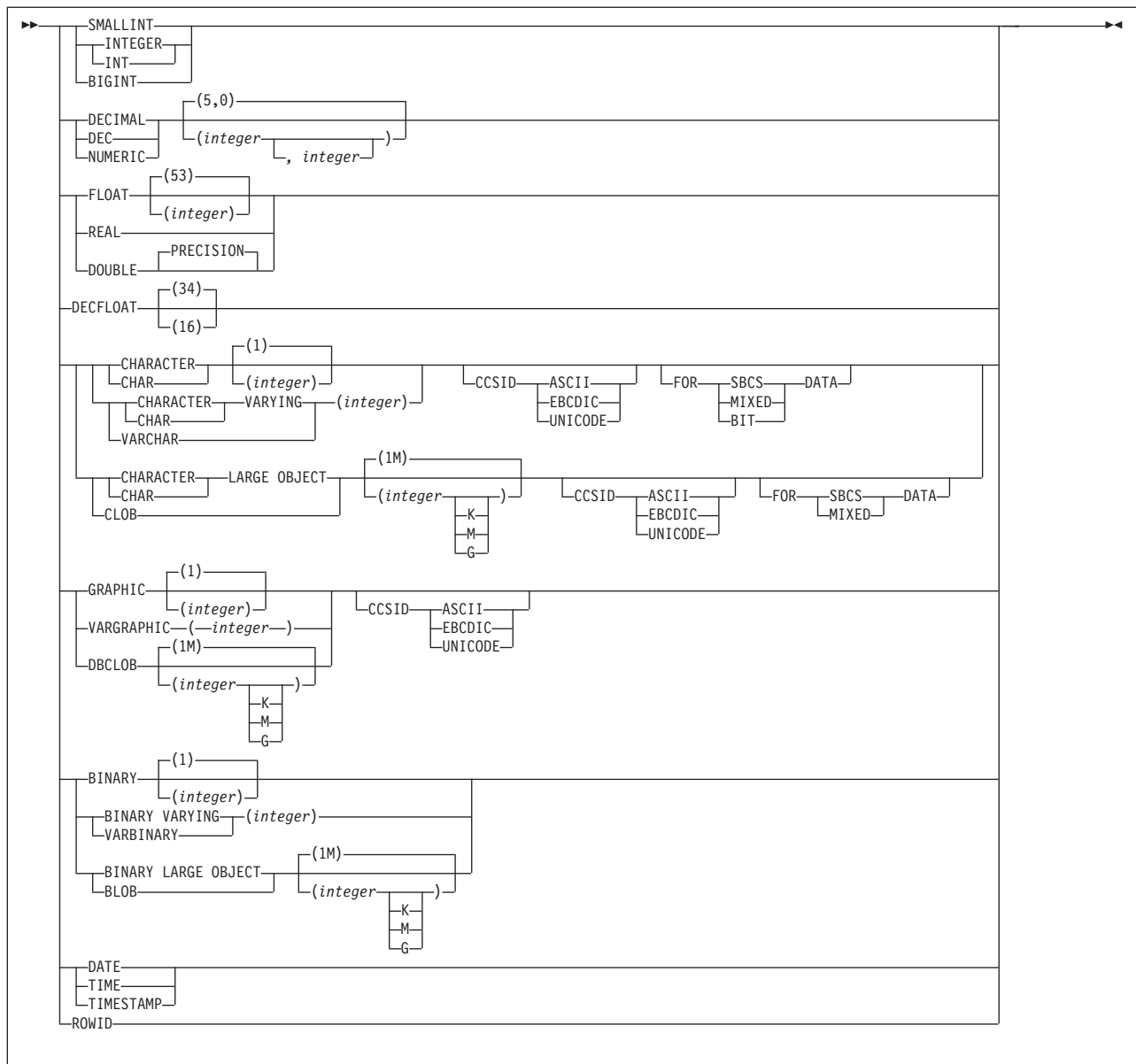
| If the statement is dynamically prepared and is not running in a trusted context for
| which the ROLE AS OBJECT OWNER clause is specified, the privilege set is the
| set of privileges that are held by the SQL authorization ID of the process. The
| specified distinct type name can include a schema name (a qualifier). If the schema
| name is not the same as the SQL authorization ID of the process, one of the
| following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Syntax



source-data-type



Description

distinct-type-name

Names the distinct type. The name, including the implicit or explicit qualifier, must not identify a distinct type that exists at the current server.

- The unqualified form of *distinct-type-name* must not be the name of a built-in data type, BOOLEAN, or any of following system-reserved keywords even if you specify them as delimited identifiers:

ALL	LIKE	UNIQUE
AND	MATCH	UNKNOWN
ANY	NOT	=
BETWEEN	NULL	≠
DISTINCT	ONLY	<
EXCEPT	OR	≤
EXISTS	OVERLAPS	≠
FALSE	SIMILAR	>

	FOR	SOME	>=
	FROM	TABLE	>
	IN	TRUE	<>
	IS	TYPE	

- | • The qualified form of *distinct-type-name* is an SQL identifier (the schema name) followed by a period and an SQL identifier.

| The schema name can be 'SYSTOOLS' if the user who executes the CREATE statement has SYSADM or SYSCTRL privilege. Otherwise, the schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

source-data-type

Specifies the data type that is used as the basis for the internal representation of the distinct type. The data type must be a built-in data type. For more information on built-in data types, see built-in-type.

| If the distinct type is based on a character or graphic string data type, the FOR clause indicates the subtype. If you do not specify the FOR clause, the distinct type is defined with the default subtype. For ASCII or EBCDIC data, the default is SBCS when the value of field MIXED DATA on installation panel DSNTIPF is NO. The default is MIXED when the value is YES. For UNICODE character data, the default subtype is mixed.

If the distinct type is based on a string data type, the CCSID clause indicates whether the encoding scheme of the data is ASCII, EBCDIC or UNICODE. If you do not specify CCSID ASCII, CCSID EBCDIC, or UNICODE, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Notes

| **Owner privileges:** The owner of the distinct type is authorized to define columns, parameters, or variables with the distinct type (USAGE privilege) with the ability to grant these privileges to others. See “GRANT (type or JAR file privileges)” on page 1359. The owner is also authorized to invoke the generated cast function (EXECUTE privilege; see “GRANT (function or procedure privileges)” on page 1340). The owner is given the USAGE and EXECUTE privileges with the GRANT option. For more information about ownership of the object, see “Authorization, privileges, and object ownership” on page 60.

Source data types with DBCS or mixed data: When the implicit or explicit encoding scheme is ASCII or EBCDIC and the source data type is graphic or a character type is MIXED DATA, then the value of field FOR MIXED DATA on installation panel DSNTIPF must be YES; otherwise, an error occurs.

Generated cast functions: The successful execution of the CREATE TYPE statement causes DB2 to generate the following cast functions:

- A function to convert from the distinct type to its source data type
- A function to convert from the source data type to the distinct type
- A function to cast from a data type *A* to distinct type *DT*, where *A* is promotable to the source data type *S* of distinct type *DT*

For some source data types, DB2 supports an additional function to convert from:

- INTEGER to the distinct type if the source type is SMALLINT
- VARCHAR to the distinct type if the source type is CHAR
- VARGRAPHIC to the distinct type if the source type is GRAPHIC

- VARBINARY to the distinct type if the source type is BINARY
- DOUBLE to the distinct type if the source type is REAL

The cast functions are created as if the following statements were executed:

```
CREATE FUNCTION source-type-name (distinct-type-name)
  RETURNS source-type-name ...
CREATE FUNCTION distinct-type-name (source-type-name)
  RETURNS distinct-type-name ...
```

Even if you specified a length, precision, or scale for the source data type in the CREATE TYPE statement, the name of the cast function that converts from the distinct type to the source type is simply the name of the source data type. The data type of the value that the cast function returns includes any length, precision, or scale values that you specified for the source data type. (See Table 112 on page 1181 for details.)

The name of the cast function that converts from the source type to the distinct type is the name of the distinct type. The input parameter of the cast function has the same data type as the source data type, including the length, precision, and scale.

For example, assume that a distinct type named T_SHOESIZE is created with the following statement:

```
CREATE TYPE CLAIRE.T_SHOESIZE AS VARCHAR(2)
```

When the statement is executed, DB2 also generates the following cast functions. VARCHAR converts from the distinct type to the source type, and T_SHOESIZE converts from the source type to the distinct type.

```
FUNCTION CLAIRE.VARCHAR (CLAIRE.T_SHOESIZE) RETURNS SYSIBM.VARCHAR (2)
FUNCTION CLAIRE.T_SHOESIZE (SYSIBM.VARCHAR (2)) RETURNS CLAIRE.T_SHOESIZE
```

Notice that function VARCHAR returns a value with a data type of VARCHAR(2) and that function T_SHOESIZE has an input parameter with a data type of VARCHAR(2).

The schema of the generated cast functions is the same as the schema of the distinct type. No other function with the same name and function signature must already exist in the database.

In the preceding example, if T_SHOESIZE had been sourced on a SMALLINT, CHAR, or GRAPHIC data type instead of a VARCHAR data type, another cast function would have been generated in addition to the two functions to cast between the distinct type and the source data type. For example, assume that T_SHOESIZE is created with this statement:

```
CREATE TYPE CLAIRE.T_SHOESIZE AS CHAR(2)
```

When the statement is executed, DB2 generates these cast functions:

```
FUNCTION CLAIRE.CHAR (CLAIRE.T_SHOESIZE) RETURNS SYSIBM.CHAR (2)
FUNCTION CLAIRE.T_SHOESIZE (SYSIBM.CHAR (2)) RETURNS CLAIRE.T_SHOESIZE
FUNCTION CLAIRE.T_SHOESIZE (SYSIBM.VARCHAR (2)) RETURNS CLAIRE.T_SHOESIZE
```

Notice that the third function enables the casting of a VARCHAR(2) to T_SHOESIZE. This additional function is created to enable casting a constant, such as 'AB', directly to the distinct type. Without the additional function, you would have to first cast 'AB', which has a data type of VARCHAR, to a data type of CHAR and then cast it to the distinct type.

You cannot explicitly drop a generated cast function. The cast functions that are generated for a distinct type are implicitly dropped when the distinct type is dropped with the DROP statement.

For each built-in data type that can be the source data type for a distinct type, Table 112 gives the names of the generated cast functions, the data types of the input parameters, and the data types of the values that the functions returns.

Table 112. CAST functions on distinct types

Source type name	Function name	Parameter-type	Return-type
SMALLINT	<i>distinct-type-name</i>	SMALLINT	<i>distinct-type-name</i>
	<i>distinct-type-name</i>	INTEGER	<i>distinct-type-name</i>
	SMALLINT	<i>distinct-type-name</i>	SMALLINT
INTEGER	<i>distinct-type-name</i>	INTEGER	<i>distinct-type-name</i>
	INTEGER	<i>distinct-type-name</i>	INTEGER
BIGINT	<i>distinct-type-name</i>	BIGINT	<i>distinct-type-name</i>
	BIGINT	<i>distinct-type-name</i>	BIGINT
DECIMAL	<i>distinct-type-name</i>	DECIMAL (p,s)	<i>distinct-type-name</i>
	DECIMAL	<i>distinct-type-name</i>	DECIMAL (p,s)
NUMERIC	<i>distinct-type-name</i>	DECIMAL (p,s)	<i>distinct-type-name</i>
	DECIMAL	<i>distinct-type-name</i>	DECIMAL (p,s)
REAL	<i>distinct-type-name</i>	REAL	<i>distinct-type-name</i>
	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	REAL	<i>distinct-type-name</i>	REAL
DECFLOAT	<i>distinct-type-name</i>	DECFLOAT(n)	DECFLOAT(n)
	DECFLOAT	<i>distinct-type-name</i>	DECFLOAT(n)
FLOAT(n) where $n \leq 21$	<i>distinct-type-name</i>	REAL	<i>distinct-type-name</i>
	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	REAL	<i>distinct-type-name</i>	REAL
FLOAT(n) where $n > 21$	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	DOUBLE	<i>distinct-type-name</i>	DOUBLE
FLOAT	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	DOUBLE	<i>distinct-type-name</i>	DOUBLE
DOUBLE	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	DOUBLE	<i>distinct-type-name</i>	DOUBLE
DOUBLE PRECISION	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	<i>distinct-type-name</i>	CHAR (n)	<i>distinct-type-name</i>
	CHAR	<i>distinct-type-name</i>	CHAR (n)
	<i>distinct-type-name</i>	VARCHAR (n)	<i>distinct-type-name</i>
	DOUBLE	<i>distinct-type-name</i>	DOUBLE
CHAR CHARACTER	<i>distinct-type-name</i>	CHAR (n)	<i>distinct-type-name</i>
	CHAR	<i>distinct-type-name</i>	CHAR (n)
	<i>distinct-type-name</i>	VARCHAR (n)	<i>distinct-type-name</i>

Table 112. CAST functions on distinct types (continued)

Source type name	Function name	Parameter-type	Return-type
VARCHAR CHARACTER VARYING CHAR VARYING	<i>distinct-type-name</i>	VARCHAR (n)	<i>distinct-type-name</i>
	VARCHAR	<i>distinct-type-name</i>	VARCHAR (n)
CLOB	<i>distinct-type-name</i>	CLOB (n)	<i>distinct-type-name</i>
	CLOB	<i>distinct-type-name</i>	CLOB (n)
GRAPHIC	<i>distinct-type-name</i>	GRAPHIC (n)	<i>distinct-type-name</i>
	GRAPHIC	<i>distinct-type-name</i>	GRAPHIC (n)
	<i>distinct-type-name</i>	VARGRAPHIC (n)	<i>distinct-type-name</i>
VARGRAPHIC	<i>distinct-type-name</i>	VARGRAPHIC (n)	<i>distinct-type-name</i>
	VARGRAPHIC	<i>distinct-type-name</i>	VARGRAPHIC (n)
DBCLOB	<i>distinct-type-name</i>	DBCLOB (n)	<i>distinct-type-name</i>
	DBCLOB	<i>distinct-type-name</i>	DBCLOB (n)
BINARY	<i>distinct-type-name</i>	BINARY(n)	<i>distinct-type-name</i>
	BINARY	<i>distinct-type-name</i>	BINARY(n)
	<i>distinct-type-name</i>	VARBINARY(n)	<i>distinct-type-name</i>
VARBINARY	<i>distinct-type-name</i>	VARBINARY(n)	<i>distinct-type-name</i>
	VARBINARY	<i>distinct-type-name</i>	VARBINARY(n)
BLOB	<i>distinct-type-name</i>	BLOB (n)	<i>distinct-type-name</i>
	BLOB	<i>distinct-type-name</i>	BLOB (n)
DATE	<i>distinct-type-name</i>	DATE	<i>distinct-type-name</i>
	DATE	<i>distinct-type-name</i>	DATE
TIME	<i>distinct-type-name</i>	TIME	<i>distinct-type-name</i>
	TIME	<i>distinct-type-name</i>	TIME
TIMESTAMP	<i>distinct-type-name</i>	TIMESTAMP	<i>distinct-type-name</i>
	TIMESTAMP	<i>distinct-type-name</i>	TIMESTAMP
ROWID	<i>distinct-type-name</i>	ROWID	<i>distinct-type-name</i>
	ROWID	<i>distinct-type-name</i>	ROWID

Notes: NUMERIC and FLOAT are not recommended when creating a distinct type for a portable application. Use DECIMAL and DOUBLE (or REAL) instead.

Built-in functions: When a distinct type is defined, the built-in functions (such as AVG, MAX, and LENGTH) are not automatically supported for the distinct type. You can use a built-in function on a distinct type only after a sourced user-defined function, which is based on the built-in function, has been created for the distinct type. For information on defining sourced user-defined functions, see “CREATE FUNCTION (sourced)” on page 958.

Syntax alternatives: The WITH COMPARISONS clause, which specifies that system-generated comparison operators are to be created for comparing two instances of the distinct type, can be specified as the last clause of the statement. Use WITH COMPARISONS only if it is required for compatibility with other

| products in the DB2 family. If the source data type is either BLOB, CLOB, or
| DBCLOB and WITH COMPARISONS is specified, a warning occurs as in previous
| releases.

| To provide compatibility with previous releases of DB2 or other products in the
| DB2 family, DB2 supports the following clauses:

- DISTINCT TYPE as a synonym for TYPE

| .

Examples

Example 1: Create a distinct type named SHOESIZE that is based on an INTEGER data type.

```
CREATE TYPE SHOESIZE AS INTEGER;
```

The successful execution of this statement also generates two cast functions. Function INTEGER(SHOESIZE) returns a value with data type INTEGER, and function SHOESIZE(INTEGER) returns a value with distinct type SHOESIZE.

Example 2: Create a distinct type named MILES that is based on a DOUBLE data type.

```
CREATE TYPE MILES AS DOUBLE;
```

The successful execution of this statement also generates two cast functions. Function DOUBLE(MILES) returns a value with data type DOUBLE, and function MILES(DOUBLE) returns a value with distinct type MILES.

CREATE VIEW

The CREATE VIEW statement creates a view on tables or views at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

For every table or view identified in the *fullselect*, the privilege set that is defined below must include at least one of the following:

- The SELECT privilege on the table or view
- Ownership of the table or view
- DBADM authority for the database (tables only)
- SYSADM authority
- SYSCTRL authority (catalog tables only)

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Authority requirements depend in part on the choice of the owner of the view. For information on how to choose the owner, see the description of *view-name* in “Description” on page 867.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the application is bound in a trusted context with the ROLE AS OBJECT OWNER clause specified, a role is the owner. Otherwise, an authorization ID is the owner.

- If this privilege set includes SYSADM authority, the owner of the view can be any authorization ID. If that set includes SYSCTRL but not SYSADM authority, the following is true: the owner of the view can be any authorization ID, provided the view does not refer to user tables or views in the first FROM clause of its defining fullselect. (It could refer instead, for example, to catalog tables or views thereof.)

If the view satisfies the rules in the preceding paragraph, and if no errors are present in the CREATE statement, the view is created, even if the owner has no privileges at all on the tables and views identified in the fullselect of the view definition.

- If the privilege set lacks SYSADM and SYSCTRL but includes DBADM authority on at least one of the databases that contains a table from which the view is created, the owner of the view can be any authorization ID if all of the following conditions are true:
 - The value of field DBADM CREATE AUTH was set to YES on panel DSNNTIPP during DB2 installation.
 - The view is not based only on views.

Note: The owner of the view must have the SELECT privilege on all tables and views in the CREATE VIEW statement, or, if the owner does not have the SELECT privilege on a table, the creator must have DBADM authority on the database that contains that table.

- If the privilege set lacks SYSADM, SYSCTRL, and DBADM authority, or if the authorization ID of the application plan or package fails to meet any of the previous conditions, the owner of the view must be the owner of the application plan or package.

If ROLE AS OBJECT OWNER is in effect, the schema qualifier must be the same as the role, unless the role has the CREATEIN privilege on the schema, SYSADM authority, or SYSCTRL authority.

If ROLE AS OBJECT OWNER is not in effect, one of the following rules applies:

- If the privilege set lacks the CREATIN privilege on the schema, SYSADM authority, or SYSCTRL authority, the schema qualifier (implicit or explicit) must be the same as one of the authorization ids of the process.
- If the privilege set includes SYSADM authority or SYSCTRL authority, the schema qualifier can be any valid schema name.

If the statement is dynamically prepared, the following rules apply:

- If the SQL authorization ID of the process has SYSADM authority, the owner of the view can be any authorization ID. If that authorization ID has SYSCTRL but not SYSADM authority, the following is true: the owner of the view can be any authorization ID, provided the view does not refer to user tables or views in the first FROM clause of its defining fullselect. (It could refer instead, for example, to catalog tables or views thereof.)

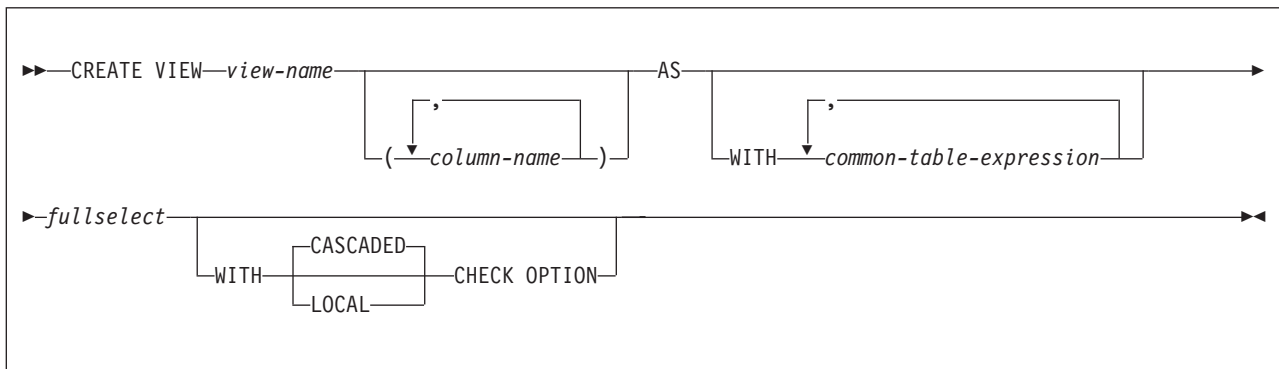
If the view satisfies the rules in the preceding paragraph, and if no errors are present in the CREATE statement, the view is created, even if the owner has no privileges at all on the tables and views identified in the fullselect of the view definition.

- If SQL authorization ID of the process lacks SYSADM and SYSCTRL but includes DBADM authority on at least one of the databases that contains a table from which the view is created, the owner of the view can be different from the SQL authorization ID if all of the following conditions are true:
 - The value of field DBADM CREATE AUTH was set to YES on panel DSNTIPP during DB2 installation.
 - The view is not based only on views.

Note: The owner of the view must have the SELECT privilege on all tables and views in the CREATE VIEW statement, or, if the owner does not have the SELECT privilege on a table, the creator must have DBADM authority on the database that contains that table.

- If the SQL authorization ID of the process lacks SYSADM, SYSCTRL, or DBADM authority, or if the SQL authorization ID of the process fails to meet any of the previous conditions, only the authorization IDs of the process can own the view. In this case, the privilege set is the privileges that are held by the authorization ID selected for ownership.

Syntax



Description

view-name

Names the view. The name, including the implicit or explicit qualifier, must not identify a table, view, alias, or synonym that exists at the current server.

If the name is qualified, the name can be a two-part or three-part name. If a three-part name is used, the first part must match the value of the field DB2 LOCATION NAME of installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.)

column-name, ...

Names the columns in the view. If you specify a list of column names, it must consist of as many names as there are columns in the result table of the fullselect. Each name must be unique and unqualified. If you do not specify a list of column names, the columns of the view inherit the names of the columns of the result table of the fullselect.

You must specify a list of column names if the result table of the fullselect has duplicate column names or an unnamed column (a column derived from a constant, function, or expression that was not given a name by the AS clause). For more details about unnamed columns, see the information about names of result columns under “select-clause” on page 633.

AS Identifies the view definition.

WITH *common-table-expression*

Defines a common table expression for use with the fullselect that follows. For an explanation of common table expression, see “common-table-expression” on page 670.

fullselect

Defines the view. At any time, the view consists of the rows that would result if the fullselect were executed.

fullselect must not refer to any declared temporary tables, host variables, or parameter markers (question marks). In addition, the FROM clause of the *fullselect* must not include a *data-change-table-reference* or a view for which an INSTEAD OF trigger is defined. For an explanation of *fullselect*, see “fullselect” on page 662.

WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTIONS

Specifies that every row that is inserted or updated through the view must

conform to the definition of the view. A row that does not conform to the definition of the view is a row that cannot be retrieved using that view.

The CHECK OPTION clause must not be specified if the view is read-only, includes a subquery, references a function that is not deterministic or has an external action, or if the fullselect of the view refers to a created temporary table. If the CHECK OPTION clause is specified for an updatable view that does not allow inserts, it applies to updates only.

If the CHECK OPTION clause is omitted, the definition of the view is not used in the checking of any insert or update operations that use the view. Some checking might still occur during insert or update operations if the view is directly or indirectly dependent on another view that includes the CHECK OPTION clause. Because the definition of the view is not used, rows might be inserted or updated through the view that do not conform to the definition of the view.

The difference between the two forms of the check option, CASCADED and LOCAL, is meaningful only when a view is dependent on another view. The default is CASCADED. The view on which another view is directly or indirectly defined is an *underlying view*.

CASCADED

Update and insert operations on view V must satisfy the search conditions of view V and all underlying views, regardless of whether the underlying views were defined with a check option. Furthermore, every updatable view that is directly or indirectly defined on view V inherits those search conditions (the search conditions of view V and all underlying views of V) as a constraint on insert or update operations. WITH CASCADED CHECK OPTION must not be specified if a view on which the specified view definition is dependent has an INSTEAD OF trigger defined.

LOCAL

Update and insert operations on view V must satisfy the search conditions of view V and underlying views that are defined with a check option (either WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTION). Furthermore, every updatable view that is directly or indirectly defined on view V inherits those search conditions (the search conditions of view V and all underlying views of V that are defined with a check option) as a constraint on insert or update operations.

The LOCAL form of the CHECK option lets you update or insert rows that do not conform to the search condition of view V. You can perform these operations if the view is directly or indirectly defined on a view that was defined without a check option.

Table 113 on page 1188 illustrates the effect of using the default check option, CASCADED. The information in Table 113 on page 1188 is based on the following views:

- CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10
- CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CASCADED CHECK OPTION
- CREATE VIEW V3 AS SELECT COL1 FROM V2 WHERE COL1 < 100

Table 113. Examples using default check option, *CASCADED*

SQL statement	Description of result
INSERT INTO V1 VALUES(5)	Succeeds because V1 does not have a check option and it is not dependent on any other view that has a check option.
INSERT INTO V2 VALUES(5)	Results in an error because the inserted row does not conform to the search condition of V1 which is implicitly is part of the definition of V2.
INSERT INTO V3 VALUES(5)	Results in an error because the inserted row does not conform to the search condition of V1.
INSERT INTO V3 VALUES(200)	Succeeds even though it does not conform to the definition of V3 (V3 does not have the view check option specified); it does conform to the definition of V2 (which does have the view check option specified).

The difference between *CASCADED* and *LOCAL* is shown best by example. Consider the following updatable views, where x and y represent either *LOCAL* or *CASCADED*:

V1 is defined on Table T0.
V2 is defined on V1 WITH x CHECK OPTION.
V3 is defined on V2.
V4 is defined on V3 WITH y CHECK OPTION.
V5 is defined on V4.

This example shows V1 as an *underlying view* for V2 and V2 as *dependent* on V1.

Table 114 shows the views in which search conditions are checked during an insert or update operation:

Table 114. Views in which search conditions are checked during insert and update operations

View used in INSERT or UPDATE operation	x = <i>LOCAL</i> y = <i>LOCAL</i>	x = <i>CASCADED</i> y = <i>CASCADED</i>	x = <i>LOCAL</i> y = <i>CASCADED</i>	x = <i>CASCADED</i> y = <i>LOCAL</i>
V1	None	None	None	None
V2	V2	V2, V1	V2	V2, V1
V3	V2	V2, V1	V2	V2, V1
V4	V4, V2	V4, V3, V2, V1	V4, V3, V2, V1	V4, V2, V1
V5	V4, V2	V4, V3, V2, V1	V4, V3, V2, V1	V4, V2, V1

Notes

Owner privileges: The owner of a view always acquires the SELECT privilege on the view and the authority to drop the view. If all of the privileges that are required to create the view are held with the GRANT option before the view is created, the owner of the view receives the SELECT privilege with the GRANT option. Otherwise, the owner receives the SELECT privilege without the GRANT option. For example, assume that a view definition also refers to a user-defined function. If the owner's EXECUTE privilege on the user-defined function is held without the GRANT option, the owner acquires the SELECT privilege on the view without the GRANT option.

The owner can also acquire INSERT, UPDATE, and DELETE privileges on the view. Acquiring these privileges is possible if the view is not "read-only", which means a single table of view is identified in the first FROM clause of the fullselect. For each privilege that the owner has on the identified table or view (INSERT, UPDATE, and DELETE) before the new view is created, the owner acquires that privilege on the view. The owner receives the privilege with the GRANT option if the privilege is held on the table or view with the GRANT option. Otherwise, the owner receives the privileges without the GRANT option.

With appropriate DB2 authority, a process can create views for those who have no authority to create the views themselves. The owner of such a view has the SELECT privilege on the view, without the GRANT option, and can drop the view.

For more information on the ownership of an object, see "Authorization, privileges, and object ownership" on page 60.

Authorization for views created for other users: When a process with appropriate authority creates a view for another user that does not have authorization for the underlying table or view, the SELECT privilege for the created view is implicitly granted to the user.

Read-only views: A view is *read-only* if one or more of the following statements is true of its definition:

- The first FROM clause identifies more than one table or view, or identifies a table function, a nested table expression, or a common table expression.
- The first SELECT clause specifies the keyword DISTINCT.
- The outer fullselect contains a GROUP BY clause.
- The outer fullselect contains a HAVING clause.
- The first SELECT clause contains an aggregate function.
- It contains a subquery such that the base object of the outer fullselect, and of the subquery, is the same table.
- The first FROM clause identifies a read-only view.
- The first FROM clause identifies a system-maintained materialized query table.
- The outer fullselect is not a subselect (contains a set operator).

A read-only view cannot be the object of an SQL data change statement or a TRUNCATE statement. A view that includes GROUP BY or HAVING cannot be referred to in a subquery of a basic predicate.

Insertable views: A view is insertable if an INSTEAD OF trigger for the insert operation has been defined for the view, or if at least one column of the view is updatable (independent of an INSTEAD OF trigger for update).

Considerations for implicitly hidden columns: It is possible that the result table of the fullselect will include a column of a base table that is defined as implicitly hidden. This can occur when the implicitly hidden column is explicitly referenced in the fullselect of the view definition. However, the corresponding column of the view does not inherit the implicitly hidden attribute. Columns of a view cannot be defined as hidden.

Testing a view definition: You can test the semantics of your view definition by executing SELECT * FROM *view-name*.

The two forms of a view definition: Both the source and the operational form of a view definition are stored in the DB2 catalog. Those two forms are not necessarily equivalent because the operational form reflects the state that exists when the view is created. For example, consider the following statement:

```
CREATE VIEW V AS SELECT * FROM S;
```

In this example, S is a synonym or alias for A.T, which is a table with columns C1, C2, and C3. The operational form of the view definition is equivalent to:

```
SELECT C1, C2, C3 FROM A.T;
```

Adding columns to A.T using ALTER TABLE and dropping S does not affect the operational form of the view definition. Thus, if columns are added to A.T or if S is redefined, the source form of the view definition can be misleading.

View restrictions: A view definition cannot contain references to remote objects. A view definition cannot map to more than 15 base table instances. A view definition cannot reference a declared global temporary table.

Examples

Example 1: Create the view DSN8910.VPROJRE1. PROJNO, PROJNAME, PROJDEP, RESPEMP, FIRSTNME, MIDINIT, and LASTNAME are column names. The view is a join of tables and is therefore read-only.

```
CREATE VIEW DSN8910.VPROJRE1
  (PROJNO,PROJNAME,PROJDEP,RESPEMP,
   FIRSTNME,MIDINIT,LASTNAME)
AS SELECT ALL
  PROJNO,PROJNAME,DEPTNO,EMPNO,
  FIRSTNME,MIDINIT,LASTNAME
FROM DSN8910.PROJ, DSN8910.EMP
WHERE RESPEMP = EMPNO;
```

In the example, the WHERE clause refers to the column EMPNO, which is contained in one of the base tables but is not part of the view. In general, a column named in the WHERE, GROUP BY, or HAVING clause need not be part of the view.

Example 2: Create the view DSN8910.FIRSTQTR that is the UNION ALL of three fullselects, one for each month of the first quarter of 2000. The common names are SNO, CHARGES, and DATE.

```
CREATE VIEW DSN8910.FIRSTQTR (SNO, CHARGES, DATE) AS
SELECT SNO, CHARGES, DATE
FROM MONTH1
WHERE DATE BETWEEN '01/01/2000' and '01/31/2000'
UNION ALL
SELECT SNO, CHARGES, DATE
FROM MONTH2
WHERE DATE BETWEEN '02/01/2000' and '02/29/2000'
UNION ALL
SELECT SNO, CHARGES, DATE
FROM MONTH3
WHERE DATE BETWEEN '03/01/2000' and '03/31/2000';
```

DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java.

Authorization

For each table or view identified in the SELECT statement of the cursor, the privilege set must include at least one of the following:

- The SELECT privilege
- Ownership of the object
- DBADM authority for the corresponding database (tables only)
- SYSADM authority
- SYSCTRL authority (catalog tables only)

If the *select-statement* contains an SQL data change statement, the authorization requirements of that statement also apply to the DECLARE CURSOR statement.

The SELECT statement of the cursor is one of the following:

- The prepared select statement identified by *statement-name*
- The specified *select-statement*

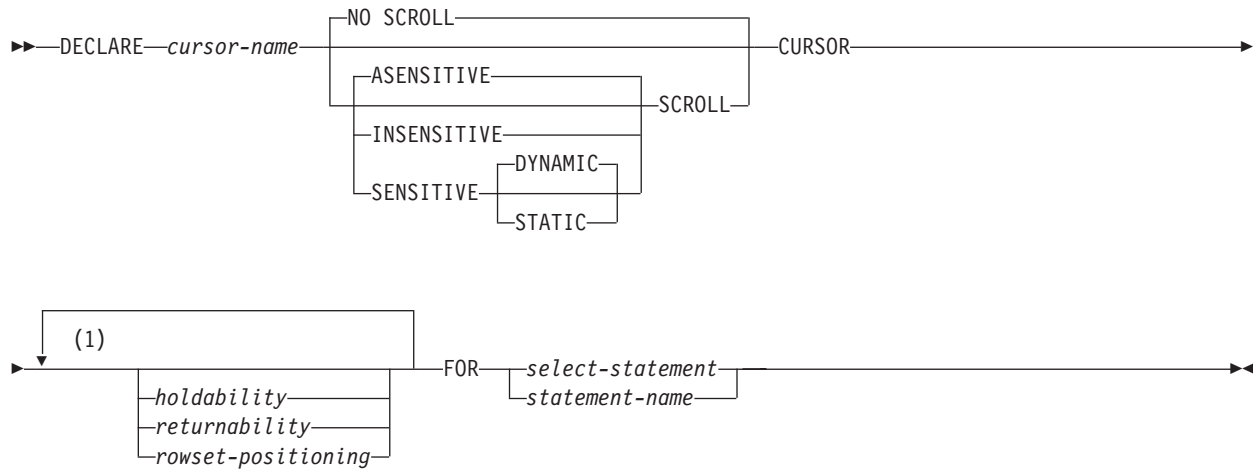
If *statement-name* is specified:

- The privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)
- The authorization check is performed when the SELECT statement is prepared.
- The cursor cannot be opened unless the SELECT statement is successfully prepared.

If *select-statement* is specified:

- The privilege set consists of the privileges that are held by the authorization ID of the owner of the plan or package.
- If the plan or package is bound with VALIDATE(BIND), the authorization check is performed at bind time, and the bind is unsuccessful if any required privilege does not exist.
- If the plan or package is bound with VALIDATE(RUN), an authorization check is performed at bind time, but all required privileges need not exist at that time. If all privileges exist at bind time, no authorization checking is performed when the cursor is opened. If any privilege does not exist at bind time, an authorization check is performed the first time the cursor is opened within a unit of work. The OPEN is unsuccessful if any required privilege does not exist.

Syntax



Notes:

- 1 The same clause must not be specified more than once.

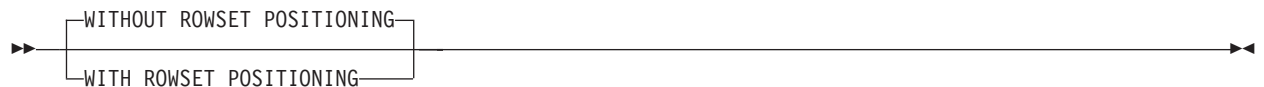
holdability:



returnability:



rowset-positioning:



Description

cursor-name

Names the cursor. The name must not identify a cursor that has already been

declared in the source program. The name is usually VARCHAR(128); however, if the cursor is defined WITH RETURN, the name is limited to VARCHAR(30).

NO SCROLL or SCROLL

Specifies whether the cursor is scrollable or not scrollable.

NO SCROLL

Specifies that the cursor is not scrollable. This is the default.

SCROLL

Specifies that the cursor is scrollable. For a scrollable cursor, whether the cursor has sensitivity to inserts, updates, or deletes depends on the cursor sensitivity option in effect for the cursor. If a sensitivity option is not specified, ASENSITIVE is the default.

ASENSITIVE

Specifies that the cursor should be as sensitive as possible. This is the default.

A cursor that defined as ASENSITIVE will be either insensitive or sensitive dynamic; it will not be sensitive static. For information about how the effective sensitivity of the cursor is returned to the application with the GET DIAGNOSTICS statement or in the SQLCA, see "OPEN" on page 1400.

INSENSITIVE

Specifies that the cursor does not have sensitivity to inserts, updates, or deletes that are made to the rows underlying the result table. As a result, the size of the result table, the order of the rows, and the values for each row do not change after the cursor is opened. In addition, the cursor is read-only. The SELECT statement or *attribute-string* of the PREPARE statement cannot contain a FOR UPDATE clause, and the cursor cannot be used for positioned updates or deletes.

SENSITIVE

Specifies that the cursor has sensitivity to changes that are made to the database after the result table is materialized. The cursor is always sensitive to updates and deletes that are made using the cursor (that is, positioned updates and deletes using the same cursor). When the current value of a row no longer satisfies the *select-statement* or *statement-name*, that row is no longer visible through the cursor. When a row of the result table is deleted from the underlying base table, the row is no longer visible through the cursor.

If DB2 cannot make changes visible to the cursor, then an error is issued at bind time for OPEN CURSOR. DB2 cannot make changes visible to the cursor when the cursor implicitly becomes read-only. For example, when the result table must be materialized, as when the FROM clause of the SELECT statement contains more than one table or view. The current list of conditions that result in an implicit read-only cursor can be found in Read-only cursors.

The default is DYNAMIC.

DYNAMIC

Specifies that the result table of the cursor is dynamic, meaning that the size of the result table might change after the cursor is opened as rows are inserted into or deleted from the underlying table, and the order of the rows might change. Rows that are inserted, deleted, or updated by statements that are executed by the same application process as the cursor are visible to the cursor

immediately. Rows that are inserted, deleted, or updated by statements that are executed by other application processes are visible only after the statements are committed. If a column for an ORDER BY clause is updated via a cursor or any means outside the process, the next FETCH statement behaves as if the updated row was deleted and re-inserted into the result table at its correct location. At the time of a positioned update, the cursor is positioned before the next row of the original location and there is no current row, making the row appear to have moved.

If a SENSITIVE DYNAMIC cursor is not possible, an error is returned. For example, if a temporary table is needed (such as for processing a FETCH FIRST n ROWS ONLY clause), an error is returned. The SELECT statement of a cursor that is defined as SENSITIVE DYNAMIC cannot contain an SQL data change statement.

STATIC

Specifies that the size of the result table and the order of the rows do not change after the cursor is opened. Rows inserted into the underlying table are not added to the result table regardless of how the rows are inserted. Rows in the result table do not move if columns in the ORDER BY clause are updated in rows that have already been materialized. Positioned updates and deletes are allowed if the result table is updatable. The SELECT statement of a cursor that is defined as SENSITIVE STATIC cannot contain an SQL data change statement.

A STATIC cursor has visibility to changes made by *this* cursor using positioned updates or deletes. Committed changes made outside this cursor are visible with the SENSITIVE option of the FETCH statement. A FETCH SENSITIVE can result in a *hole* in the result table (that is, a difference between the result table and its underlying base table). If an updated row in the base table of a cursor no longer satisfies the predicate of its SELECT statement, an update hole occurs in the result table. If a row of a cursor was deleted in the base table, a delete hole occurs in the result table. When a FETCH SENSITIVE detects an update hole, no data is returned (a warning is issued), and the cursor is left positioned on the update hole. When a FETCH SENSITIVE detects a delete hole, no data is returned (a warning is issued), and the cursor is left positioned on the delete hole.

Updates through a cursor result in an automatic re-fetch of the row. This re-fetch means that updates can create a hole themselves. The re-fetched row also reflects changes as a result of triggers updating the same row. It is important to reflect these changes to maintain the consistency of data in the row.

Using a function that is not deterministic (built-in or user-defined) in the WHERE clause of the *select-statement* or *statement-name* of a SENSITIVE STATIC cursor can cause misleading results. This situation occurs because DB2 constructs a temporary result table and retrieves rows from this table for FETCH INSENSITIVE statements. When DB2 processes a FETCH SENSITIVE statement, rows are fetched from the underlying table and predicates are re-evaluated. Using a function that is not deterministic can yield a

different result on each FETCH SENSITIVE of the same row, which could also result in the row no longer being considered a match.

A FETCH INSENSITIVE on a SENSITIVE STATIC SCROLL cursor is not sensitive to changes made outside the cursor, unless a previous FETCH SENSITIVE has already refreshed that row; however, positioned updates and delete changes with the cursor are visible.

STATIC cursors are insensitive to insertions.

WITHOUT HOLD or WITH HOLD

Specifies whether the cursor should be prevented from being closed as a consequence of a commit operation.

WITHOUT HOLD

Does not prevent the cursor from being closed as a consequence of a commit operation. This is the default.

WITH HOLD

Prevents the cursor from being closed as a consequence of a commit operation. A cursor declared with WITH HOLD is closed at commit time if one of the following is true:

- The connection associated with the cursor is in the release pending status.
- The bind option DISCONNECT(AUTOMATIC) is in effect.
- The environment is one in which the option WITH HOLD is ignored.

When WITH HOLD is specified, a commit operation commits all of the changes in the current unit of work. For example, with a non-scrollable cursor, an initial FETCH statement is needed after a COMMIT statement to position the cursor on the row that follows the row that the cursor was positioned on before the commit operation.

WITH HOLD has no effect on an SQL data change statement within a SELECT statement. When a COMMIT is issued, the changes caused by the SQL data change statement are committed, regardless of whether or not the cursor is declared WITH HOLD.

All cursors are implicitly closed by a connect (Type 1) or rollback operation. A cursor is also implicitly closed by a commit operation if WITH HOLD is ignored or not specified.

Cursors that are declared with WITH HOLD in CICS or in IMS non-message-driven programs will not be closed by a rollback operation if the cursor was opened in a previous unit of work and no changes have been made to the database in the current unit of work. The cursor cannot be closed because CICS and IMS do not broadcast the rollback request to DB2 for a null unit of work.

If a cursor is closed before the commit operation, the effect is the same as if the cursor was declared without the option WITH HOLD.

WITH HOLD is ignored in IMS message driven programs (MPP, IFP, and message-driven BMP). WITH HOLD maintains the cursor position in a CICS pseudo-conversational program until the end-of-task (EOT).

For details on restrictions that apply to declaring cursors with WITH HOLD, see *DB2 Application Programming and SQL Guide*.

WITHOUT RETURN or WITH RETURN TO CALLER

Specifies the intended use of the result table of the cursor. The default is WITHOUT RETURN, except in Java procedures, where the default is WITH RETURN TO CALLER.

WITHOUT RETURN

Specifies that the result table of the cursor is not intended to be used as a result set that will be returned from the program or procedure.

WITH RETURN TO CALLER

Specifies that the result table of the cursor is intended to be used as a result set that will be returned from the program or procedure to the caller. Specifying TO CALLER is optional.

WITH RETURN TO CALLER is relevant when the SQL CALL statement is used to invoke a procedure that either contains the DECLARE CURSOR statement, or directly or indirectly invokes a program or procedure that contains the DECLARE CURSOR statement. In other cases, the precompiler might accept the clause, but the clause has no effect.

When a cursor that is declared using the WITH RETURN TO CALLER clause remains open at the end of a program or procedure, that cursor defines a result set from the program or procedure. Use the CLOSE statement to close cursors that are not intended to be a result set from the program or procedure. Although DB2 will automatically close any cursors that are not declared using WITH RETURN TO CALLER, the use of the CLOSE statement is recommended to increase the portability of applications.

For non-scrollable cursors, the result set consists of all rows from the current cursor position to the end of the result table. For scrollable cursors, the result set consists of all rows of the result table.

The caller is the program or procedure that executed the SQL CALL statement that either invokes the procedure that contains the DECLARE CURSOR statement, or directly or indirectly invokes the program that contains the DECLARE CURSOR statement. For example, if the caller is a procedure, the result set is returned to the procedure. If the caller is a client application, the result set is returned to the client application.

rowset-positioning

Specifies whether multiple rows of data can be accessed as a rowset on a single FETCH statement for the cursor. The default is WITHOUT ROWSET POSITIONING.

WITHOUT ROWSET POSITIONING

Specifies that the cursor can be used only with row-positioned FETCH statements. The cursor is to return a single row for each FETCH statement and the FOR *n* ROWS clause cannot be specified on a FETCH statement for this cursor. WITHOUT ROWSET POSITIONING or single row access refers to how data is fetched from the database engine. For remote access, data might be blocked and returned to the client in blocks.

WITH ROWSET POSITIONING

Specifies that the cursor can be used with either row-positioned or rowset-positioned FETCH statements. This cursor can be used to return either a single row or multiple rows, as a rowset, with a single FETCH statement. ROWSET POSITIONING refers to how data is fetched from the database engine. For remote access, if any row qualifies, at least 1 row is returned as a rowset. The size of the rowset depends on the number of

rows specified on the FETCH statement and on the number of rows that qualify. Data might be blocked and returned to the client in blocks.

select-statement

Specifies the result table of the cursor. See “select-statement” on page 669 for an explanation of *select-statement*.

The *select-statement* must not include parameter markers (except for REXX), but can include references to host variables. In host languages, other than REXX, the declarations of the host variables must precede the DECLARE CURSOR statement in the source program. In REXX, parameter markers must be used in place of host variables and the statement must be prepared.

The USING clause of the OPEN statement can be used to specify host variables that will override the values of the host variables or parameter markers that are specified as part of the statement in the DECLARE CURSOR statement.

The *select-statement* of the cursor must not contain an SQL data change statement if the cursor is defined as SENSITIVE DYNAMIC or SENSITIVE STATIC.

statement-name

Identifies the prepared *select-statement* that specifies the result table of the cursor whenever the cursor is opened. The *statement-name* must not be identical to a statement name specified in another DECLARE CURSOR statement of the source program. For an explanation of prepared SELECT statements, see “PREPARE” on page 1405.

The prepared *select-statement* of the cursor must not contain an SQL data change statement if the cursor is defined as SENSITIVE DYNAMIC or SENSITIVE STATIC.

Notes

A cursor in the open state designates a result table and a position relative to the rows of that table. The table is the result table specified by the SELECT statement of the cursor.

Read-only cursors: If the result table is *read-only*, the cursor is *read-only*. The cursor that references a view with instead of triggers are read-only since positioned UPDATE and positioned DELETE statements are not allowed using those cursors. The result table is *read-only* if one or more of the following statements is true about the SELECT statement of the cursor:

- The first FROM clause identifies or contains any of the following:
 - More than one table or view
 - A catalog table with no updatable columns
 - A read-only view
 - A nested table expression
 - A table function
 - A system-maintained materialized query table
- The first SELECT clause specifies the keyword DISTINCT, contains an aggregate function, or uses both
- It contains an SQL data change statement
- The outer subselect contains a GROUP BY clause, a HAVING clause, or both clauses

- It contains a subquery such that the base object of the outer subselect, and of the subquery, is the same table
- Any of the following operators or clauses are specified:
 - A set operator
 - An ORDER BY clause (except when the cursor is declared as SENSITIVE STATIC scrollable)
 - A FOR READ ONLY clause
- It is executed with isolation level UR and a FOR UPDATE clause is not specified.

If the result table is not *read-only*, the cursor can be used to update or delete the underlying rows of the result table.

Work file database requirement for static scrollable cursors: To use a static scrollable cursor, you must first create a work file database and at least one table space with a 32KB page size in this database because a static scrollable cursor requires a temporary table for its result table while the cursor is open. DB2 chooses a table space to use for the temporary result table. Dynamic scrollable cursors do not require a declared temporary table.

Cursors in COBOL and Fortran programs: In COBOL and Fortran source programs, the DECLARE CURSOR statement must precede all statements that explicitly refer to the cursor by name. This rule does not necessarily apply to the other host languages because the precompiler provides a two-pass option for these languages. This rule applies to other host languages if the two-pass option is not used.

Cursors in REXX: If host variables are used in a DECLARE CURSOR statement within a REXX procedure, the DECLARE CURSOR statement must be the object of a PREPARE and EXECUTE.

Scope of a cursor: The scope of *cursor-name* is the source program in which it is defined; that is, the application program submitted to the precompiler. Thus, you can only refer to a cursor by statements that are precompiled with the cursor declaration. For example, a COBOL program called from another program cannot use a cursor that was opened by the calling program. Furthermore, a cursor defined in a Fortran subprogram can only be referred to in that subprogram. Cursors that specify WITH RETURN in a procedure and are left open are returned as result sets.

Although the scope of a cursor is the program in which it is declared, each package (or DBRM of a plan) created from the program includes a separate instance of the cursor, and more than one instance of the cursor can be used in the same execution of the program. For example, assume a program is precompiled with the CONNECT(2) option and its DBRM is used to create a package at location X and a package at location Y. The program contains the following SQL statements:

```
DECLARE C CURSOR FOR ...
CONNECT TO X
OPEN C
FETCH C INTO ...
CONNECT TO Y
OPEN C
FETCH C INTO ...
```


The second OPEN C statement does not cause an error because it refers to a different instance of cursor C. The same notion applies to a single location if the packages are in different collections.

A SELECT statement is evaluated at the time the cursor is opened. If the same cursor is opened, closed, and then opened again, the results can be different. If the SELECT statement of the cursor contains CURRENT DATE, CURRENT TIME or CURRENT TIMESTAMP, all references to these special registers yields the same respective datetime value on each FETCH operation. The value is determined when the cursor is opened. Multiple cursors using the same SELECT statement can be opened concurrently. They are each considered independent activities.

Blocking of data: To process data more efficiently, DB2 might block data for read-only cursors. If a cursor is not going to be used in a positioned UPDATE or positioned DELETE statement, define the cursor as FOR READ ONLY.

Positioned deletes and isolation level UR: Specify FOR UPDATE if you want to use the cursor for a positioned DELETE and the isolation level is UR because of a BIND option. In this case, the isolation level is CS.

Returning a result set from a stored procedure: A cursor that is declared in a stored procedure returns a result set when all of the following conditions are true:

- The cursor is declared with the WITH RETURN option. In a distributed environment, blocks of each result set of the cursor's data are returned with the CALL statement reply.
- The cursor is left open after exiting from the stored procedure. A cursor declared with the SCROLL option must be left positioned *before* the first row before exiting from the stored procedure.
- The cursor is declared with the WITH HOLD option if the stored procedure is defined to commit on return.

The result set is the set of all rows after the current position of the cursor after exiting the stored procedure. The result set is assumed to be read-only. If that same procedure is reinvoked, open result set cursors for a stored procedure at a given site are automatically closed by the database management system.

Scrollable cursors specified with user-defined functions: A row can be fetched more than once with a scrollable cursor. Therefore, if a scrollable cursor is defined with a function that is not deterministic in the select list of the cursor, a row can be fetched multiple times with different results for each fetch. (However, the value of a function that is not deterministic in the WHERE clause of a scrollable cursor is captured when the cursor is opened and remains unchanged until the cursor is closed.) Similarly, if a scrollable cursor is defined with a user-defined function with external action, the action is executed with every fetch.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Declare C1 as the cursor of a query to retrieve data from the table DSN8910.DEPT. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8910.DEPT
  WHERE ADMRDEPT = 'A00';
```

Example 2: Declare C1 as the cursor of a query to retrieve data from the table DSN8810.DEPT. Assume that the data will be updated later with a searched update and should be locked when the query executes. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8910.DEPT
  WHERE ADMRDEPT = 'A00'
  FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS;
```

Example 3: Declare C2 as the cursor for a statement named STMT2.

```
EXEC SQL DECLARE C2 CURSOR FOR STMT2;
```

Example 4: Declare C3 as the cursor for a query to be used in positioned updates of the table DSN8910.EMP. Allow the completed updates to be committed from time to time without closing the cursor.

```
EXEC SQL DECLARE C3 CURSOR WITH HOLD FOR
  SELECT * FROM DSN8910.EMP
  FOR UPDATE OF WORKDEPT, PHONENO, JOB, EDLEVEL, SALARY;
```

Instead of specifying which columns should be updated, you could use a FOR UPDATE clause without the names of the columns to indicate that all updatable columns are updated.

Example 5: In stored procedure SP1, declare C4 as the cursor for a query of the table DSN8910.PROJ. Enable the cursor to return a result set to the caller of SP1, which performs a commit on return.

```
EXEC SQL DECLARE C4 CURSOR WITH HOLD WITH RETURN FOR
  SELECT PROJNO, PROJNAME
  FROM DSN8910.PROJ
  WHERE DEPTNO = 'A01';
```

Example 6: In the following example, the DECLARE CURSOR statement associates the cursor name C5 with the results of the SELECT and specifies that the cursor is scrollable. C5 allows positioned updates and deletes because the result table can be updated.

```
EXEC SQL DECLARE C5 SENSITIVE STATIC SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8910.DEPT
  WHERE ADMRDEPT = 'A00';
```

Example 7: In the following example, the DECLARE CURSOR statement associates the cursor name C6 with the results of the SELECT and specifies that the cursor is scrollable.

```
EXEC SQL DECLARE C6 INSENSITIVE SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8910.DEPT
  WHERE DEPTNO;
```

Example 8: The following example illustrates how an application program might use dynamic scrollable cursors. First create and populate a table.

```
CREATE TABLE ORDER
  (ORDERNUM INTEGER,
   CUSTNUM INTEGER,
   CUSTNAME VARCHAR(20),
   ORDERDATE CHAR(8),
   ORDERAMT DECIMAL(8,3),
   COMMENTS VARCHAR(20));
```

Populate the table by inserting or loading about 500 rows.

```
EXEC SQL DECLARE CURSOR ORDERSROLL
SENSITIVE DYNAMIC SCROLL FOR
SELECT ORDERNUM, CUSTNAME, ORDERAMT, ORDERDATE FROM ORDER
WHERE ORDERAMT > 1000
FOR UPDATE OF COMMENTS;
```

Open the scrollable cursor.

```
OPEN CURSOR ORDERSROLL;
```

Fetch forward from the scrollable cursor.

```
-- Loop-to-fill-screen
-- do 10 times
  FETCH FROM ORDERSROLL INTO :HV1, :HV2, :HV3, :HV4;
-- end
```

Fetch RELATIVE from the scrollable cursor.

```
-- Skip-forward-100-rows
  FETCH RELATIVE +100
  FROM ORDERSROLL INTO :HV1, :HV2, :HV3, :HV4;
-- Skip-backward-50-rows
  FETCH RELATIVE -50
  FROM ORDERSROLL INTO :HV1, :HV2, :HV3, :HV4;
```

Fetch ABSOLUTE from the scrollable cursor.

```
-- Re-read-the-third-row
  FETCH ABSOLUTE +3
  FROM ORDERSROLL INTO :HV1, :HV2, :HV3, :HV4;
```

Fetch RELATIVE from scrollable cursor.

```
-- Read-the-third-row-from current position
  FETCH SENSITIVE RELATIVE +3
  FROM ORDERSROLL INTO :HV1, :HV2, :HV3, :HV4;
```

Do a positioned update through the scrollable cursor.

```
-- Update-the-current-row
  UPDATE ORDER SET COMMENTS = "Expedite"
  WHERE CURRENT OF ORDERSROLL;
```

Close the scrollable cursor.

```
CLOSE CURSOR ORDERSROLL;
```

Example 9: Declare C1 as the cursor of a query to retrieve a rowset from the table DEPT. The prepared statement is MYCURSOR.

```
EXEC SQL DECLARE C1 CURSOR
  WITH ROWSET POSITIONING FOR MYCURSOR;
```

DECLARE GLOBAL TEMPORARY TABLE

The DECLARE GLOBAL TEMPORARY TABLE statement defines a declared temporary table for the current application process. The declared temporary table resides in the work file database and its description does not appear in the system catalog. It is not persistent and cannot be shared with other application processes. Each application process that defines a declared temporary table of the same name has its own unique description and instance of the temporary table. When the application process terminates, the temporary table is dropped.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

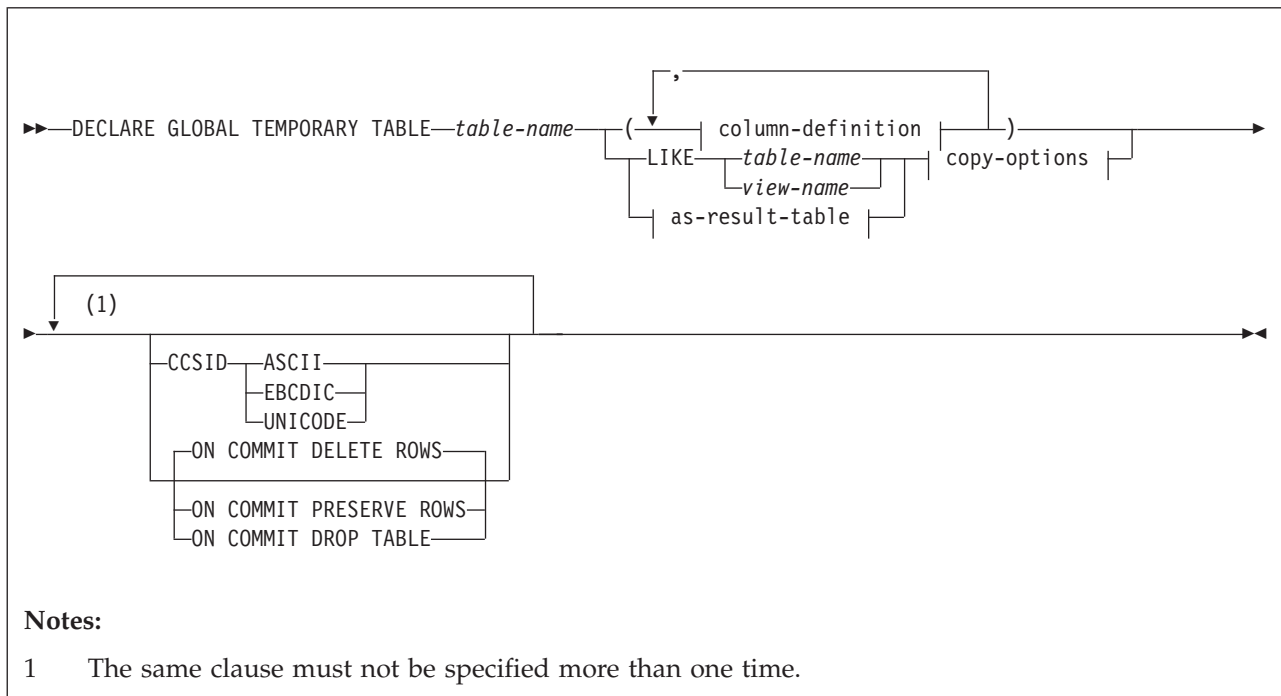
None are required, unless the LIKE clause is specified when additional privileges might be required.

PUBLIC implicitly has the following privileges without GRANT authority for declared temporary tables:

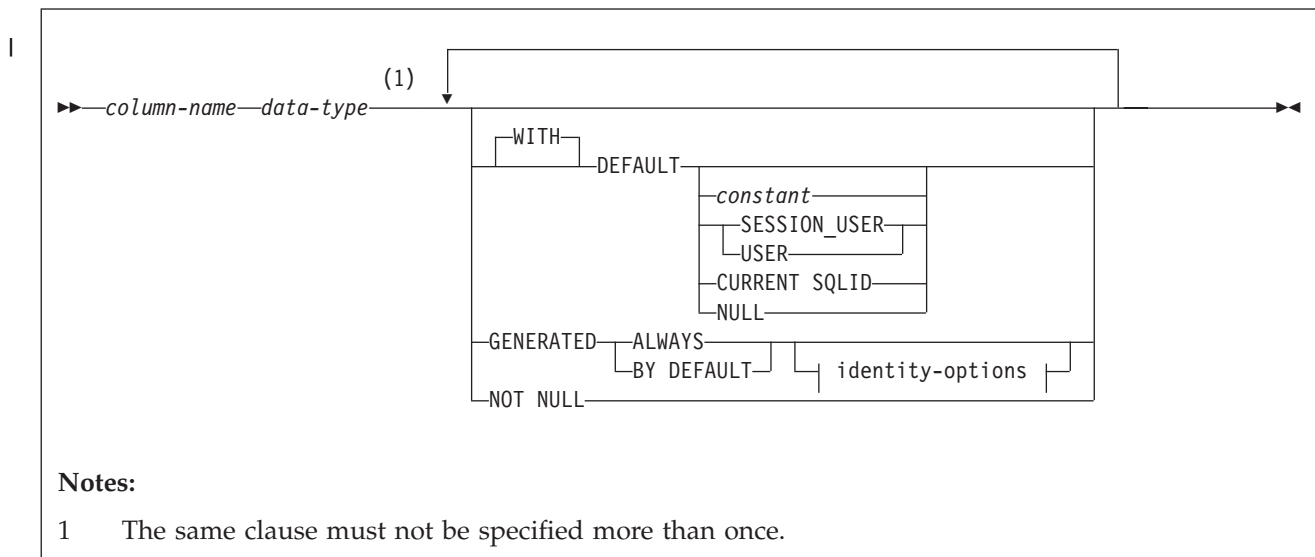
- The CREATETAB privilege to define a declared temporary table in the database that is defined AS WORKFILE, which is the database for declared temporary tables.
- The USE privilege to use the table spaces in the database that is defined as WORKFILE.
- All table privileges on the table and authority to drop the table. (Table privileges for a declared temporary table cannot be granted or revoked.)

These implicit privileges are not recorded in the DB2 catalog and cannot be revoked.

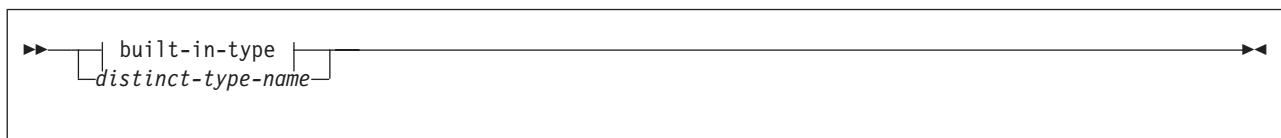
Syntax



column-definition:

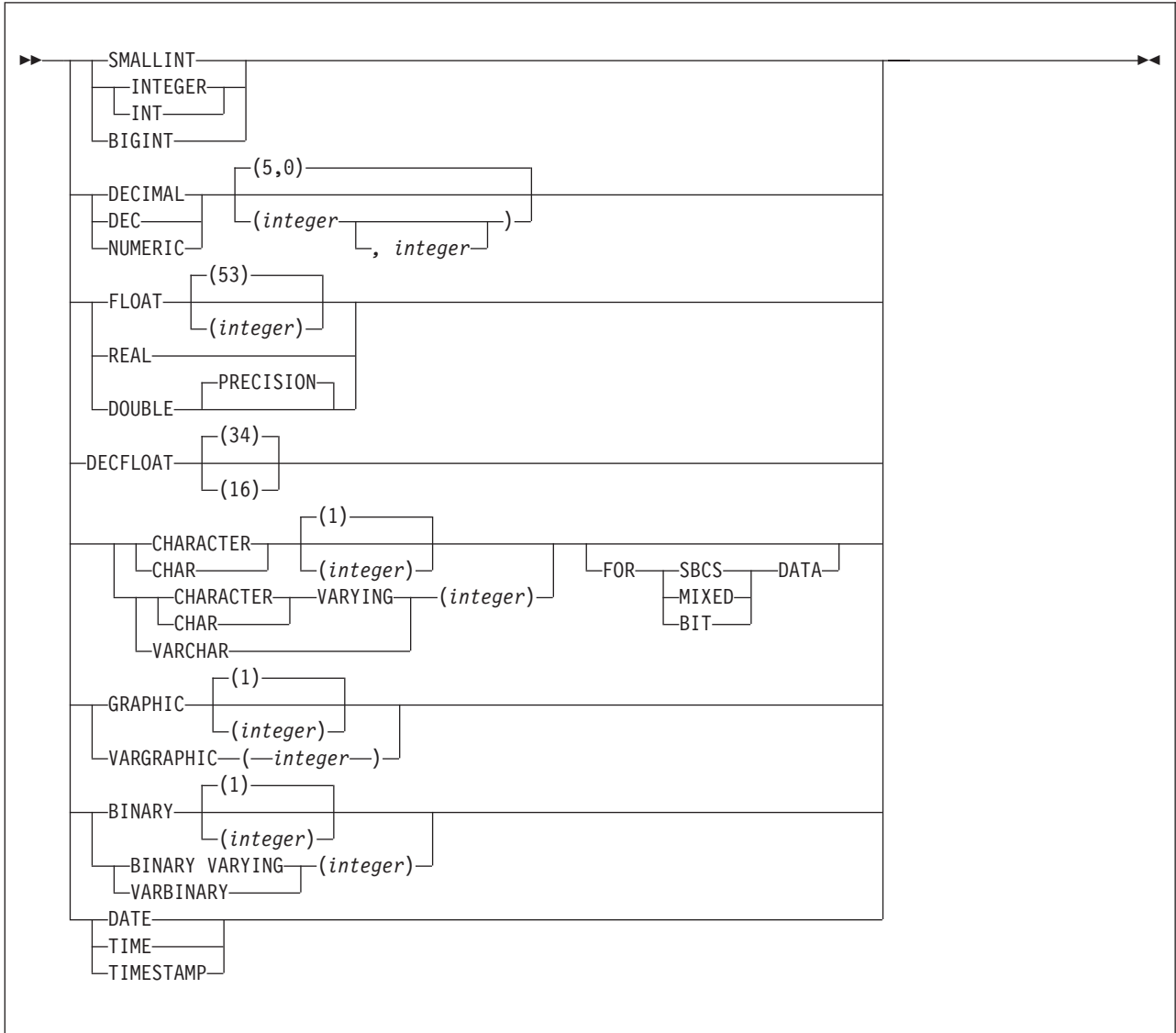


data-type:



built-in-type:

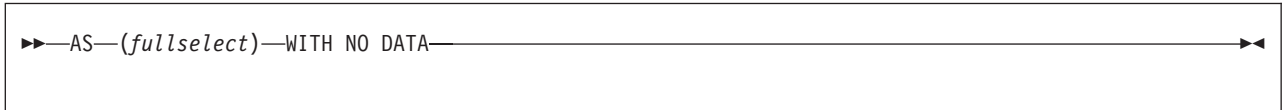
|



|

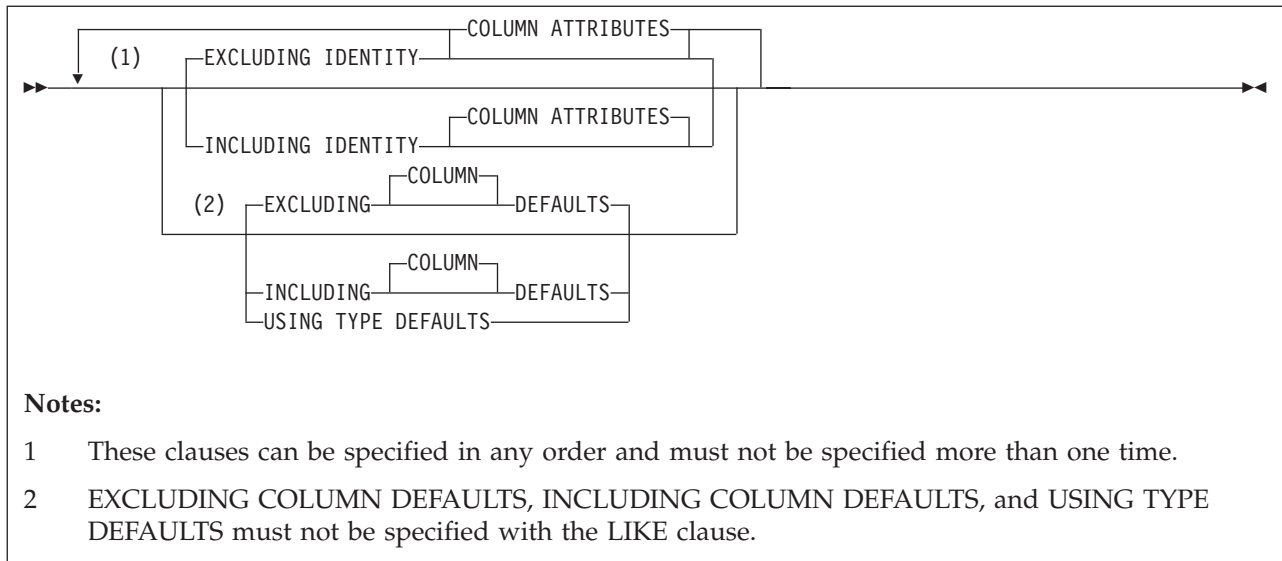
| as-result-table:

|

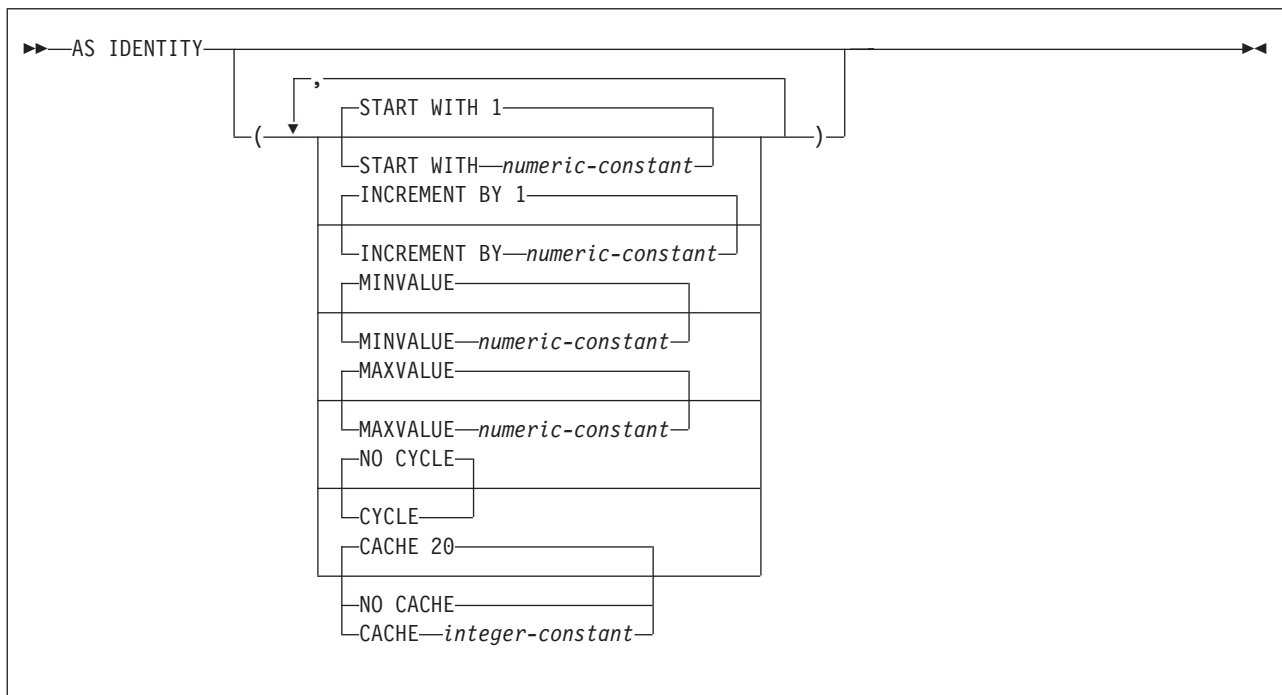


|

copy-options:



identity-options:



Description

table-name

Names the temporary table. The qualifier, if specified explicitly, must be SESSION. If the qualifier is not specified, it is implicitly defined to be SESSION.

If a table, view, synonym, or alias already exists with the same name and an implicit or explicit qualifier of SESSION:

- The declared temporary table is still defined with *SESSION.table-name*. An error is not issued because the resolution of a declared temporary table name does not include the persistent and shared names in the DB2 catalog tables.
- Any references to *SESSION.table-name* will resolve to the declared temporary table rather than to any existing *SESSION.table-name* whose definition is persistent and is in the DB2 catalog tables.

column-definition

Defines the attributes of a column for each instance of the table. The number of columns defined must not exceed 750. The maximum record size must not exceed 32683 bytes. The maximum row size must not exceed 32675 bytes (8 bytes less than the maximum record size).

column-name

Names the column. The name must not be qualified and must not be the same as the name of another column in the table.

data-type

Specifies the data type of the column. The data type can be any built-in data type that can be specified for the CREATE TABLE statement except for a LOB (BLOB, CLOB, and DBCLOB), ROWID, or XML type. The FOR *subtype* DATA clause can be specified as part of *data-type*. For more information on the data types and the rules that apply to them, see built-in-type.

DEFAULT

Specifies a default value for the column. This clause must not be specified more than once in the same *column-definition*.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

If DEFAULT is specified without a value after it, the default value of the column depends on the data type of the column, as follows:

Data type

Default value

Numeric

0

Fixed-length character string

A string of blanks

Fixed-length graphic string

A string of blanks

Fixed-length binary string

Hexadecimal zeros

Varying-length string

A string of length 0

Date CURRENT DATE

Time CURRENT TIME

Timestamp

CURRENT TIMESTAMP

A default value other than the one that is listed above can be specified in one of the following forms:

constant

Specifies a constant as the default value for the column. The value of the constant must conform to the rules for assigning that value to the column. A hexadecimal graphic string constant (GX) cannot be specified.

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register at the time of an insert or update operation or LOAD as the default value for the column. The data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the SESSION_USER special register.

CURRENT SQLID

Specifies the value of the SQL authorization ID of the process at the time of an SQL data change statement or LOAD as the default value for the column. The data type of the column must be a character string with a length attribute greater than or equal to the length attribute of the CURRENT SQLID special register.

NULL

Specifies null as the default value for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same *column-definition*.

GENERATED

Specifies that DB2 generates values for the column. GENERATED must be specified if the column is to be considered an IDENTITY column. If DEFAULT is specified for the column for an update operation, DB2 generates a value for both GENERATED ALWAYS and GENERATED BY DEFAULT.

ALWAYS

Specifies that DB2 always generates a value for the column when a row is inserted into the table.

BY DEFAULT

Specifies that DB2 generates a value for the column when a row is inserted into the table unless a value is specified. BY DEFAULT is the recommended value only when you are using data propagation.

Defining a column as GENERATED BY DEFAULT does not necessarily guarantee the uniqueness of the values. To ensure uniqueness of the values, define a unique, single-column index on the column.

AS IDENTITY

Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero).

An identity column is implicitly NOT NULL. An identity column cannot have a DEFAULT clause. For the descriptions of the identity attributes, see the description of the AS IDENTITY clause in "CREATE TABLE" on page 1079.

NOT NULL

Specifies that the column cannot contain nulls. Omission of NOT NULL indicates that the column can contain nulls.

LIKE table-name or view-name

Specifies that the columns of the table have the same name, data type, and nullability attributes as the columns of the identified table or view. If a table is identified, the column default attributes are also defined by that table. The name specified must identify a table, view, synonym, or alias that exists at the current server. The identified table must not be an auxiliary table.

The LIKE clause must not be specified in a Common Criteria environment.

The privilege set must include the SELECT privilege on the identified table or view.

This clause is similar to the LIKE clause on CREATE TABLE, but it has the following differences:

- If LIKE results in a column having a LOB data type, a ROWID data type, or distinct type, the DECLARE GLOBAL TEMPORARY TABLE statement fails.
- In addition to these data type restrictions, if any column has any other attribute value that is not allowed in a declared temporary table, that attribute value is ignored. The corresponding column in the new temporary table has the default value for that attribute unless otherwise indicated.

When the identified object is a table, the column name, data type, nullability, and default attributes are determined from the columns of the specified table; any identity column attributes are inherited only if the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified.

as-result-table

Specifies that the table definition is based on the column definitions from the result of a query expression.

The behavior of these column attributes is controlled with the INCLUDING or USING TYPE DEFAULTS clauses, which are defined below.

AS (*fullselect*)

Specifies an implicit definition of n columns for the declared global temporary table, where n is the number of columns that would result from the *fullselect*. The columns of the new table are defined by the columns that result from the *fullselect*. Every select list element must have a unique name. The AS clause can be used in the select-clause to provide unique names. The implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of *fullselect*. A column of the new table that corresponds to an implicitly hidden column of a base table referenced in the *fullselect* is not considered hidden in the new table.

The **AS** (*fullselect*) clause must not be specified in a Common Criteria environment.

The result table of the *fullselect* must not contain a column that has a LOB data type, a ROWID data type, an XML data type or a distinct type.

If *fullselect* results in other column attributes that are not applicable for a declared temporary table, those attributes are ignored in the implicit definition for the declared temporary table.

If *fullselect* results in a row change timestamp column, the corresponding column of the new table inherits only the data type of the row change timestamp column. The new column is not considered as a generated column.

The *fullselect* must not refer to host variables or include parameter markers (question marks). The outermost SELECT list of the *fullselect* must not reference data that is encoded with different CCSID sets.

WITH NO DATA

Specifies that the *fullselect* is not executed. You can use the INSERT INTO statement with the same *fullselect* specified in the AS clause to populate the declared temporary table with the set of rows from the result table of the *fullselect*.

copy-options

Specifies whether identity column attributes and column defaults are inherited from the definition of the source of the result table.

EXCLUDING IDENTITY COLUMN ATTRIBUTES or INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies whether identity column attributes are inherited from the columns resulting from the *fullselect*, *table-name*, or *view-name*.

EXCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table does not inherit the identity attributes of the columns resulting from the *fullselect*, *table-name*, or *view-name*.

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table inherits the identity attributes, if any, of the columns resulting from the *fullselect* or *table-name*. In general, the identity attributes are copied if the element of the corresponding column in the table or *fullselect* is the name of a table column that directly or indirectly maps to the name of a base table column that is an identity column.

If the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified with the AS fullselect clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the *fullselect* includes multiple instances of an identity column name (that is, selecting the same column more than once).
- The select list of the *fullselect* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *fullselect* includes a set operation.

If INCLUDING IDENTITY COLUMN ATTRIBUTES is not specified, the new table will not have an identity column.

If the LIKE clause identifies a view, INCLUDING IDENTITY COLUMN ATTRIBUTES must not be specified.

EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, or USING TYPE DEFAULTS

Specifies whether the table inherits the default values of the columns of the fullselect. EXCLUDING COLUMN DEFAULTS, INCLUDING COLUMN DEFAULTS, and USING TYPE DEFAULTS must not be specified if the LIKE clause is specified.

EXCLUDING COLUMN DEFAULTS

Specifies that the table does not inherit the default values of the columns of the fullselect. The default values of the column of the new table are either null or there are no default values. If the column can be null, the default is the null value. If the column cannot be null, there is no default value, and an error occurs if a value is not provided for a column on an insert operation for the new table.

INCLUDING COLUMN DEFAULTS

Specifies that the table inherits the default values of the columns of the fullselect. A default value is the value that is assigned to the column when a value is not specified on an insert operation or LOAD.

Columns resulting from the fullselect that are not updatable will not have a default defined in the corresponding column of the created table.

USING TYPE DEFAULTS

Specifies that the default values for the declared temporary table depend on the data type of the columns that result from *fullselect*, as follows:

Data type	Default value
Numeric	0
Fixed-length character string	Blanks
Fixed-length graphic string	Blanks
Fixed-length binary string	Hexadecimal zeros
Varying-length string	A string of length 0
Date	CURRENT DATE
Time	CURRENT TIME
Timestamp	CURRENT TIMESTAMP

CCSID *encoding-scheme*

Specifies the encoding scheme for string data that is stored in the table. For declared temporary tables, the encoding scheme for the data cannot be specified for the table space or database, and all data in one table space or the database need not use the same encoding scheme. Because there can be only one work file database for all declared temporary tables for each DB2 member, there can be a mixture of encoding schemes in both the database and each table space.

For the creation of temporary tables, the CCSID clause can be specified whether or not the LIKE clause is specified. If the CCSID clause is specified, the encoding scheme of the new table is the scheme that is specified in the CCSID clause. If the CCSID clause is not specified, the encoding scheme of the new table is the same as the scheme for the table specified in the LIKE clause or as the scheme for the table identified by the AS (fullselect) clause.

ASCII Specifies that the data is encoded by using the ASCII CCSIDs of the server.

EBCDIC

Specifies that the data is encoded by using the EBCDIC CCSIDs of the server.

UNICODE

Specifies that the data is encoded by using the UNICODE CCSIDs of the server.

An error occurs if the CCSIDs for the encoding scheme have not been defined. Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed, graphic, or UNICODE data is used.

ON COMMIT

Specifies what happens to the table for a commit operation. The default is ON COMMIT DELETE ROWS.

DELETE ROWS

Specifies that all of the rows of the table are deleted if there is no open cursor that is defined as WITH HOLD that references the table.

PRESERVE ROWS

Specifies that all of the rows of the table are preserved. Thread reuse capability is not available to any application process or thread that contains, at its most recent commit, an active declared temporary table that was defined with the ON COMMIT PRESERVE ROWS clause.

DROP TABLE

Specifies that the table is implicitly dropped at commit if there is no open cursor that is defined as WITH HOLD that references the table. If there is an open cursor defined as WITH HOLD on the table at commit, the rows are preserved.

Notes

Instantiation, scope, and termination: For the following explanations, P denotes an application process, and T is a declared temporary table executed in P:

- An empty instance of T is created when a DECLARE GLOBAL TEMPORARY TABLE statement is executed in P.
- Any SQL statement in P can reference T, and any of those references to T in P is a reference to that same instance of T. ()

If a DECLARE GLOBAL TEMPORARY statement is specified within an SQL PL compound statement, the scope of the declared temporary table is the application process and not just the compound statement. A declared temporary table cannot be defined multiple times by the same name in other compound statements in that application process, unless the table has been dropped explicitly.

- If T was declared at a remote server, the reference to T must use the same DB2 connection that was used to declare T and that connection must not have been terminated after T was declared. When the connection to the database server at which T was declared terminates, T is dropped.
- If T was defined with the ON COMMIT DELETE ROWS clause specified implicitly or explicitly, when a commit operation terminates a unit of work in P and there is no open WITH HOLD cursor in P that is dependent on T, the commit deletes all rows from T.
- If T is defined with the ON COMMIT DROP TABLE clause, when a commit operation terminates a unit of work in P and no program in P has a WITH HOLD cursor open that is dependent on T, the commit includes the operation DROP TABLE T.
- When a rollback operation terminates a unit of work or savepoint in P, and that unit of work or savepoint includes the declaration of SESSION.T, the changes to table T are undone.

When a rollback operation terminates a unit of work or savepoint in P, and that unit of work or savepoint includes the declaration of SESSION.T, the rollback drops table T.

When a rollback operation terminates a unit of work or savepoint in P, and that unit of work or savepoint includes the drop of the declaration of declared temporary table SESSION.T, the rollback undoes the drop of table T.

- When the application process that declared T terminates, T is dropped.

Privileges: When a declared temporary table is defined, PUBLIC is implicitly granted all table privileges on the table and authority to drop the table. These implicit privileges are not recorded in the DB2 catalog and cannot be revoked. This enables any SQL statement in the application process to reference a declared temporary table that has already been defined in that application process.

Referring to a declared temporary table in other SQL statements: Many SQL statements support declared temporary tables. To refer to a declared temporary table in an SQL statement other than DECLARE GLOBAL TEMPORARY TABLE, you must qualify the table name with SESSION. You can either specify SESSION explicitly in the table name or use the QUALIFIER bind option to specify SESSION as the qualifier for all SQL statements in the plan or package.

If you use SESSION as the qualifier for a table name but the application process does not include a DECLARE GLOBAL TEMPORARY TABLE statement for the table name, DB2 assumes that you are not referring to a declared temporary table. DB2 resolves such table references to a table whose definition is persistent and appears in the DB2 catalog tables.

With the exception of the DECLARE GLOBAL TEMPORARY TABLE statement, any static SQL statement that references a declared temporary table is incrementally bound at run time. This is because the definition of the declared temporary table does not exist until the DECLARE GLOBAL TEMPORARY statement is executed in the application process that contains those SQL statements and the definition does not persist when the application process finishes running.

When a plan or package is bound, any static SQL statement (other than the DECLARE GLOBAL TEMPORARY TABLE statement) that references a *table-name* that is qualified by SESSION, regardless of whether the reference is for a declared temporary table, is not completely bound. However, the bind of the plan or package succeeds if there are no other errors. These static SQL statements are then incrementally bound at run time when the static SQL statement is issued. Object dependencies are not recorded in SYSIBM.SYSPLANDEP or SYSIBM.SYSPACKDEP tables. These incremental binds are necessary because:

- The definition of the declared temporary table does not exist until the DECLARE GLOBAL TEMPORARY TABLE statement for the table is executed in the same application process that contains those SQL statements. Therefore, DB2 must wait until the plan or package is run to determine if SESSION.*table-name* refers to a base table or a declared temporary table.
- The definition of a declared temporary table does not persist after the table it is explicitly dropped (DROP statement), implicitly dropped (ON COMMIT DROP TABLE), or the application process that defined it finishes running. When the application process terminates or is re-used as a reusable application thread, the instantiated rows of the table are deleted and the definition of the declared temporary table is dropped if it has not already been explicitly or implicitly dropped.

After the plan or package is bound, any static SQL statement that refers to a *table-name* that is qualified by SESSION has a new statement status of M in the DB2 catalog table (STATUS column of SYSIBM.SYSSTMT or SYSIBM.SYSPACKSTMT).

Thread reuse: If a declared temporary table is defined in an application process that is running as a local thread, the application process or local thread that declared the table qualifies for explicit thread reuse if:

- The table was defined with the ON COMMIT DELETE ROWS attribute, which is the default.
- The table was defined with the ON COMMIT PRESERVE ROWS attribute and the table was explicitly dropped with the DROP TABLE statement before the thread's commit operation.
- The table was defined with the ON COMMIT DROP TABLE attribute. When a declared temporary table is defined with the ON COMMIT DROP TABLE and a commit occurs, the table is implicitly dropped if there are no open cursors defined with the WITH HOLD option.

When the thread is reused, the declared temporary table is dropped and its rows are destroyed. However, if you do not explicitly or implicitly drop all declared temporary tables before or when your thread performs a commit and the thread becomes idle waiting to be reused, as with all thread reuse situations, the idle thread holds resources and locks. This includes some declared temporary table resources and locks on the table spaces and the database descriptor (DBD) for the work file database. So, instead of using the implicit drop feature of thread reuse to drop your declared temporary tables, it is recommended that you:

- Use the DROP TABLE statement to explicitly drop your declared temporary tables before the thread performs a commit and becomes idle.
- Define the declared temporary tables with ON COMMIT DROP TABLE clause so that the tables are implicitly dropped when a commit occurs.

Explicitly dropping the tables before a commit occurs or having them implicitly dropped when the commit occurs enables you to maximize the use of declared temporary table resources and release locks when multiple threads are using declared temporary table.

Remote threads qualify for thread reuse differently than local threads. If a declared temporary table is defined (with or without ON COMMIT DELETE ROWS) in an application process that is running as a remote or DDF thread (also known as Database Access Thread or DBAT), the remote thread qualifies for thread reuse only when the declared temporary table is explicitly dropped before the thread performs a commit operation. Dropping the declared temporary table enables the remote thread to qualify for the implicit thread reuse that is supported for DDF threads via connection pooling and to become an inactive DBAT (type 1 inactive thread) or an inactive connection (type 2 inactive thread).

Parallelism support: Only I/O and CP parallelism are supported. Any query that involves a declared temporary table is limited to parallel tasks on a single CPC.

Restrictions on the use of declared temporary tables: Declared temporary tables cannot:

- Be specified in referential constraints.
- Be referenced in any SQL statements that are defined in a trigger body (CREATE TRIGGER statement). If you refer a table name that is qualified with SESSION in a trigger body, DB2 assumes that you are referring to a base table.
- Be referenced in a CREATE INDEX statement unless the schema name of the index is SESSION.

In addition, do not refer to a declared temporary table in any of the following statements.

ALTER INDEX	CREATE VIEW
ALTER TABLE	GRANT (table or view privileges)
COMMENT	LABEL
CREATE ALIAS	LOCK TABLE
CREATE FUNCTION (TABLE LIKE clause)	REFRESH TABLE
CREATE PROCEDURE (TABLE LIKE clause)	RENAME
CREATE TRIGGER	REVOKE (table or view privileges)

Declared global temporary tables and dynamic statement caching: The DB2 dynamic statement cache feature does not support dynamic SQL statements that reference declared temporary tables, even if the SQL statement also includes references to base or persistent tables. DB2 will not insert such statements into the dynamic statement cache. Instead, these dynamic statements are processed as if statement caching is not in effect. Declared temporary tables are unique and specific to an application process or DB2 thread, cannot be shared across threads, are not described in the DB2 catalog, and do not persist beyond termination of the DB2 thread or application process. These attributes prevent the use of the dynamic statement cache feature where tables and SQL statements are shared across threads or application processes.

Table space requirements in the work file database: DB2 stores all declared temporary tables in the work file database. You cannot define a declared temporary table unless a table space with at least an 32KB page size exists in the work file database.

Alternative syntax and synonyms: To provide compatibility with previous releases, DB2 allows you to specify:

- LONG VARCHAR as a synonym for VARCHAR(*integer*) and LONG VARGRAPHIC as a synonym for VARGRAPHIC(*integer*) when defining the data type of a column.

However, the use of these synonyms is not encouraged because after the statement is processed, DB2 considers a LONG VARCHAR column to be VARCHAR and a LONG VARGRAPHIC column to be VARGRAPHIC.

- DEFINITION ONLY as a synonym for WITH NO DATA.

Examples

Example 1: Define a declared temporary table with column definitions for an employee number, salary, commission, and bonus.

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
  (EMPNO    CHAR(6)    NOT NULL,
   SALARY   DECIMAL(9, 2),
   BONUS    DECIMAL(9, 2),
   COMM     DECIMAL(9, 2))
  CCSID EBCDIC
  ON COMMIT PRESERVE ROWS;
```

Example 2: Assume that base table USER1.EMPTAB exists and that it contains three columns, one of which is an identity column. Declare a temporary table that has the same column names and attributes (including identity attributes) as the base table.


```
DECLARE GLOBAL TEMPORARY TABLE TEMPTAB1  
  LIKE USER1.EMPTAB  
  INCLUDING IDENTITY  
  ON COMMIT PRESERVE ROWS;
```

In the above example, DB2 uses SESSION as the implicit qualifier for TEMPTAB1.

DECLARE STATEMENT

The DECLARE STATEMENT statement is used for application program documentation. It declares names that are used to identify prepared SQL statements.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax

```

  >> DECLARE statement-name , statement-name , ... STATEMENT <<

```

Description

statement-name **STATEMENT**

Lists one or more names that are used in your application program to identify prepared SQL statements.

Example

This example shows the use of the DECLARE STATEMENT statement in a PL/I program.

```

EXEC SQL DECLARE OBJECT_STATEMENT STATEMENT;

EXEC SQL INCLUDE SQLDA;
EXEC SQL DECLARE C1 CURSOR FOR OBJECT_STATEMENT;

( SOURCE_STATEMENT IS "SELECT DEPTNO, DEPTNAME,
  MGRNO FROM DSN8910.DEPT WHERE ADMRDEPT = 'A00'" )

EXEC SQL PREPARE OBJECT_STATEMENT FROM SOURCE_STATEMENT;
EXEC SQL DESCRIBE OBJECT_STATEMENT INTO SQLDA;

/* Examine SQLDA */

EXEC SQL OPEN C1;

DO WHILE (SQLCODE = 0);
  EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA;

/* Print results */

END;

EXEC SQL CLOSE C1;
```

DECLARE TABLE

The DECLARE TABLE statement is used for application program documentation. It also provides the precompiler with information used to check your embedded SQL statements. (The DCLGEN subcommand can be used to generate declarations for tables and views described in any accessible DB2 catalog.

For more on DCLGEN, see *DB2 Application Programming and SQL Guide* and *DB2 Command Reference*.)

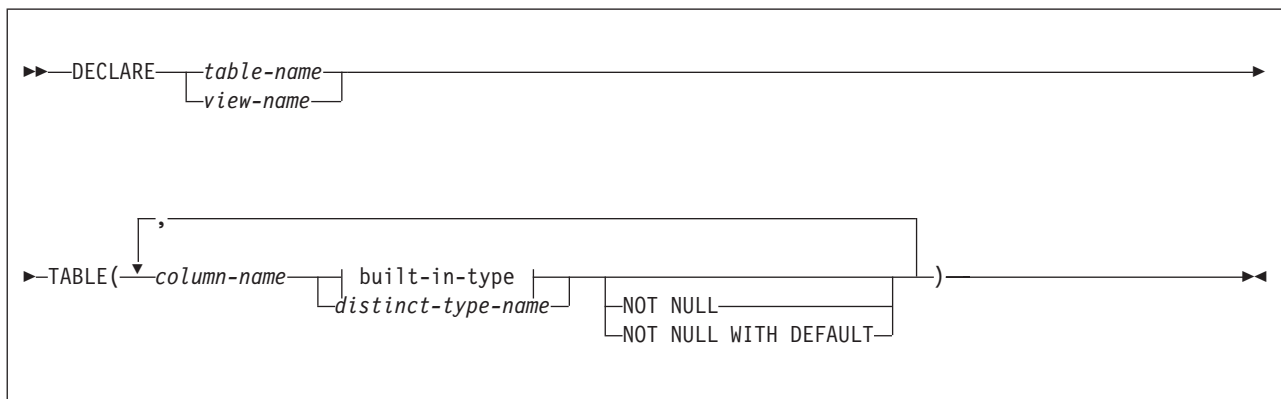
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

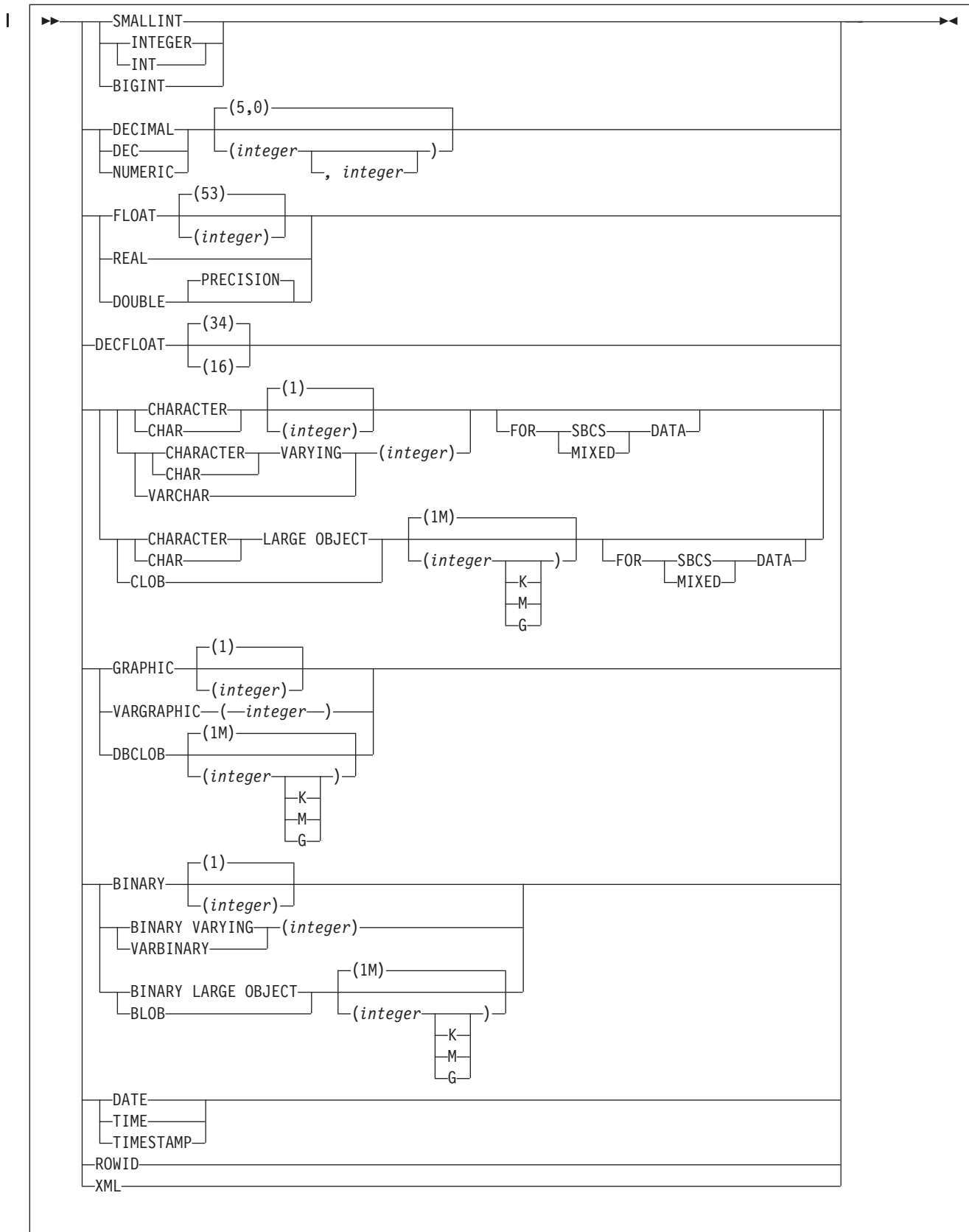
Authorization

None required.

Syntax



built-in-type:



Description

table-name or view-name

Specifies the name of the table or view to document. If the table is defined in your application program, the description of the table in the SQL statement in which it is defined (for example, CREATE TABLE or DECLARE GLOBAL TEMPORARY TABLE statement) and the DECLARE TABLE statement must be identical.

column-name

Specifies the name of a column of the table or view.

The precompiler uses these names to check for consistency of names within your SQL statements. It also uses the data type to check for consistency of names and data types within your SQL statements.

built-in-type

Specifies the built-in data type of the column. Use one of the built-in data types. For more information on the data types, see built-in-type.

distinct-type-name

Specifies the distinct type (user-defined data type) of the column. An implicit or explicit schema name qualifies the name.

NOT NULL

Specifies that the column does not allow null values and does not provide a default value.

NOT NULL WITH DEFAULT

Specifies that the column does not allow null values but provides a default value.

Notes

Error handling during processing: If an error occurs during the processing of the DECLARE TABLE statement, a warning message is issued, and the precompiler continues processing your source program.

Documenting a distinct type column: Although you can specify the name of a distinct type as the data type of a column in the DECLARE TABLE statement, use the built-in data type on which the distinct type is based instead. Using the base type enables the precompiler to check the embedded SQL statements for errors; otherwise, error checking is deferred until bind time.

To determine the source data type of the distinct type, check the value of column SOURCETYPE in catalog table SYSDATATYPES.

Examples

Example 1: Declare the sample employee table, DSN8910.EMP.

```
EXEC SQL DECLARE DSN8910.EMP TABLE
  (EMPNO      CHAR(6)      NOT NULL,
   FIRSTNAME  VARCHAR(12)  NOT NULL,
   MIDINIT    CHAR(1)      NOT NULL,
   LASTNAME   VARCHAR(15)  NOT NULL,
   WORKDEPT   CHAR(3)      ,
   PHONENO    CHAR(4)      ,
   HIREDATE   DATE          ,
   JOB        CHAR(8)      ,
   EDLEVEL    SMALLINT     ,
   SEX        CHAR(1)      ,
```

```

BIRTHDATE DATE
SALARY DECIMAL(9,2)
BONUS DECIMAL(9,2)
COMM DECIMAL(9,2)
);

```

Example 2: Assume that table CANADIAN_SALES keeps information for your company's sales in Canada. The table was created with the following definition:

```

CREATE TABLE CANADIAN_SALES
(PRODUCT_ITEM INTEGER,
MONTH INTEGER,
YEAR INTEGER,
TOTAL CANADIAN_DOLLAR);

```

CANADIAN_DOLLAR is a distinct type that was created with the following statement:

```

CREATE TYPE CANADIAN_DOLLAR AS DECIMAL(9,2);

```

Declare the CANADIAN_SALES table, using the source type for CANADIAN_DOLLAR instead of the distinct type name.

```

DECLARE TABLE CANADIAN_SALES
(PRODUCT_ITEM INTEGER,
MONTH INTEGER,
YEAR INTEGER,
TOTAL DECIMAL(9,2);

```

DECLARE VARIABLE

The DECLARE VARIABLE statement defines a CCSID for a host variable and the subtype of the variable. When it appears in an application program, the DECLARE VARIABLE statement causes the DB2 precompiler to tag a host variable with a specific CCSID. When the host variable appears in an SQL statement, the DB2 precompiler places this CCSID into the structures that it generates for the SQL statement.

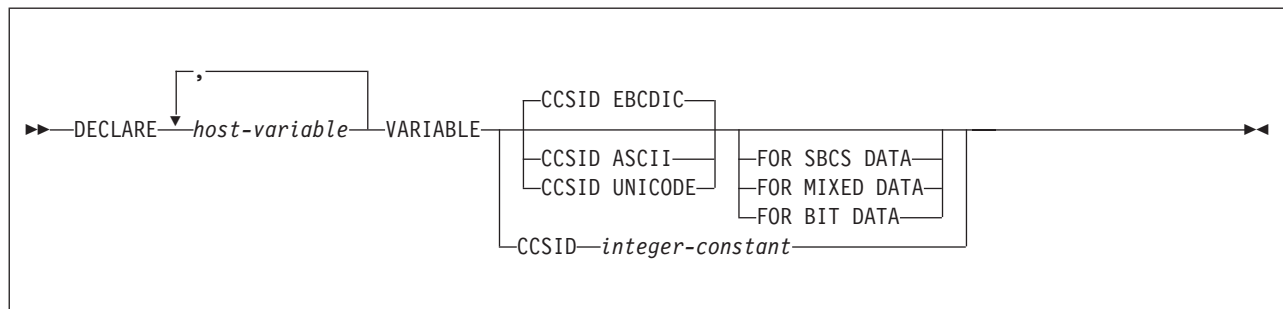
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax



Description

host-variable

Identifies a character or graphic string host variable defined in the program. An indicator variable cannot be specified for the host-variable.

CCSID ASCII, EBCDIC, or UNICODE

Specifies that the appropriate default CCSID for the specified encoding scheme of the server should be used. If this clause is not specified, the CCSID of the variable is the appropriate default EBCDIC CCSID of the server.

CCSID ASCII

Specifies that the default ASCII CCSID for the type of the variable at the server should be used.

CCSID EBCDIC

Specifies that the default EBCDIC CCSID for the type of the variable at the server should be used. CCSID EBCDIC is the default if this option is not specified.

CCSID UNICODE

Specifies that the default UNICODE CCSID for the type of the variable at the server should be used.

FOR SBCS DATA, FOR MIXED DATA, or FOR BIT DATA

Specifies the type of data contained in the variable *host-variable*. The FOR clause cannot be specified when declaring a graphic host variable.

For ASCII or EBCDIC data, if this clause is not specified when declaring a character host variable, the default is FOR SBCS DATA if MIXED DATA = NO on the install panel DSNTIPF. The default is FOR MIXED DATA if MIXED DATA = YES on the install panel DSNTIPF.

For UNICODE data, the default is always FOR MIXED DATA, regardless of the setting of MIXED DATA on the install panel DSNTIPF.

FOR SBCS DATA

Specifies that the values of the host variable can contain only SBCS (single-byte character set) data.

FOR MIXED DATA

Specifies that the values of the host variable can contain both SBCS data and DBCS data.

FOR BIT DATA

Specifies that the values of the host-variable are not associated with a coded character set and, therefore, are never converted. The CCSID of a FOR BIT DATA host variable is 65535.

CCSID *integer-constant*

Specifies that the values of the host variable contain data that is encoded using CCSID *integer-constant*. If the integer is an SBCS CCSID, the host variable is SBCS data. If the integer is a mixed data CCSID, the host variable is mixed data. For character host variables, the CCSID specified must be an SBCS, mixed CCSID, or UNICODE (UTF-8) CCSID. For graphic host variables, the CCSID specified must be a DBCS or UNICODE (UTF-16) CCSID. The valid range of values for the integer is 1 - 65533.

Notes

Placement of statement: The DECLARE VARIABLE statement can be specified anywhere in an application program that SQL statements are valid with the following exception. The DECLARE VARIABLE statement must occur before an SQL statement that refers to a host variable specified in the DECLARE VARIABLE statement.

CCSID exceptions for EXECUTE IMMEDIATE or PREPARE: When the host variable appears in an SQL statement, the DB2 precompiler places the appropriate numeric CCSID into the structures it generates for the SQL statement. This placement of the CCSID occurs for any SQL statement other than the EXECUTE IMMEDIATE or PREPARE statements. The placement of the CCSID also occurs for a *host-variable* in an EXECUTE IMMEDIATE or PREPARE statement, but it does not occur for a variable in a *string-expression* in an EXECUTE IMMEDIATE or PREPARE statement.

If a PL/1 application program contains at least one DECLARE VARIABLE statement, a *string-expression* in any EXECUTE IMMEDIATE or PREPARE statement cannot be preceded by a colon. An expression that consists of just a variable name preceded by a colon is interpreted as a *host-variable*.

Specific host languages: If a DECLARE VARIABLE statement is used in an assembler source program, the ONEPASS precompiler option must not be used. If a DECLARE VARIABLE statement is used in a C, C++, or PL/I source program, the TWOPASS precompiler option must be used. For those languages, or COBOL, the host-variable definition can either precede or follow a DECLARE VARIABLE statement that refers to that variable. If a DECLARE VARIABLE statement is used

in a FORTRAN source program, then the host-variable definition must precede the DECLARE VARIABLE statement.

Example

Example: Define the following host variables using PL/I data types: FRED as fixed length bit data, JEAN as fixed length UTF-8 (mixed) data, DAVE as varying length UTF-8 (mixed) data, PETE as fixed length graphic UTF-16 data, and AMBER as varying length graphic UTF-16 data.

Use the DECLARE VARIABLE statement to specify a data subtype or CCSID for these host variables: FRED as CCSID EBCDIC, JEAN as CCSID 1208 or CCSID UNICODE, DAVE as CCSID 1208 or CCSID UNICODE, PETE as CCSID 1200 or CCSID UNICODE, and AMBER as CCSID 1200 or CCSID UNICODE.

```
EXEC SQL BEGIN DECLARE SECTION;
  DCL FRED CHAR(10);
    EXEC SQL DECLARE :FRED VARIABLE CCSID EBCDIC FOR BIT DATA;
  DCL JEAN CHAR(30);
    EXEC SQL DECLARE :JEAN VARIABLE CCSID 1208;
  DCL DAVE CHAR(9) VARYING;
    EXEC SQL DECLARE :DAVE VARIABLE CCSID UNICODE;
  DCL PETE GRAPHIC(10);
    EXEC SQL DECLARE :PETE VARIABLE CCSID 1200;
  DCL AMBER VARGRAPHIC(20);
    EXEC SQL DECLARE :AMBER VARIABLE CCSID UNICODE;
EXEC SQL END DECLARE SECTION;
```

DELETE

The DELETE statement deletes rows from a table or view or activates an instead of delete trigger. The table or view can be at the current server or any DB2 subsystem with which the current server can establish a connection. Deleting a row from a view deletes the row from the table on which the view is based if no instead of trigger is defined for the delete operation on this view. If such a trigger is defined, the trigger is activated instead of the delete operation.

There are two forms of this statement:

- The *searched* DELETE form is used to delete one or more rows, optionally determined by a search condition.
- The *positioned* DELETE form specifies that one or more rows corresponding to the current cursor position are to be deleted.

Invocation

This statement can be embedded in an application program or issued interactively. A positioned DELETE is embedded in an application program. Both the embedded and interactive forms are executable statements that can be dynamically prepared.

Authorization

Authority requirements depend on whether the object identified in the statement is a user-defined table, a catalog table, or a view, and whether the statement is a searched DELETE and SQL standard rules are in effect:

When a table other than a catalog table is identified: The privilege set must include at least one of the following:

- The DELETE privilege on the table
- Ownership of the table
- DBADM authority on the database that contains the table
- SYSADM authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

When a catalog table is identified: The privilege set must include at least one of the following:

- DBADM authority on the catalog database
- SYSCTRL authority
- SYSADM authority

When a view is identified: The privilege set must include at least one of the following:

- The DELETE privilege on the view
- SYSADM authority

In a searched delete, the SELECT privilege is required in addition to the DELETE privilege when the option for the SQL standard is set as follows:

Searched DELETE and SQL standard rules: If SQL standard rules are in effect and the search-condition in a searched DELETE contains a reference to a column of the table or view, the privilege set must include at least one of the following:

- The SELECT privilege on the table or view
- Ownership of the table or view
- DBADM authority on the database that contains the table, if the target is a table and that table that is not a catalog table
- SYSADM authority

SQL standard rules are in effect as follows:

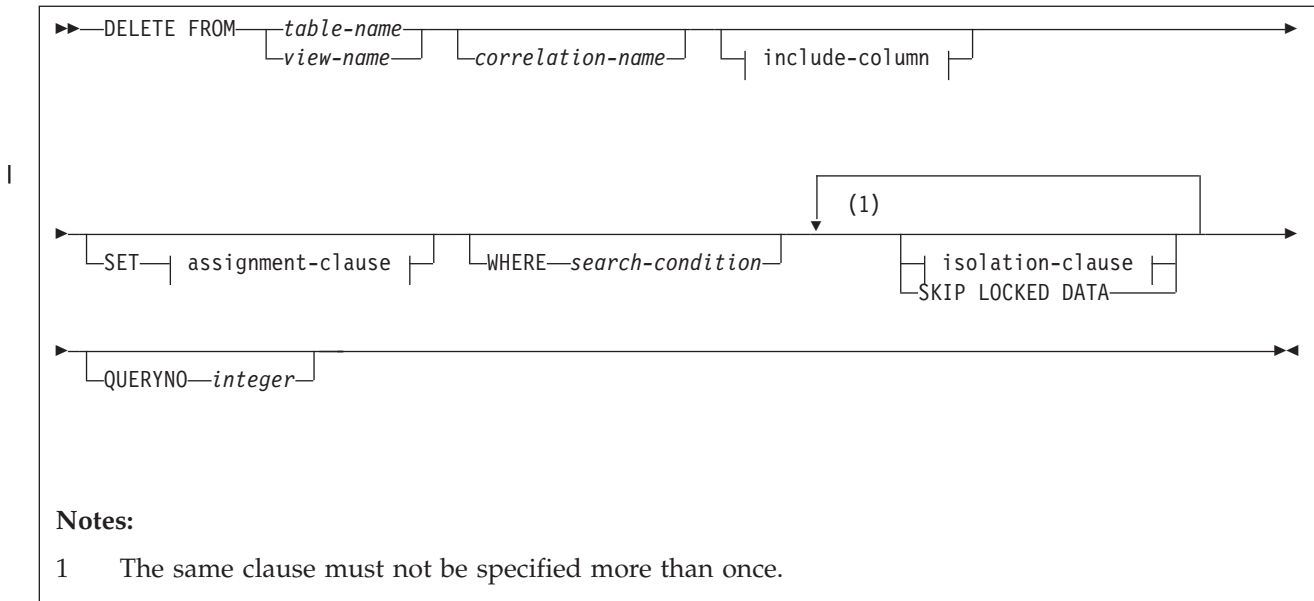
- For static SQL statements, if the SQLRULES(STD) bind option was specified
- For dynamic SQL statements, if the CURRENT RULES special register is set to 'STD'

The owner of a view, unlike the owner of a table, might not have DELETE authority on the view (or might have DELETE authority without being able to grant it to others). The nature of the view itself can preclude its use for DELETE. For more information, see the description of authority in “CREATE VIEW” on page 1184.

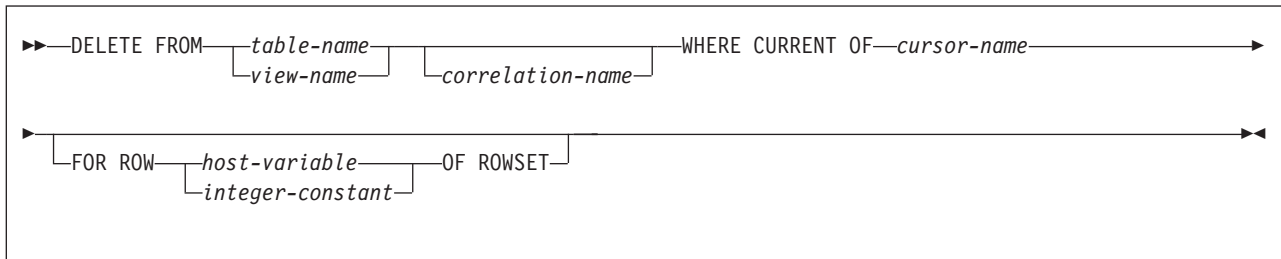
If a subselect is specified, the privilege set must include authority to execute the subselect. For more information about the subselect authorization rules, see “Authorization” on page 630.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)

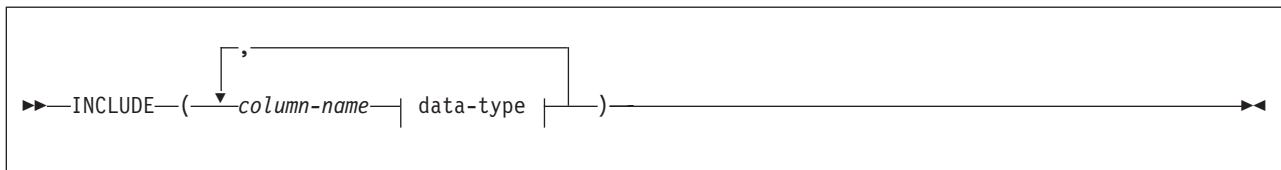
searched delete:



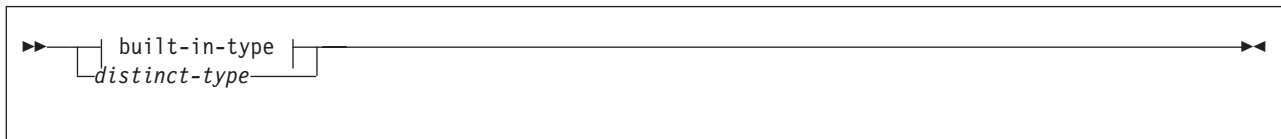
positioned delete:



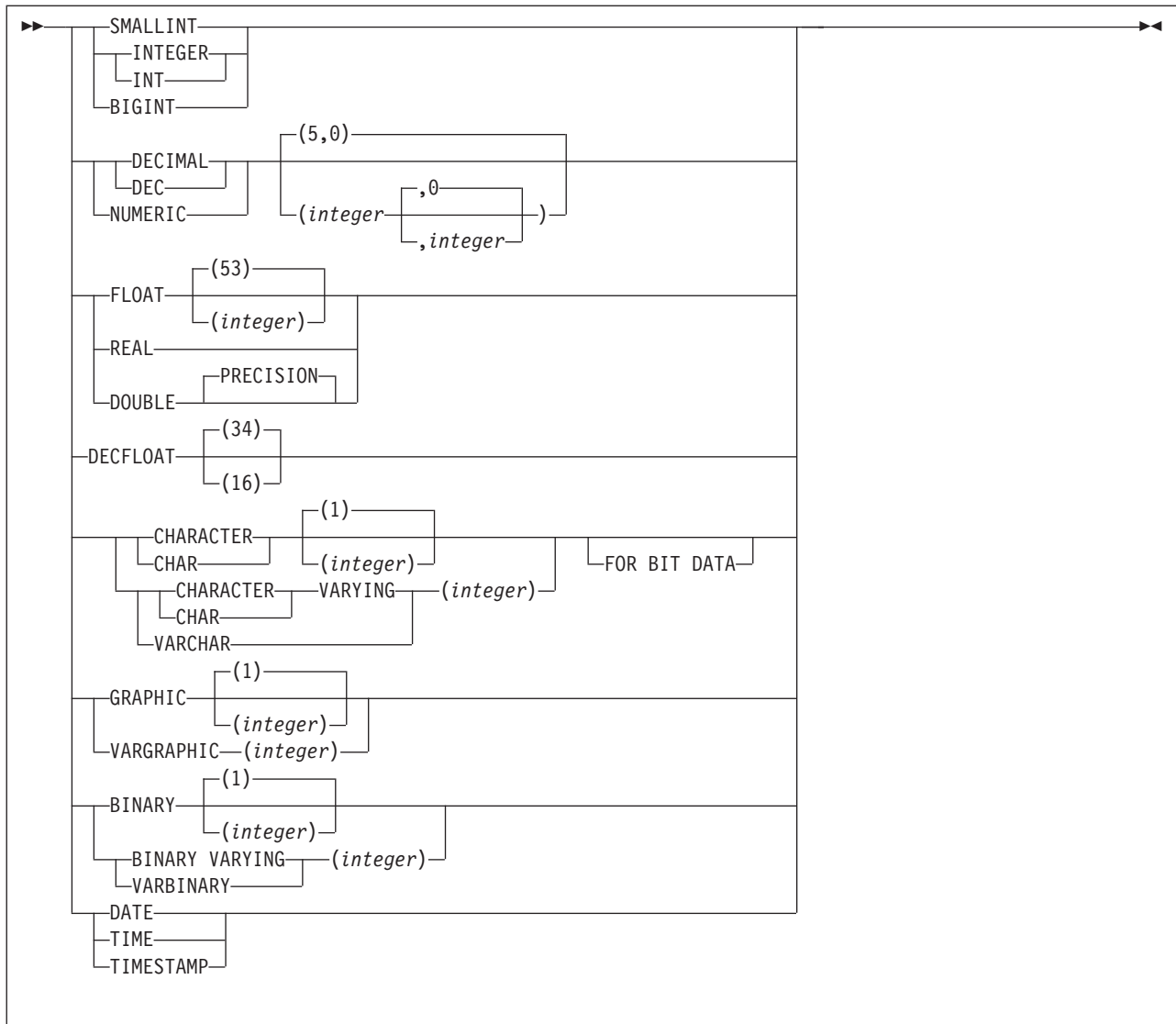
include-column:



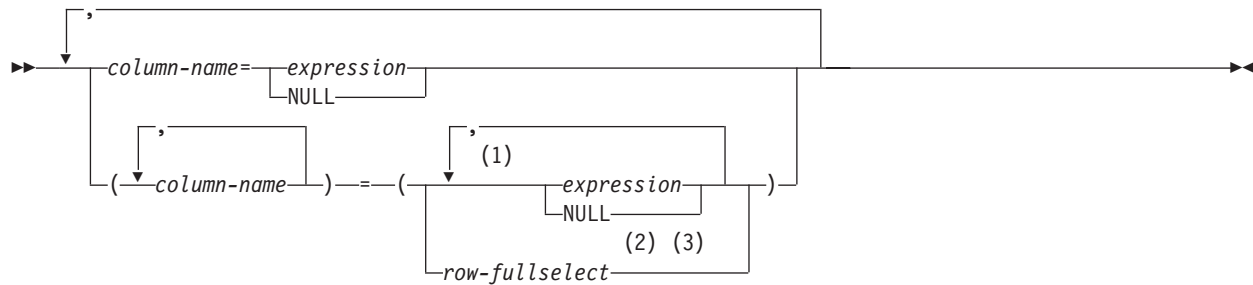
data-type:



built-in-type:



assignment clause:



Notes:

- 1 The number of *expressions* and NULL keywords must match the number of *column-names*.
- 2 The number of columns in the select list must match the number of *column-names*.
- 3

isolation-clause:

Description

FROM *table-name* **or** *view-name*

Identifies the table or view from which rows are to be deleted. The name must identify a table or view that exists at the DB2 subsystem identified by the implicitly or explicitly specified location name. The name must not identify:

- An auxiliary table
- A catalog table for which deletes are not allowed
- A view of such a catalog table
- A read-only view (For a description of a read-only view, see “CREATE VIEW” on page 1184.)
- A system-maintained materialized query table
- A table that is implicitly created for an XML column

In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must be a remote server that supports two-phase commit.

correlation-name

Specifies an alternate name that can be used within the *search-condition* to designate the table or view. (For an explanation of correlation names, see “Correlation names” on page 154.)

include-column

Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the result table of the DELETE statement when it is nested in the FROM clause of the outer fullselect that is used in a subselect, SELECT statement, or in a SELECT INTO statement. The included columns are appended to the end of the list of columns that is identified by *table-name* or *view-name*. If no value is assigned to a column that is specified by an *include-column*, a NULL value is returned for that column.

INCLUDE

Introduces a list of columns that are to be included in the result table of the DELETE statement. The included columns are only available if the DELETE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

column-name

Specifies the name for a column of the result table of the DELETE statement that is not the same name as another included column nor a column in the table or view that is specified in *table-name* or *view-name*.

data-type

Specifies the data type of the included column. The included columns are nullable.

built-in-type

Specifies a built-in data type. See “CREATE TABLE” on page 1079 for a description of each built-in type.

distinct-type

Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

SET

Introduces the assignment of values to columns.

assignment-clause

The assignment-clause introduces a list of one or more *column-names* and the values that are to be assigned to the columns. The *column-names* are the only columns that can be set using the *assignment-clause*.

column-name

Identifies an INCLUDE column.

Assignments to included columns are only processed when the DELETE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. The columns that are named in the INCLUDE clause are the only columns that can be set using the SET clause. The null value is returned for an included column that is not set by using an explicit SET clause.

expression

Indicates the new value of the column. The *expression* is any expression of the type described in “Expressions” on page 180. It must not include an aggregate function.

A *column-name* in an expression must identify a column of the table or view. For each row that is deleted, the value of the column in the expression is the value of the column in the row before the row is deleted.

NULL

Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

row-fullselect

Specifies a fullselect that returns a single row. The column values are assigned to each of the corresponding *column-names*. If the fullselect returns no rows, the null value is assigned to each column; an error occurs if any column that is to be deleted is not nullable. An error also occurs if there is more than one row in the result.

If the fullselect refers to columns that are to be deleted, the value of such a column in the fullselect is the value of the column in the row before the row is deleted.

WHERE

Specifies the rows to be deleted. You can omit the clause, give a search condition, or specify a cursor. For a created temporary table or a view of a created temporary table, you must omit the clause. When the clause is omitted, all the rows of the table or view are deleted.

search-condition

Is any search condition as described in Chapter 2, “Language elements,” on page 47. Each *column-name* in the search condition, other than in a subquery, must identify a column of the table or view.

The search condition is applied to each row of the table or view and the deleted rows are those for which the result of the search condition is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, a subquery with no correlated references is executed just once, whereas it is possible that a subquery with a correlated reference must be executed once for each row.

Let T2 denote the object table of a DELETE statement and let T1 denote a table that is referred to in the FROM clause of a subquery of that statement. T1 must not be a table that can be affected by the DELETE on T2. Thus, the following rules apply:

- T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL, unless the result of the subquery is materialized before the DELETE action is executed.
- T1 must not be a dependent of T3 in a relationship with a delete rule of CASCADE or SET NULL if deletes of T2 cascade to T3.

WHERE CURRENT OF *cursor-name*

Identifies the cursor to be used in the delete operation. *cursor-name* must identify a declared cursor as explained in the description of the DECLARE CURSOR statement in “DECLARE CURSOR” on page 1191. If the DELETE statement is embedded in a program, the DECLARE CURSOR statement must include *select-statement* rather than *statement-name*.

The table or view named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must be capable of being deleted. For an explanation of read-only result tables, see Read-only cursors. Note that the object of the DELETE statement must not be identified as the object of the subquery in the WHERE clause of the SELECT statement of the cursor.

If the cursor is ambiguous and the plan or package was bound with CURRENTDATA(NO), DB2 might return an error to the application if DELETE WHERE CURRENT OF is attempted for any of the following:

- A cursor that is using block fetching
- A cursor that is using query parallelism
- A cursor that is positioned on a row that has been modified by this or another application process

When the DELETE statement is executed, the cursor must be open and positioned on a row or rowset of the result table.

- If the cursor is positioned on a single row, that row is the one deleted, and after the deletion the cursor is positioned before the next row of its result table. If there is no next row, the cursor positioned after the last row.
- If the cursor is positioned on a rowset, all rows corresponding to the rows of the current rowset are deleted, and after the deletion the cursor is positioned before the next rowset of its result table. If there is no next rowset, the cursor positioned after the last rowset.

A positioned DELETE must not be specified for a cursor that references a view on which an instead of delete trigger is defined, even if the view is an updatable view.

FOR ROW n OF ROWSET

Specifies which row of the current rowset is to be deleted. The corresponding row of the rowset is deleted, and the cursor remains positioned on the current rowset. If the rowset consists of a single row, or all other rows in the rowset have already been deleted, then the cursor is positioned before the next rowset of the result table. If there is no next rowset, the cursor is positioned after the last rowset.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, must not include an indicator variable, and *k* must be in the range of 1 to 32767. The cursor must be positioned on a rowset, and the specified value must be a valid value for the set of rows most recently retrieved for the cursor.

If the specified row cannot be deleted, an error is returned. It is possible that the specified row is within the bounds of the rowset most recently requested, but the current rowset contains less than the number of rows that were implicitly or explicitly requested when that rowset was established.

If, via a positioned delete against a sensitive static cursor that specifies a particular row of the current rowset, and that row has been identified as a delete hole (that is, a row in the result table whose corresponding row has been deleted from the base table), an error is returned.

If, via a positioned delete against a sensitive static cursor that specifies a particular row of the current rowset, and that row has been identified as an update hole (that is, a row in the result table whose corresponding row has been updated so that it no longer satisfies a predicate of the SELECT statement), an error is returned.

It is possible for another application process to delete a row in the base table of the SELECT statement so that the specified row of the cursor no longer has a corresponding row in the base table. An attempt to delete such a row results in an error.

If the FOR ROW n OF ROWSET clause is not specified, the current position of cursor determines the rows that are affected by the statement:

- If the cursor is positioned on a single row, that row is the one deleted. After the row is deleted, the cursor is positioned before the next row of its result table. If there is no next row, the cursor positioned after the last row.
- If the cursor is positioned on a rowset, all rows corresponding to the rows of the current rowset are deleted. After the rows are deleted, the cursor is positioned before the next rowset of its result table. If there is no next rowset, the cursor positioned after the last rowset.

isolation-clause

Specifies the isolation level used when locating the rows to be deleted by the statement.

WITH

Introduces the isolation level, which may be one of the following:

- RR** Repeatable read
- RS** Read stability
- CS** Cursor stability

The default isolation level of the statement is the isolation level of the package or plan in which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

SKIP LOCKED DATA

The SKIP LOCKED DATA clause specifies that rows are skipped when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only when isolation CS or RS is in effect and applies only to row level or page level locks.

For DELETE statements, SKIP LOCKED DATA can be specified only in the searched form of the DELETE statement. SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is repeatable read (WITH RR) or uncommitted read (WITH UR). The default isolation level of the statement depends on the isolation level of the package or plan with which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

QUERYNO *integer*

Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:

- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

Notes

Delete operation errors: If an error occurs during the execution of any delete operation, no changes are made. If an error occurs during the execution of a positioned delete, the position of the cursor is unchanged. However, it is possible for an error to make the position of the cursor invalid, in which case the cursor is closed. It is also possible for a delete operation to cause a rollback, in which case the cursor is closed.

Position of cursor: If an application process deletes a row on which any of its cursors are positioned, those cursors are positioned before the next row of the

result table. Let C be a cursor that is positioned before row R (as a result of an OPEN, a DELETE through C, a DELETE through some other cursor, or a searched DELETE). In the presence of an SQL data change statements that affect the base table from which R is derived, the next FETCH operation referencing C does not necessarily position C on R. For example, the operation can position C on R', where R' is a new row that is now the next row of the result table.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful delete operation. Until the locks are released by a commit or rollback operation, the effect of the delete operation can only be perceived by the application process that performed the deletion and the locks can prevent other application processes from performing operations on the table. Locks are not acquired when rows are deleted from a declared temporary table unless all the rows are deleted (DELETE FROM T). When all the rows are deleted from a declared temporary table, a segmented table lock is acquired on the pages for the table and no other table in the table space is affected.

Triggers: Delete operations can cause triggers to be activated. A trigger might cause other statements to be executed or might raise error conditions that are based on the deleted rows. If a DELETE statement on a view causes an INSTEAD OF trigger to be activated, referential integrity is checked against the updates that are performed in the trigger and not against the underlying tables of the view that cause the trigger to be activated.

Referential integrity: If the identified table or the base table of the identified view is a parent, the rows selected must not have any dependents in a relationship with a delete rule of RESTRICT or NO ACTION. In addition, the delete operation must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT or NO ACTION.

If the delete operation is not prevented by a RESTRICT or NO ACTION delete rule, the selected rows are deleted and any rows that are dependents of the selected rows are also deleted.

- The nullable columns of foreign keys in any rows that are their dependents in a relationship governed by a delete rule of SET NULL are set to the null value.
- Any rows that are their dependents in a relationship governed by a delete rule of CASCADE are also deleted, and these rules apply, in turn, to those rows.

The only difference between NO ACTION and RESTRICT is when the referential constraint is enforced. RESTRICT (IBM SQL rules) enforces the rule immediately, and NO ACTION (SQL standard rules) enforces the rule at the end of the statement. This difference matters only in the case of a searched DELETE involving a self-referencing constraint that deletes more than one row. NO ACTION might allow the DELETE to be successful where RESTRICT (if it were allowed) would prevent it.

Check constraint: A check constraint can prevent the deletion of a row in a parent table when there are dependents in a relationship with a delete rule of SET NULL. If deleting a row in the parent table would cause a column in a dependent table to be set to null and there is a check constraint that specifies that the column must not be null, the row is not deleted.

Nesting user-defined functions or stored procedures: A DELETE statement can implicitly or explicitly refer to user-defined functions or stored procedures. This is

known as *nesting* of SQL statements. A user-defined function or stored procedure that is nested within the DELETE must not access the table from which you are deleting rows.

Indexes with VARBINARY columns: If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.

Number of rows deleted: Except as noted below, a delete operation sets SQLERRD(3) in the SQLCA to the number of deleted rows. This number does not include any rows that were deleted as a result of a CASCADE delete rule or a trigger.

DELETE FROM T without a WHERE clause deletes all rows of T. If a table T is contained in a segmented table space and is not a parent table, this deletion will be performed without accessing T. The SQLERRD(3) field is set to -1. (For a complete description of the SQLCA, including exceptions to the above, see “SQL communication area (SQLCA)” on page 1646.

Rules for positioned DELETE with SENSITIVE STATIC scrollable cursor: When a SENSITIVE STATIC scrollable cursor has been declared, the following rules apply:

- *Delete attempt of delete holes or update holes.* If, with a positioned delete against a SENSITIVE STATIC scrollable cursor, an attempt is made to delete a row that has been identified as a delete hole (that is, a row in the result table whose corresponding row has been deleted from the base table), an error occurs.
If an attempt is made to delete a row that has been identified as an update hole (that is, a row in the result table whose corresponding row has been updated so that it no longer satisfies the predicate of the SELECT statement), an error occurs.
- *Delete operations.* Positioned delete operations with SENSITIVE STATIC scrollable cursors perform as follows:
 1. The SELECT list items in the target row of the base table of the cursor are compared with the values in the corresponding row of the result table (that is, the result table must still agree with the base table). If the values are not identical, the delete operation is rejected and an error occurs. The operation can be attempted again after a successful FETCH SENSITIVE has occurred for the target row.
 2. The WHERE clause of the SELECT statement is re-evaluated to determine whether the current values in the base table still satisfy the search criteria. The values in the SELECT list are compared to determine that these values have not changed. If the WHERE clause evaluates as true, and the values in the SELECT list have not changed, the delete operation is allowed to proceed. Otherwise, an error occurs, the delete operation is rejected, and an update hole appears in the cursor.
 3. After the base table row is successfully deleted, the temporary result table is updated and the row is marked as a delete hole.
- *Rollback of delete holes.* Delete holes are usually permanent. Once a delete hole is identified, it remains a delete hole until the cursor is closed. However, if a positioned delete using *this* cursor actually caused the creation of the hole (that is, this cursor was used to make the changes that resulted in the hole) and the delete was subsequently rolled back, then the row is no longer considered a delete hole.

- *Result table.* Any deletes, either positioned or searched, to rows of the base table on which a SENSITIVE STATIC scrollable cursor is defined are reflected in the result table if a positioned update or positioned delete is attempted with the scrollable cursor. A SENSITIVE STATIC scrollable cursor sees these deletes when a FETCH SENSITIVE is attempted.

If the FOR ROW *n* OF ROWSET clause is not specified, the entire rowset fetched by the most recent FETCH statement that returned data for the specified cursor is deleted.

Deleting rows from a table with multilevel security: When you delete rows from a table with multilevel security, DB2 compares the security label of the user (the primary authorization ID) to the security label of the row. The delete proceeds according to the following rules:

- If the security label of the user and the security label of the row are equivalent, the row is deleted.
- If the security label of the user dominates the security label of the row, the user's write-down privilege determines the security the result of the DELETE statement:
 - If the user has write-down privilege or write-down control is not enabled, the row is deleted.
 - If the user does not have write-down privilege and write-down control is enabled, the row is not deleted.
- If the security label of the row dominates the security label of the user, the row is not deleted.

Other SQL statements in the same unit of work: The following statements cannot follow a DELETE statement in the same unit of work:

- An ALTER TABLE statement that changes the data type of a column (ALTER COLUMN SET DATA TYPE)
- An ALTER INDEX statement that changes the padding attribute of an index with varying-length columns (PADDED to NOT PADDED or vice versa)

Examples

Assume that the statements in the examples are embedded in PL/I programs.

Example 1: From the table DSN8910.EMP delete the row on which the cursor C1 is currently positioned.

```
EXEC SQL DELETE FROM DSN8910.EMP WHERE CURRENT OF C1;
```

Example 2: From the table DSN8910.EMP, delete all rows for departments E11 and D21.

```
EXEC SQL DELETE FROM DSN8910.EMP
WHERE WORKDEPT = 'E11' OR WORKDEPT = 'D21';
```

Example 3: From employee table X, delete the employee who has the most absences.

```
EXEC SQL DELETE FROM EMP X
WHERE ABSENT = (SELECT MAX(ABSENT) FROM EMP Y
WHERE X.WORKDEPT = Y.WORKDEPT);
```

Example 4: Assuming that cursor CS1 is positioned on a rowset consisting of 10 rows of table T1, delete all 10 rows in the rowset.

```
EXEC SQL DELETE FROM T1 WHERE CURRENT OF CS1;
```

Example 5: Assuming cursor CS1 is positioned on a rowset consisting of 10 rows of table T1, delete the fourth row of the rowset.

```
EXEC SQL DELETE FROM T1 WHERE CURRENT OF CS1 FOR ROW 4 OF ROWSET;
```

DESCRIBE

The DESCRIBE statement obtains information about an object. You can obtain the following types of information with this statement, each of which is described separately.

- Cursors

Gets information about the result set that is associated with the cursor. This information, such as column information, is put into a descriptor. See “DESCRIBE CURSOR” on page 1238.

- Input parameter markers of a prepared statement.

Gets information about the input parameter markers in a prepared statement. This information is put into a descriptor. See “DESCRIBE INPUT” on page 1240.

- The output of a prepared statement

Gets information about a prepared statement or information about the select list columns in a prepared SELECT statement. This information is put into a descriptor. See “DESCRIBE OUTPUT” on page 1243.

- Procedures

Gets information about the result sets returned by a stored procedure. The information, such as the number of result sets, is put into a descriptor. See “DESCRIBE PROCEDURE” on page 1250.

- Tables

Gets information about a table or view. This information is put into a descriptor. See “DESCRIBE TABLE” on page 1253.

DESCRIBE CURSOR

The DESCRIBE CURSOR statement gets information about the result set that is associated with the cursor. The information, such as column information, is put into a descriptor. Use DESCRIBE CURSOR for result set cursors from stored procedures. The cursor must be defined with the ALLOCATE CURSOR statement.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

►►—DESCRIBE CURSOR—*cursor-name*
host-variable—INTO—*descriptor-name*—►►

Description

cursor-name **or** *host-variable*

Identifies a cursor by the specified *cursor-name* or the cursor name contained in *host-variable*. The name must identify a cursor that has already been allocated in the source program.

If *host-variable* is used:

- It must be a character string variable that has a maximum length of 18 bytes.
- It must not be followed by an indicator variable.
- The cursor name must be left justified within the host variable and must not contain embedded blanks.
- If the length of the cursor name is less than the length of the host variable, it must be padded on the right with blanks.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA). The information returned in the SQLDA describes the columns in the result set associated with the named cursor.

The considerations for allocating and initializing the SQLDA are similar to those of a varying-list SELECT statement. For more information, see *DB2 Application Programming and SQL Guide*.

For REXX: The SQLDA is not allocated before it is used.

After the DESCRIBE CURSOR statement is executed, the contents of the SQLDA are the same as after a DESCRIBE for a SELECT statement, with the following exceptions:

- The first 5 bytes of the SQLDAID field are set to 'SQLRS'.
- Bytes 6 to 8 of the SQLDAID field are reserved. If the cursor is declared WITH HOLD in a stored procedure, the high-order bit of the 8th byte is set to 1.

These exceptions do not apply to a REXX SQLDA, which does not include the SQLDAID field.

Notes

Using cursors for result sets: Column names are included in the information that DESCRIBE CURSOR obtains when the statement that generates the result set is either:

- Dynamic
- Static and the value of field DESCRIBE FOR STATIC on installation panel DSNTIP4 was YES when the package or stored procedure was bound. If the value of the field was NO, the returned information includes only the data type and length of the columns.

Using host variables: If the DESCRIBE CURSOR statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Place information about the result set associated with cursor C1 into the descriptor named by :sqlda1.

```
EXEC SQL DESCRIBE CURSOR C1 INTO :sqlda1
```

Example 2: Place information about the result set associated with the cursor named by :hv1 into the descriptor named by :sqlda2.

```
EXEC SQL DESCRIBE CURSOR :hv1 INTO :sqlda2
```

DESCRIBE INPUT

The DESCRIBE INPUT statement obtains information about the input parameter markers of a prepared statement.

For an explanation of prepared statements, see “PREPARE” on page 1405 and “DESCRIBE PROCEDURE” on page 1250.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required if the statement is used for a prepared statement.

Syntax

►►—DESCRIBE INPUT—*statement-name*—INTO—*descriptor-name*—◄◄

Description

statement-name

Identifies the prepared statement. When the DESCRIBE INPUT statement is executed, the name must identify a statement that has been prepared by the application process at the current server.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA), which is described in “SQL descriptor area (SQLDA)” on page 1656. See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C. The information returned in the SQLDA describes the parameter markers.

Before the DESCRIBE INPUT statement is executed, the user must set the SQLN field in the SQLDA and the SQLDA must be allocated. Considerations for initializing and allocating the SQLDA are similar to those for the DESCRIBE statement (see “DESCRIBE” on page 1237). An occurrence of an extended SQLVAR is needed for each parameter in addition to the required base SQLVAR only if the input data contains LOBs.

For REXX: The SQLDA is not allocated before it is used.

After the DESCRIBE INPUT statement is executed, all the fields in the SQLDA except SQLN are either set by DB2 or ignored. The SQLDA contents are similar to the contents returned for the DESCRIBE statement (see page The SQLDA contents returned after DESCRIBE) with these exceptions:

- In the SQLDAID, DB2 sets the value of the seventh byte only to the space character or '2'. A value of '3' is never used. The value '2' indicates that two SQLVAR entries (an occurrence of both a base SQLVAR and an extended SQLVAR) are required for each parameter because the input data contains LOBs. The seventh byte is a space character when either of the following conditions is true:

- The input data does not contain LOBs. Only a base SQLVAR occurrence is needed for each parameter.
- Only a base SQLVAR occurrence is needed for each column of the result, and the SQLDA is not large enough to contain the returned information.
- The SQLD field is set to the number of parameter markers being described. The value is 0 if the statement being described does not have input parameter markers.
- The SQLNAME field is not used.
- The SQLDATATYPE is set to a nullable, regardless of the usage of the parameter markers in the prepared statement.
- The SQLDATATYPE-NAME is not used if an extended SQLVAR entry is present. DESCRIBE INPUT does not return information about distinct types.

For complete information on the contents of the fields, see “SQL descriptor area (SQLDA)” on page 1656.

Notes

Preparing the SQLDA for OPEN or EXECUTE: This note is relevant if you are applying DESCRIBE INPUT to a prepared statement and you intend to use the SQLDA in an OPEN or EXECUTE statement. To prepare the SQLDA for that purpose:

- Set SQLDATA to a valid address.
- If SQLTYPE is odd, set SQLIND to a valid address.

For the meaning of those fields in that context, see “SQL descriptor area (SQLDA)” on page 1656.

Support for extended dynamic SQL in a distributed environment: Unlike the DESCRIBE statement, which can be used in a distributed environment to describe static SQL statements generated by extended dynamic SQL, you cannot describe host variables in static SQL statements that are generated by extended dynamic SQL. A DESCRIBE INPUT statement issued against such static SQL statements always fails.

For information on how the DESCRIBE statement supports extended dynamic SQL, see Support for extended dynamic SQL in a distributed environment.

Using host variables: If the DESCRIBE INPUT statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Example

Execute a DESCRIBE INPUT statement with an SQLDA that has enough SQLVAR occurrences to describe any number of input parameters a prepared statement might have. Assume that five parameter markers at most will need to be described and that the input data does not contain LOBs.

```

/* STMT1_STR contains INSERT statement with VALUES clause */
EXEC SQL PREPARE STMT1_NAME FROM :STMT1_STR;
... /* code to set SQLN to 5 and to allocate the SQLDA */
EXEC SQL DESCRIBE INPUT STMT1_NAME INTO :SQLDA;
.
.
.
```

This example uses the first technique described in *Allocating the SQLDA to allocate the SQLDA*.

DESCRIBE OUTPUT

The DESCRIBE OUTPUT statement obtains information about a prepared statement.

For an explanation of prepared statements, see “PREPARE” on page 1405 and “DESCRIBE PROCEDURE” on page 1250.

Invocation

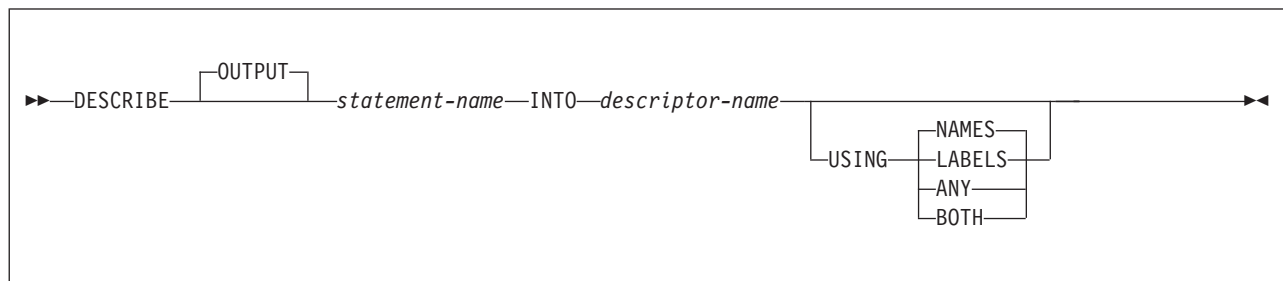
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required.

See “PREPARE” on page 1405 for the authorization required to create a prepared statement.

Syntax



Description

OUTPUT

When a *statement-name* is specified, optional keyword to indicate that the describe will return information about the select list columns in a the prepared SELECT statement.

statement-name

Identifies the prepared statement. When the DESCRIBE statement is executed, the name must identify a statement that has been prepared by the application process at the current server.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA), which is described in “SQL descriptor area (SQLDA)” on page 1656. See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C.

For languages other than REXX: Before the DESCRIBE statement is executed, the user must set the following variable in the SQLDA and the SQLDA must be allocated.

SQLN Indicates the number of SQLVAR occurrences provided in the SQLDA. DB2 does not change this value. For techniques to determine the number of required occurrences, see Allocating the SQLDA.

For REXX: The SQLDA is not allocated before it is used. An SQLDA consists of a set of stem variables. There is one occurrence of variable *stem.SQLD*, followed by zero or more occurrences of a set of variables that is equivalent to an SQLVAR structure. Those variables begin with *stem.n*.

After the DESCRIBE statement is executed, all the fields in the SQLDA except SQLN are either set by DB2 or ignored. For information on the contents of the fields, see The SQLDA contents returned after DESCRIBE.

USING

Indicates what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to a length of 0.

NAMES

Assigns the name of the column. This is the default.

LABELS

Assigns the label of the column. (Column labels are defined by the LABEL statement.)

ANY

Assigns the column label, and if the column has no label, the column name.

BOTH

Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $2 \times n$ or $3 \times n$, where n is the number of columns in the object being described. For each of the columns, the first n occurrences of SQLVAR, which are the base SQLVAR entries, contain the column names. Either the second or third n occurrences of SQLVAR, which are the extended SQLVAR entries, contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

Notes

Using PREPARE INTO clause: Information about a prepared statement can also be obtained by using the INTO clause of the PREPARE statement.

Allocating the SQLDA: Before the DESCRIBE or PREPARE INTO statement is executed, the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA. Also, enough storage must be allocated to contain the number of occurrences that SQLN specifies. To obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR must be at least equal to the number of columns. Furthermore, if USING BOTH is specified, or if the columns include LOBs or distinct types, the number of occurrences of SQLVAR should be two or three times the number of columns. See "Determining how many SQLVAR occurrences are needed" on page 1661 for more information.

First technique: Allocate an SQLDA with enough occurrences of SQLVAR to accommodate any select list that the application will have to process. At the extreme, the number of SQLVARs could equal three times the maximum number of columns allowed in a result table. After the SQLDA is allocated, the application can use the SQLDA repeatedly.

This technique uses a large amount of storage that is never deallocated, even when most of this storage is not used for a particular select list.

Second technique: Repeat the following two steps for every processed select list:

1. Execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR; that is, an SQLDA for which SQLN is zero.
2. Allocate a new SQLDA with enough occurrences of SQLVAR. Use the values that are returned in SQLD and SQLCODE to determine the number of SQLVAR entries that are needed. The value of SQLD is the number of columns in the result table, which is either the required number of occurrences of SQLVAR or a fraction of the required number (see “Determining how many SQLVAR occurrences are needed” on page 1661 for details). If the SQLCODE is +236, +237, +238, or +239, the number of SQLVAR entries that is needed is two or three times the value in SQLD, depending on whether USING BOTH was specified. Set SQLN to reflect the number of SQLVAR entries that have been allocated.
3. Execute the DESCRIBE statement again, using the new SQLDA.

This technique allows better storage management than the first technique, but it doubles the number of DESCRIBE statements.

Third technique: Allocate an SQLDA that is large enough to handle most (hopefully, all) select lists but is also reasonably small. If an execution of DESCRIBE fails because SQLDA is too small, allocate a larger SQLDA and execute the DESCRIBE statement again.

For the new larger SQLDA, use the values that are returned in SQLD and SQLCODE from the failing DESCRIBE statement to calculate the number of occurrences of SQLVAR that are needed, as described in technique two. Remember to check for SQLCODEs +236, +237, +238, and +239, which indicate whether *extended* SQLVAR entries are needed because the data includes LOBs or distinct types.

This third technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

The SQLDA contents returned on DESCRIBE: After a DESCRIBE statement is executed, the following list describes the contents of the SQLDA fields as they are set by DB2 or ignored. These descriptions do not necessarily apply to the uses of an SQLDA in other SQL statements (EXECUTE, OPEN, FETCH). For more on the other uses, see “SQL descriptor area (SQLDA)” on page 1656.

SQLDAID

DB2 sets the first 6 bytes to 'SQLDA ' (5 letters followed by the space character) and the eighth byte to a space character. The seventh byte is set to indicate the number of SQLVAR entries that are needed to describe each column of the result table as follows:

space The value of space occurs when:

- USING BOTH was not specified and the columns being described do not include LOBs or distinct types. Each column only needs one SQLVAR entry. If the SQL standard option is yes, DB2 sets SQLCODE to warning code +236. Otherwise, SQLCODE is zero.
- USING BOTH was specified and the columns being described do not include LOBs or distinct types. Each column needs two

SQLVAR entries. DB2 sets SQLD to two times the number of columns of the result table. The second set of SQLVARs is used for the labels.

- 2 Each column needs two SQLVAR entries. Two entries per column are required when:
 - USING BOTH was not specified and the columns being described include LOBs or distinct types or both. DB2 sets the second set of SQLVAR entries with information for the LOBs or distinct types being described.
 - USING BOTH was specified and the columns include LOBs but not distinct types. DB2 sets the second set of SQLVAR entries with information for the LOBs and labels for the columns being described.
- 3 Each column needs three SQLVAR entries. Three entries are required only when USING BOTH is specified and the columns being described include distinct types. The presence of LOB data does not matter. It is the distinct types and not the LOBs that cause the need for three SQLVAR entries per column when labels are also requested. DB2 sets the second set of SQLVAR entries with information for the distinct types (and LOBs, if any) and the third set of SQLVAR entries with the labels of the columns being described.

A REXX SQLDA does not contain this field.

SQLDABC

The length of the SQLDA in bytes. DB2 sets the value to $SQLN \times 44 + 16$.

A REXX SQLDA does not contain this field.

SQLD If the prepared statement is a query, DB2 sets the value to the number of columns in the object being described (the value is actually twice the number of columns in the case where USING BOTH was specified and the result table does not include LOBs or distinct types). Otherwise, if the statement is not a query, DB2 sets the value to 0.

SQLVAR

An array of field description information for the column being described. There are two types of SQLVAR entries—the base SQLVAR and the extended SQLVAR.

If the value of SQLD is 0, or is greater than the value of SQLN, no values are assigned to any occurrences of SQLVAR. If the value of SQLN was set so that there are enough SQLVAR occurrences to describe the specified columns (columns with LOBs or distinct types and a request for labels increase the number of SQLVAR entries that are needed), the values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column, the second occurrence of SQLVAR contains a description of the second column, and so on. This first set of SQLVAR entries are referred to as *base SQLVAR* entries. Each column always has a base SQLVAR entry.

If the DESCRIBE statement included the USING BOTH clause, or the columns being described include LOBs or distinct types, additional SQLVAR entries are needed. These additional SQLVAR entries are referred to as the *extended SQLVAR* entries. There can be up to two sets of extended SQLVAR entries for each column.

For REXX, the SQLVAR is a set of stem variables that begin with *stem.n*, instead of a structure. The REXX SQLDA uses only a base SQLVAR. The way in which DB2 assigns values to the SQLVAR variables is the same as for other languages. That is, the *stem.1* variables describe the first column in the result table, the *stem.2* variables describe the second column in the result table, and so on. If USING BOTH is specified, the *stem.n+1* variables also describe the first column in the result table, the *stem.n+2* variables also describe the second column in the result table, and so on.

The base SQLVAR:

SQLTYPE

A code that indicates the data type of the column and whether the column can contain null values. For the possible values of SQLTYPE, see Table 146 on page 1666.

SQLLEN

A length value depending on the data type of the result columns. SQLLEN is 0 for LOB and XML data types. For the other possible values of SQLLEN, see Table 146 on page 1666.

In a REXX SQLDA, for DECIMAL or NUMERIC columns, DB2 sets the SQLPRECISION and SQLSCALE fields instead of the SQLLEN field.

SQLDATA

The CCSID of a string column. For possible values, see Table 147 on page 1668.

In a REXX SQLDA, DB2 sets the SQLCCSID field instead of the SQLDATA field.

SQLIND

Reserved.

SQLNAME

The unqualified name or label of the column, depending on the value of USING (NAMES, LABELS, ANY, or BOTH). The field is a string of length 0 if the column does not have a name or label. For more details on unnamed columns, see the discussion of the names of result columns under “select-clause” on page 633. This value is returned in the encoding scheme specified by the ENCODING bind option for the plan or package that contains the statement.

The extended SQLVAR:

SQLLONGLEN

The length attribute of a BLOB, CLOB, or DBCLOB column.

* Reserved.

SQLDATALEN

Not Used.

SQLDATATYPE-NAME

For a distinct type, the fully qualified distinct type name. Otherwise, the value is the fully qualified name of the built-in data type.

For a label, the label for the column.

This value is returned in the encoding scheme specified by the ENCODING bind option for the plan or package that contains this statement.

The REXX SQLDA does not use the extended SQLVAR.

Performance considerations: Although DB2 does not change the value of SQLN, you might want to reset this value after the DESCRIBE statement is executed. If the contents of SQLDA from the DESCRIBE statement is used in a later FETCH statement, set SQLN to n (where n is the number of columns of the result table) before executing the FETCH statement. For details, see *Preparing the SQLDA for data retrieval*.

Preparing the SQLDA for data retrievals: This note is relevant if you are applying DESCRIBE to a prepared query and you intend to use the SQLDA in the FETCH statements you employ to retrieve the result table rows. To prepare the SQLDA for that task, you must set the SQLDATA field of SQLVAR. SQLIND must be set if SQLTYPE is odd, and SQLNAME must be set when overriding the CCSID. For the meaning of those fields in that context, see “SQL descriptor area (SQLDA)” on page 1656.

Also, SQLN and SQLDABC should be reset (if necessary) to n and $n \times 44 + 16$, where n is the number of columns in the result table. Doing so can improve performance when the rows of the result table are fetched.

Support for extended dynamic SQL in a distributed environment: In a distributed environment where DB2 for z/OS is the server and the requester supports extended dynamic SQL, such as DB2 Server for VSE & VM, a DESCRIBE statement that is executed against an SQL statement in the extended dynamic package appears to DB2 as a DESCRIBE statement against a static SQL statement in the DB2 package. A DESCRIBE statement cannot normally be issued against a static SQL statement. However, a DESCRIBE against a static SQL statement that is generated by extended dynamic SQL executes without error if the package has been rebound after field DESCRIBE FOR STATIC on installation panel DSNTIP4 has been set to YES.

YES indicates that DB2 generates an SQLDA for the DESCRIBE at bind time so that DESCRIBE requests for static SQL statements can be satisfied at execution time. For more information, see *DB2 Installation Guide*.

Avoiding double preparation when using REOPT(ALWAYS) or REOPT(ONCE): If bind option REOPT(ALWAYS) or REOPT(ONCE) is in effect, DESCRIBE causes the statement to be prepared if it is not already prepared. If issued before an OPEN or an EXECUTE, the DESCRIBE causes the statement to be prepared without input variables. If the statement has input variables, the statement must be prepared again when it is opened or executed. When REOPT(ONCE) is in effect, the statement is always prepared twice even if there are no input variables. Therefore, to avoid preparing statements twice, issue the DESCRIBE after the OPEN. For non-cursor statements, open and fetch processing are performed on the EXECUTE. So, if a DESCRIBE must be issued, the statement will be prepared twice.

The use of a prepared statement for an EXPLAIN statement can cause duplicate entries in the explain tables when the prepared statement specifies the REOPT(ALWAYS) bind option and is executed using the jcc driver.

Errors occurring on DESCRIBE: In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/REOPT(ONCE) bind options can cause some errors that are normally issued during PREPARE processing to be issued on DESCRIBE.

Considerations for implicitly hidden columns: A DESCRIBED OUTPUT statement only returns information about implicitly hidden columns if the column (of a base table that is defined as implicitly hidden) is explicitly specified as part of the SELECT list of the final result table of the query described. If implicitly hidden columns are not part of the result table of a query, a DESCRIBE OUTPUT statement that returns information about that query will not contain information about any implicitly hidden columns.

Using host variables: If the DESCRIBE statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Example

In a PL/I program, execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR and then execute a DESCRIBE statement using that SQLDA. This is the second technique described in Allocating the SQLDA.

```
EXEC SQL BEGIN DECLARE SECTION;
      DCL STMT1_STR CHAR(200) VARYING;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLDA;
EXEC SQL DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;
... /* code to prompt user for a query, then to generate */
      /* a select-statement in the STMT1_STR */
EXEC SQL PREPARE STMT1_NAME FROM :STMT1_STR;
... /* code to set SQLN to zero and to allocate the SQLDA */
EXEC SQL DESCRIBE STMT1_NAME INTO :SQLDA;
... /* code to check that SQLD is greater than zero, to set */
      /* SQLN to SQLD, then to re-allocate the SQLDA */
EXEC SQL DESCRIBE STMT1_NAME INTO :SQLDA;
... /* code to prepare for the use of the SQLDA */
EXEC SQL OPEN DYN_CURSOR;
... /* loop to fetch rows from result table */
EXEC SQL FETCH DYN_CURSOR USING DESCRIPTOR :SQLDA;
.
.
.
```

DESCRIBE PROCEDURE

The DESCRIBE PROCEDURE statement gets information about the result sets returned by a stored procedure. The information, such as the number of result sets, is put into a descriptor.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

►►—DESCRIBE PROCEDURE—*procedure-name*
host-variable—INTO—*descriptor-name*—◄◄

Description

procedure-name **or** *host-variable*

Identifies the stored procedure that returned one or more result sets. When the DESCRIBE PROCEDURE statement is executed, the procedure name must identify a stored procedure that the requester has already invoked using the SQL CALL statement. The procedure name can be specified as a one, two, or three-part name. The procedure name in the DESCRIBE PROCEDURE statement must be specified the same way that it was specified on the CALL statement. For example, if a two-part procedure name was specified on the CALL statement, you must specify a two-part procedure name in the DESCRIBE PROCEDURE statement.

If a host variable is used:

- It must be a character string variable with a length attribute that is not greater than 254.
- It must not be followed by an indicator variable.
- The value of the host variable is a specification that depends on the database server. Regardless of the server, the specification must:
 - Be left justified within the host variable
 - Not contain embedded blanks
 - Be padded on the right with blanks if its length is less than that of the host variable

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA). The information returned in the SQLDA describes the result sets returned by the stored procedure.

Considerations for allocating and initializing the SQLDA are similar to those for DESCRIBE TABLE.

The contents of the SQLDA after executing a DESCRIBE PROCEDURE statement are:

- The first 5 bytes of the SQLDAID field are set to 'SQLPR'.
A REXX SQLDA does not contain SQLDAID.
- Bytes 6 to 8 of the SQLDAID field are reserved.
- The SQLD field is set to the total number of result sets. A value of 0 in the field indicates there are no result sets.
- There is one SQLVAR entry for each result set.
- The SQLDATA field of each SQLVAR entry is set to the result set locator value associated with the result set.
For a REXX SQLDA, SQLLOCATOR is set to the result set locator value.
- The SQLIND field of each SQLVAR entry is set to the estimated number of rows in the result set
- The SQLNAME field is set to the name of the cursor used by the stored procedure to return the result set. This value is returned in the encoding scheme specified by the ENCODING bind option for the plan or package that contains this statement.

Notes

SQLDA information: A value of -1 in the SQLIND field indicates that an estimated number of rows in the result set is not provided. DB2 for z/OS always sets SQLIND to -1.

DESCRIBE PROCEDURE does not return information about the parameters expected by the stored procedure.

Assignment of locator values: Locator values are assigned to the SQLVAR entries in the SQLDA in the order that the associated cursors are opened at run time. Locator values are not provided for cursors that are closed when control is returned to the invoking application. If a cursor was closed and later re-opened before returning to the invoking application, the most recently executed OPEN CURSOR statement for the cursor is used to determine the order in which the locator values are returned for the procedure result sets. For example, assume procedure P1 opens three cursors A, B, C, closes cursor B and then issues another OPEN CURSOR statement for cursor B before returning to the invoking application. The locator values are assigned in the order A, C, B.

Alternatively, an ASSOCIATE LOCATORS statement can be used to copy the locator values to result set locator variables.

Using host variables: If the DESCRIBE PROCEDURE statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1: Place information about the result sets returned by stored procedure P1 into the descriptor named by SQLDA1. Assume that the stored procedure is called with a one-part name from current server SITE2.

```
EXEC SQL CONNECT TO SITE2;
EXEC SQL CALL P1;
EXEC SQL DESCRIBE PROCEDURE P1 INTO :SQLDA1;
```

Example 2: Repeat the scenario in Example 1, but use a two-part name to specify an explicit schema name for the stored procedure to ensure that stored procedure P1 in schema MYSCHEMA is used.

```
EXEC SQL CONNECT TO SITE2;  
EXEC SQL CALL MYSCHEMA.P1;  
EXEC SQL DESCRIBE PROCEDURE MYSCHEMA.P1 INTO :SQLDA1;
```

Example 3: Place information about the result sets returned by the stored procedure identified by host variable HV1 into the descriptor named by SQLDA2. Assume that host variable HV1 contains the value SITE2.MYSCHEMA.P1 and the stored procedure is called with a three-part name.

```
EXEC SQL CALL SITE2.MYSCHEMA.P1;  
EXEC SQL DESCRIBE PROCEDURE :HV1 INTO :SQLDA2;
```

The preceding example would be invalid if host variable HV1 had contained the value MYSCHEMA.P1, a two-part name. For the example to be valid with that two-part name in host variable HV1, the current server must be the same as the location name that is specified on the CALL statement as the following statements demonstrate. This is the only condition under which the names do not have to be specified the same way and a three-part name on the CALL statement can be used with a two-part name on the DESCRIBE PROCEDURES statement.

```
EXEC SQL CONNECT TO SITE2;  
EXEC SQL CALL SITE2.MYSCHEMA.P1;  
EXEC SQL ASSOCIATE LOCATORS (:LOC1, :LOC2)  
      WITH PROCEDURE :HV1;
```

DESCRIBE TABLE

The DESCRIBE TABLE statement obtains information about a designated table or view.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

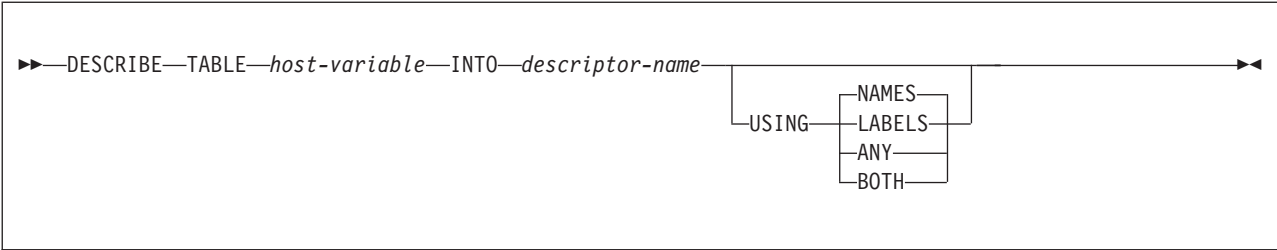
The privileges that are held by the authorization ID that owns the plan or package must include at least one of the following (if there is a plan, authorization checking is done only against the plan owner):

- Ownership of the table or view
- The SELECT, INSERT, UPDATE, DELETE, or REFERENCES privilege on the object
- The ALTER or INDEX privilege on the object (tables only)
- DBADM authority over the database that contains the object (tables only)
- SYSADM or SYSCTRL authority

| If the database is implicitly created, the database privileges must be on the implicit
| database or on DSNDB04.

For an RRSF application that does not have a plan and in which the requester and the server are DB2 for z/OS systems, authorization to execute the package is performed against the primary or secondary authorization ID of the process.

Syntax



Description

TABLE *host-variable*

Identifies the table or view. The name must not identify an auxiliary table. When the DESCRIBE statement is executed, the host variable must contain a name which identifies a table or view that exists at the current server. This variable must be a fixed-length or varying-length character string with a length attribute less than 256. The name must be followed by one or more blanks if the length of the name is less than the length of the variable. It cannot contain a period as the first character and it cannot contain embedded blanks. In addition, the quotation mark is the escape character regardless of the value of the string delimiter option. An indicator variable must not be specified for the host variable.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA), which is described in “SQL descriptor area (SQLDA)” on page 1656. See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C.

For languages other than REXX: Before the DESCRIBE statement is executed, the user must set the following variable in the SQLDA and the SQLDA must be allocated.

SQLN Indicates the number of SQLVAR occurrences provided in the SQLDA. DB2 does not change this value. For techniques to determine the number of required occurrences, see Allocating the SQLDA.

For REXX: The SQLDA is not allocated before it is used. An SQLDA consists of a set of stem variables. There is one occurrence of variable *stem.SQLD*, followed by zero or more occurrences of a set of variables that is equivalent to an SQLVAR structure. Those variables begin with *stem.n*.

After the DESCRIBE statement is executed, all the fields in the SQLDA except SQLN are either set by DB2 or ignored. For information on the contents of the fields, see The SQLDA contents returned after DESCRIBE.

USING

Indicates what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to a length of 0.

NAMES

Assigns the name of the column. This is the default.

LABELS

Assigns the label of the column. (Column labels are defined by the LABEL statement.)

ANY

Assigns the column label, and if the column has no label, the column name.

BOTH

Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $2 \times n$ or $3 \times n$, where n is the number of columns in the object being described. For each of the columns, the first n occurrences of SQLVAR, which are the base SQLVAR entries, contain the column names. Either the second or third n occurrences of SQLVAR, which are the extended SQLVAR entries, contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

For a declared temporary table, the name of the column is assigned regardless of the value specified in the USING clause because declared temporary tables cannot have labels.

Notes

See “DESCRIBE OUTPUT” on page 1243 for information about the following topics:

- Allocating the SQLDA
- The SQLDA contents that are returned on DESCRIBE

- Performance considerations for DESCRIBE
- Using host variables in DESCRIBE statements

| *Considerations for implicitly hidden columns:* A DESCRIBE TABLE statement does
| return information about implicitly hidden columns in tables.

DROP

The DROP statement removes an object at the current server. Except for storage groups, any objects that are directly or indirectly dependent on that object are deleted. Whenever an object is deleted, its description is deleted from the catalog at the current server, and any plans or packages that refer to the object are invalidated.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

To drop a table, table space, or index, the privilege set that is defined below must include at least one of the following:

- Ownership of the object (for an index, the owner is the owner of the table or index)
- DBADM authority
- SYSADM or SYSCTRL authority

If the table space is in a database that is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

To drop an alias, storage group, or view, the privilege set that is defined below must include at least one of the following:

- Ownership of the object
- SYSADM or SYSCTRL authority

To drop a role or a trusted context, the privilege set that is defined below must include at least one of the following:

- Ownership of the object
- SYSADM or SYSCTRL authority

To drop a database, the privilege set that is defined below must include at least one of the following:

- The DROP privilege on the database
- DBADM or DBCTRL authority for the database
- SYSADM or SYSCTRL authority

If the database is implicitly created, the privileges must be on the implicit database or on DSNDB04.

To drop a package, the privilege set that is defined below must include at least one of the following:

- Ownership of the package
- The BINDAGENT privilege granted from the package owner
- PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority

To drop a synonym, the privilege set that is defined below must include ownership of the synonym.

To drop a distinct type, stored procedure, trigger, user-defined function, or sequence, the privilege set that is defined below must include at least one of the following:

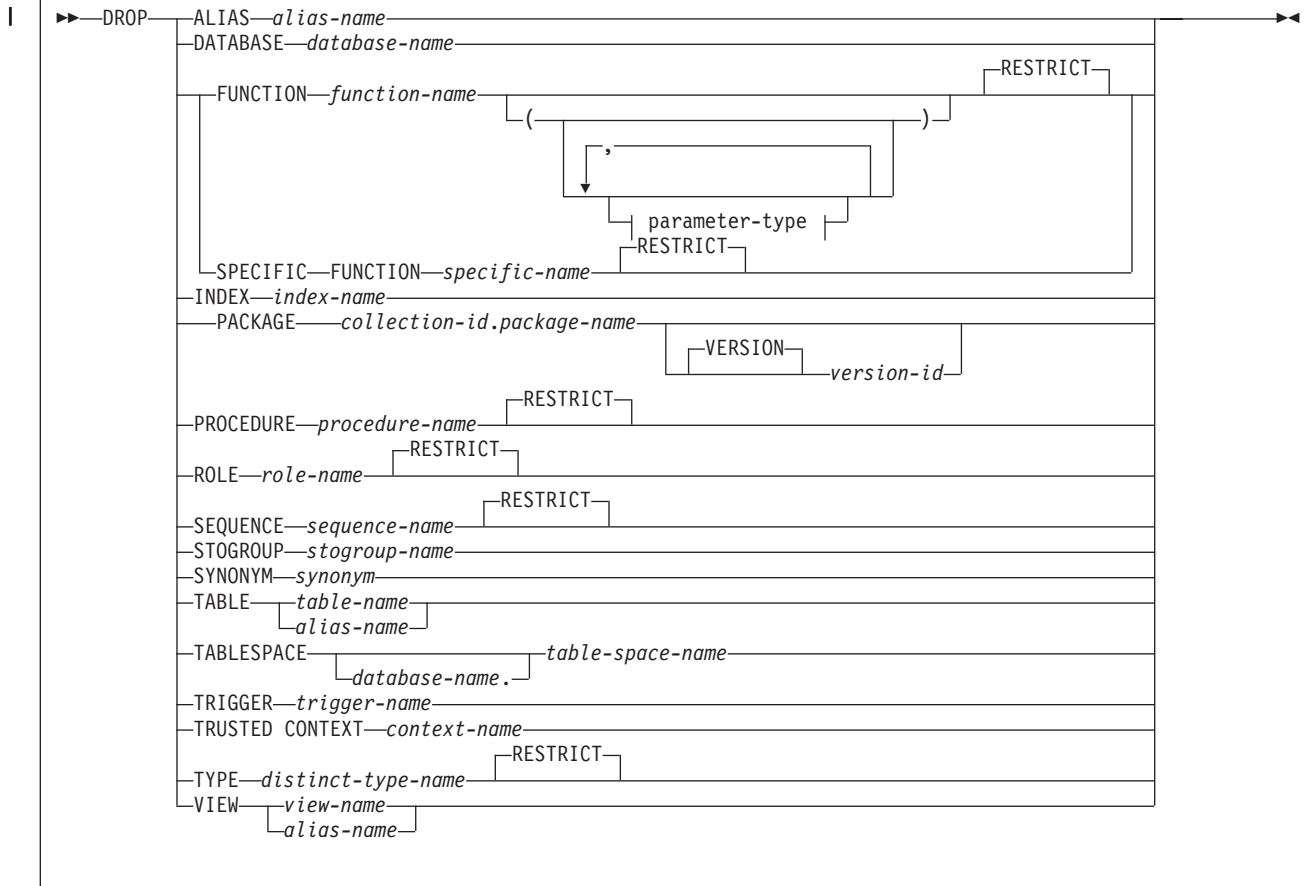
- Ownership of the object ³³
- The DROPIN privilege on the schema
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the DROPIN privilege on the schema.

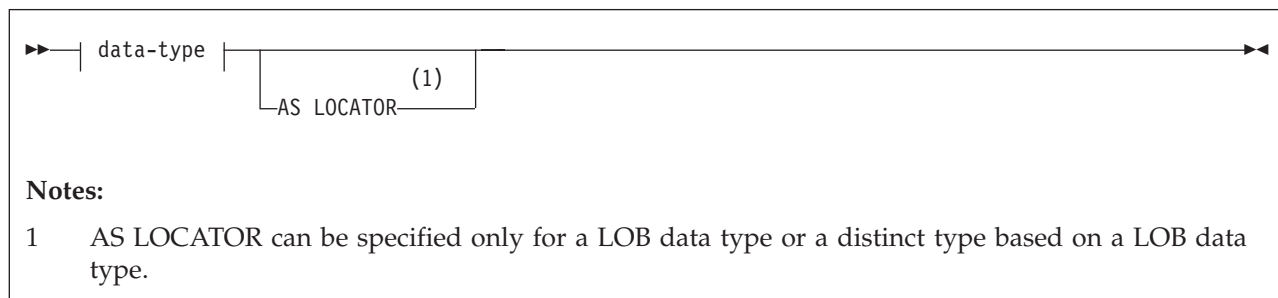
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process. If running in a trusted context with a role, the privilege set also includes those privileges that are held by the role that is associated with the primary authorization ID. However, the implicit schema match does not apply to the role when determining if DROPIN schema privilege is held.

33. Not applicable for stored procedures defined in releases of DB2 for z/OS prior to Version 6.

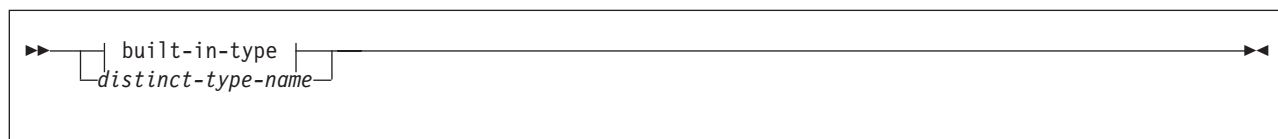
Syntax



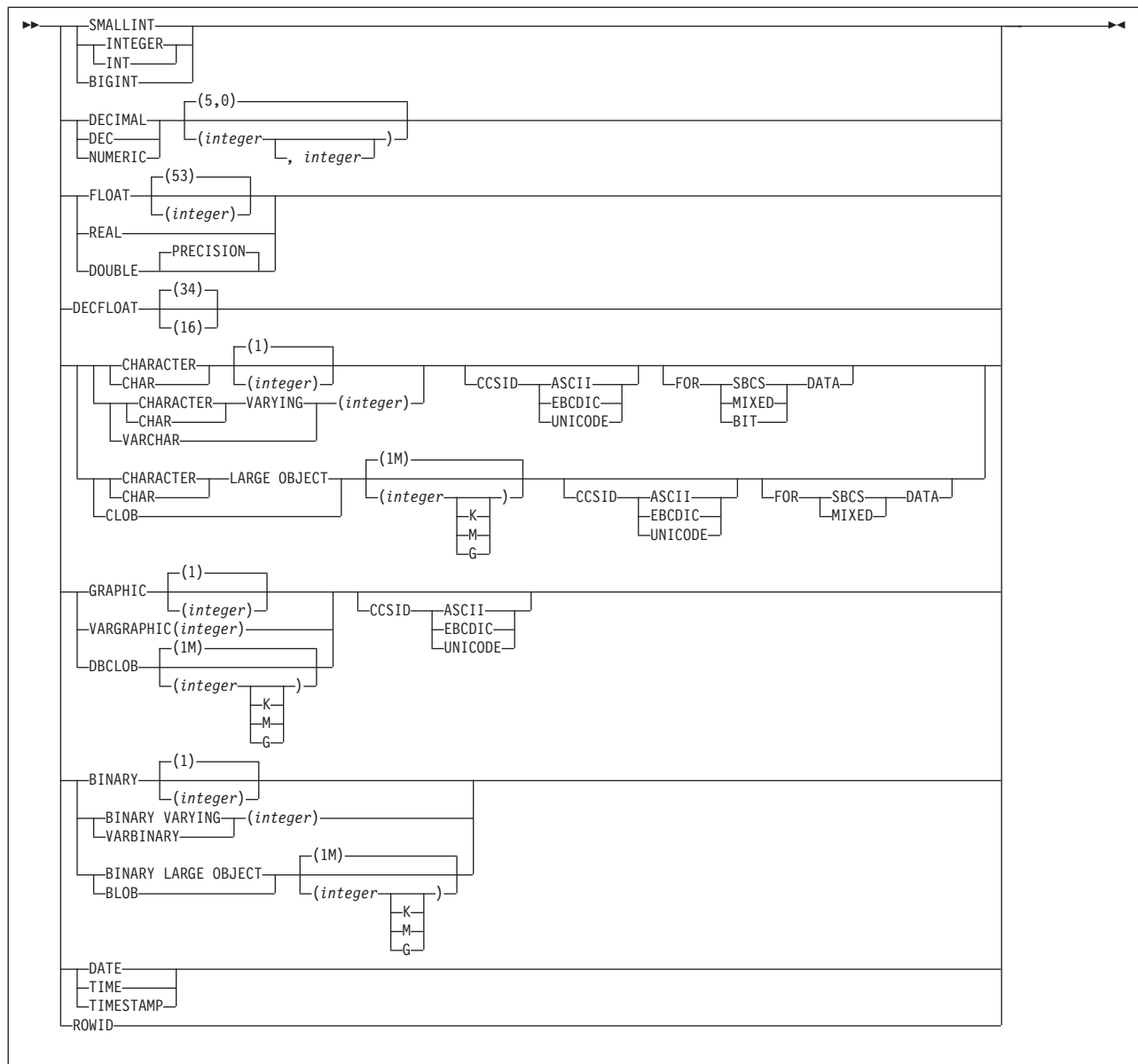
parameter type:



data type:



built-in-type:



Description

ALIAS *alias-name*

Identifies the alias to drop. The name must identify an alias that exists at the current server. Dropping an alias has no effect on any view, materialized query table, or synonym that was defined using the alias.

DATABASE *database-name*

Identifies the database to drop. The name must identify a database that exists at the current server. DSNCB04 or DSNCB06 must not be specified. The privilege set must include SYSADM authority.

Whenever a database is dropped, all of its table spaces, tables, index spaces, and indexes are also dropped.

FUNCTION or SPECIFIC FUNCTION

Identifies the function to drop. The function must exist at the current server,

and it must have been defined with the CREATE FUNCTION statement. The particular function can be identified by its name, function signature, or specific name.

Functions that are implicitly generated by the CREATE TYPE statement cannot be dropped using the DROP statement. They are implicitly dropped when the distinct type is dropped.

As indicated by the default keyword RESTRICT, the function is not dropped if any of the following dependencies exist:

- Another function is sourced on the function.
- A view uses the function.
- A trigger package uses the function.
- The definition of a materialized query table uses the function.

When a function is dropped, all privileges on the function are also dropped. Any plans or packages that are dependent on the function dropped are made inoperative.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function can have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is to be dropped. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly

DECFLOAT() will be considered a match for DECFLOAT(16) or DECFLOAT(34). However, FLOAT cannot be specified with empty parentheses because its parameter value indicates a specific data type (REAL or DOUBLE).

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the FOR *subtype* DATA clause or the CCSID clause is optional. Omission of either clause indicates that DB2 ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX *index-name*

Identifies the index to drop. The name must identify a user-defined index that exists at the current server but must not identify a populated index on an auxiliary table or an index that was implicitly created for a table that contains an XML column. (For details on dropping user-defined indexes on catalog tables, see “SQL statements allowed on the catalog” on page 1689.) A populated index on an auxiliary table can only be dropped by dropping the base table. The name must not identify an auxiliary table for an object that is involved in a clone relationship.

If the index that is dropped was created by specifying the ENDING AT clause to define partition boundaries, the table is converted to use table-controlled partitioning. The high limit key for the last partition is set to the highest possible value for ascending key columns or the lowest possible value for descending key columns.

Whenever an index is directly or indirectly dropped, its index space is also dropped. The name of a dropped index space cannot be reused until a commit operation is performed.

If the index is a unique index used to enforce a unique constraint (primary or unique key), the unique constraint must be dropped before the index can be dropped. In addition, if a unique constraint supports a referential constraint, the index cannot be dropped unless the referential constraint is dropped.

However, a unique index (for a unique key only) can be dropped without first dropping the unique key constraint if the unique key was created in a release

of DB2 before Version 7 and if the unique key constraint has no associated referential constraints. For information about dropping constraints, see “ALTER TABLE” on page 789.

If the table space is explicitly created and a unique index is dropped and that index was defined on a ROWID column that is defined as GENERATED BY DEFAULT, the table can still be used, but rows cannot be inserted into that table.

If the table space is implicitly created, the index cannot be dropped if it is defined on a ROWID column that is defined as GENERATED BY DEFAULT.

If an empty index on an auxiliary table is dropped, the base table is marked incomplete. If the base table space is implicitly created, the index on an auxiliary table cannot be dropped.

Drop index will result in the deletion of rows in the SYSCOLDIST and SYSCOLDISTATS catalog tables if no other indexes on the table have the same column group in their key sequence prefix.

PACKAGE *collection-id.package-name*

Identifies the package version to drop. The name plus the implicitly or explicitly specified *version-id* must identify a package version that exists at the current server. Omission of the *version-id* is an implicit specification of the null version.

The name must not identify a trigger package or a package that is associated with an SQL routine. A trigger package can only be dropped by dropping the associated trigger or subject table. A package that is associated with a native SQL procedure can only be dropped with an ALTER PROCEDURE statement with a DROP VERSION clause that specifies the particular version that is to be dropped, or with a DROP PROCEDURE statement if it is the only version that is defined for the procedure.

Specify this clause to drop a package that is created as the result of a BIND COPY command used to deploy a version of a native SQL procedure.

If a package has current, previous, and original copies, the DROP statement will drop all copies.

VERSION *version-id*

version-id is the version identifier that was assigned to the package's DBRM when the DBRM was created. If *version-id* is not specified, a null version is used as the version identifier.

Delimit the version identifier when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

For more on version identifiers, see the information on preparing an application program for execution in *DB2 Application Programming and SQL Guide*.

PROCEDURE *procedure-name*

Identifies the stored procedure to drop. The name must identify a stored procedure that has been defined with the CREATE PROCEDURE statement at the current server.

All versions of the native SQL procedure are dropped; all privileges on the procedure are also dropped. In addition, any plans or packages that are dependent on the procedure are marked invalid.

If the procedure is a native SQL procedure, use an ALTER PROCEDURE statement with the DROP VERSION clause to drop a specific version of a procedure. Use a DROP PACKAGE statement to drop a package for a version of the procedure that is created using the BIND COPY command.

ROLE *role-name*

Identifies the role to drop. *role-name* must identify a role that exists at the current server.

When a role is dropped, all privileges and authorities that have been previously granted to that role are revoked. If the role that is dropped is the owner of statements in the dynamic statement cache, the cached statements are invalidated.

The role is not dropped if any REVOKE restrictions are encountered. REVOKE restrictions include those that are encountered during the cascading of the revocation of a role's privileges. If RESTRICT is specified, the role is not dropped if any of the following dependencies exist:

- The role is associated with any trusted context or any user in a trusted context.
- The role is associated with a currently running thread.
- The role is the owner of any of the following objects:

Alias	Sequence
	Storage group
Database	Stored procedure
Distinct type	Table
Index	Table space
a JAR file	Trigger
Materialized query table	Trusted context
Plans	User-defined function
Packages	View
Role	

SEQUENCE *sequence-name*

Identifies the sequence to drop. The name must identify an existing sequence at the current server.

sequence-name must not be the name of an internal sequence object that is used by DB2 (including an implicitly generated sequence for a DB2_GENERATED_DOCID_FOR_XML column). Sequences that are generated by the system for identity columns or implicitly created databases cannot be dropped by using the DROP SEQUENCE statement. A sequence object for an identity column is implicitly dropped when the table that contains the identity column is dropped.

The default keyword RESTRICT indicates that the sequence is not dropped if any of the following dependencies exist:

- A trigger that uses the sequence in a NEXT VALUE or PREVIOUS VALUE expression exists.
- An inline SQL function that uses the sequences in a NEXT VALUE or PREVIOUS VALUE expression exists.

Whenever a sequence is dropped, all privileges on the sequence are also dropped, and the plans and packages that refer to the sequence are invalidated. Dropping a sequence, even if the drop process is rolled back, results in the loss of the still-unassigned cache values for the sequence.

STOGROUP *stogroup-name*

Identifies the storage group to drop. The name must identify a storage group that exists at the current server but not a storage group that is used by any table space or index space.

For information on the effect of dropping the default storage group of a database, see [Dropping a default storage group](#).

SYNONYM *synonym*

Identifies the synonym to drop. In a static DROP SYNONYM statement, the name must identify a synonym that is owned by the owner of the plan or package. In a dynamic DROP SYNONYM statement, the name must identify a synonym that is owned by the SQL authorization ID. Thus, using interactive SQL, a user with SYSADM authority can drop any synonym by first setting CURRENT SQLID to the owner of the synonym.

Dropping a synonym has no effect on any view, materialized query table, or alias that was defined using the synonym, nor does it invalidate any plans or packages that use such views, materialized query tables, or aliases.

TABLE *table-name or alias-name*

Identifies the table to drop. The name must identify a table that exists at the current server but must not identify a catalog table, a table in a partitioned table space, a table that is implicitly created for an XML column, or a populated auxiliary table. A table in a partitioned table space can only be dropped by dropping the table space. A populated auxiliary table or a table that is implicitly created for an XML column can only be dropped by dropping the associated base table.

If *alias-name* is specified, the actual table will be dropped as if *table-name* were specified. However, the alias is not dropped and can be dropped using the DROP ALIAS statement.

Whenever a table is directly or indirectly dropped, all privileges on the table, all referential constraints in which the table is a parent or dependent, and all synonyms, views, and indexes defined on the table are also dropped. If the table space for the table was implicitly created, it is also dropped.

Whenever a table is directly or indirectly dropped, all materialized query tables defined on the table are also dropped. Whenever a materialized query table is directly or indirectly dropped, all privileges on the materialized query table and all synonyms, views, and indexes that are defined on the materialized query table are also dropped. Any alias defined on the materialized query table is not dropped. Any plans or packages that are dependent on the dropped materialized query table are marked invalid.

You cannot use DROP TABLE to drop a clone table. You must use the ALTER TABLE statement with the DROP CLONE clause to drop a clone table. If a base table that is involved in a clone relationship is dropped, the associated clone table is also dropped. You cannot drop an auxiliary table for an object that is involved in a clone relationship.

If a table with LOB columns is dropped, the auxiliary tables associated with the table and the indexes on the auxiliary tables are also dropped. Any LOB table spaces that were implicitly created for the auxiliary tables are also dropped.

If a table with XML columns is dropped, all implicitly created objects for all XML columns are also dropped.

If an empty auxiliary table is dropped, the definition of the base table is marked incomplete. If the base table space is implicitly created, the auxiliary table cannot be dropped.

If the table has a security label column, the primary authorization ID of the DROP statement must have a valid security label, and the RACF SECLABEL class must be active.

TABLESPACE *database-name.table-space-name*

Identifies the table space to drop. The name must identify a table space that exists at the current server. The database name must not be DSNDB06. Omission of the database name is an implicit specification of DSNDB04. *table-space-name* must not identify a table space that is implicitly created for an XML column.

Whenever a table space is directly or indirectly dropped, all the tables in the table space are also dropped. The name of a dropped table space cannot be reused until a commit operation is performed.

A LOB table space can be dropped only if it does not contain an auxiliary table. If the LOB table space is implicitly created, it cannot be dropped.

Whenever a base table space that contains tables with LOB columns is dropped, all the auxiliary tables and indexes on those auxiliary tables that are associated with the base table space are also dropped.

Whenever a base table space that contains tables with XML columns is dropped, all implicitly created objects for all XML columns are also dropped.

TRIGGER *trigger-name*

Identifies the trigger to drop. The name must identify a trigger that exists at the current server.

Whenever a trigger is directly or indirectly dropped, all privileges on the trigger are also dropped and the associated trigger package is freed. The name of that trigger package is the same as the trigger name and the collection ID is the schema name.

When an INSTEAD OF trigger is dropped, the associated privilege is revoked from anyone that possesses the privilege as a result of an implicit grant that occurred when the trigger is created.

Dropping triggers causes certain packages to be marked invalid. For example, if *trigger-name* specifies an INSTEAD OF trigger on a view V, another trigger might depend on *trigger-name* through an update to the view V, and that trigger package is invalidated.

If a trigger has current, previous, and original copies, the DROP statement will drop all copies.

TRUSTED CONTEXT *context-name*

Identifies the trusted context to drop. The *context-name* must identify a trusted context that exists at the current server. When a trusted context is dropped, all associations to attributes (IP addresses, job names) and associations to users of the trusted context are dropped. If the trusted context is dropped while trusted connections for the context are active, the connections remain active until they terminate or the next attempt at reuse is made.

TYPE *distinct-type-name*

Identifies the distinct type to drop. The name must identify a distinct type that exists at the current server. The default keyword RESTRICT indicates that the distinct type is not dropped if any of the following dependencies exist:

- The definition of a column of a table uses the distinct type.
- The definition of an input or result parameter of a user-defined function uses the distinct type.
- The definition of a parameter of a stored procedure uses the distinct type.
- The definition of an extended index uses a cast function that is implicitly generated for the distinct type.
- A sequence exists for which the data type of the sequence is the distinct type.
- One of the following dependencies exists on one of the cast functions that are generated for the distinct type:
 - Another function is sourced from one of the cast functions
 - A view uses one of the cast functions
 - A trigger package uses one of the cast functions
 - The definition of a materialized query table uses one of the cast functions

Whenever a distinct type is dropped, all privileges on the distinct type are also dropped. In addition, the cast functions that were generated when the distinct type was created and the privileges on those cast functions are also dropped.

VIEW *view-name* **or** *alias-name*

Identifies the view to drop. The name must identify a view that exists at the current server.

Whenever a view is directly or indirectly dropped, all privileges on the view and all synonyms and views that are defined on the view are also dropped. Whenever a view is directly or indirectly dropped, all materialized query tables defined on the view are also dropped.

If *alias-name* is specified, the actual view will be dropped as if *view-name* were specified. However, the alias is not dropped and can be dropped using the DROP ALIAS statement.

Notes

Restrictions on DROP: DROP is subject to these restrictions:

- DROP DATABASE cannot be performed while a DB2 utility has control of any part of the database.
- DROP INDEX cannot be performed while a DB2 utility has control of the index or its associated table space.
- DROP TABLE cannot be performed while a DB2 utility has control of the table space that contains the table.
- DROP TABLESPACE cannot be performed while a DB2 utility has control of the table space.

In a data sharing environment, the following restrictions also apply:

- If any member has an active resource limit specification table (RLST) you cannot drop the database or table space that contains the table, the table itself, or any index on the table.
- If the member executing the drop cannot access the DB2-managed data sets, only the catalog and directory entries for those data sets are removed.

Objects that have certain dependencies cannot be dropped. For information on these restrictions, see Table 117 on page 1270.

Recreating objects: After an index or table space is dropped, a commit must be performed before the object can be recreated with the same name. If a table that was created without an IN clause (thereby causing a table space to be implicitly created) is dropped, a table cannot be recreated with the same name until a commit is performed.

Dropping a parent table: DROP is not DELETE and therefore does not involve delete rules.

Dropping a default storage group: If you drop the default storage group of a database, the database no longer has a legitimate default. You must then specify USING in any statement that creates a table space or index in the database. You must do this until you either:

- Create another storage group with the same name using the CREATE STOGROUP statement, or
- Designate another default storage group for the database using the ALTER DATABASE statement.

Dropping a table space or index: To drop a table space or index, the size of the buffer pool associated with the table space or index must not be zero.

Dropping a LOB table space: If the base table space is explicitly created, both explicitly created LOB table spaces and implicitly created LOB table spaces can be dropped if it does not contain any auxiliary tables. If the LOB table space is implicitly created, it will be dropped automatically when the auxiliary table is dropped. If the LOB table space is explicitly created, it is not dropped when the auxiliary table is dropped, and can be explicitly dropped later.

If the base table space is implicitly created, the LOB table space cannot be dropped. If the LOB table space is explicitly created, it can be dropped when the auxiliary table is dropped. The following table shows the relationship between the base table space, the LOB table space, and the use of DROP for the LOB table space and base table space:

Table 115. Use of DROP for LOB table space

How base table was created	How LOB table space was created	Whether DROP can be used on LOB table space	State of LOB table space if base table space is dropped
Explicitly	Explicitly	Yes	LOB table space remains
Explicitly	Implicitly	Yes	LOB table space is dropped
Implicitly	Explicitly	Yes	LOB table space remains
Implicitly	Implicitly	No	N/A

Dropping a database when data sets for DB2 objects have already been deleted: When some of the data sets for DB2 objects that associated with the database have already been deleted, DROP DATABASE will perform in the following manner:

For DB2-managed objects:

The DROP DATABASE statement will delete the underlying data sets if they exist. If the data sets do not exist, DROP DATABASE will delete only the catalog entries for those data sets.

For user-managed objects:

The DROP DATABASE statement will delete only the catalog entries for the data sets. The underlying data sets will need to be manually deleted after the DROP DATABASE statement is complete.

Dropping a table space in a work file database: If one member of a data sharing group drops a table space in a work file database, or an entire work file database, that belongs to another member, DB2-managed data sets that the executing member cannot access are not dropped. However, the catalog and directory entries for those data sets are removed.

Dropping resource limit facility (governor) indexes, tables, and table spaces: While the RLST is active, you cannot issue a DROP DATABASE, DROP INDEX, DROP TABLE, or DROP TABLESPACE statement for an object associated with an RLST that is active on any member of a data sharing group. See *DB2 Performance Monitoring and Tuning Guide* for details.

Dropping a temporary table: To drop a created temporary table or a declared temporary table, use the DROP TABLE statement.

Dropping a materialized query table: To drop a materialized query table, use the DROP TABLE statement.

Dropping an alias: Dropping a table or view does not drop its aliases. However, if you use the DROP TABLE statement and specify an alias for a table or view, the table or view will be dropped. To drop an alias, use the DROP ALIAS statement.

Dropping a table from an implicitly created table space: If you drop a table from an implicitly created table space, the following related objects are also dropped:

- The enforcing primary and unique key indexes
- Any LOB table spaces, auxiliary tables, and auxiliary indexes
- The ROWID index (if the ROWID column is defined as GENERATED BY DEFAULT)

If any LOB columns are defined on the table, the LOB table space is dropped if it was implicitly created. You can use the DROP statement to drop a LOB table space only if one of the following conditions is true:

- The base table space is explicitly created
- The base table space is implicitly created but the LOB table space is explicitly created

You cannot use the DROP statement to drop a LOB table space if both the base table space and the LOB table space are implicitly created.

Dropping an index on an auxiliary table and an auxiliary table: You can explicitly drop an empty index on an auxiliary table with the DROP INDEX statement, unless the base table space is implicitly created. An empty or populated index on an auxiliary table is implicitly dropped when:

- The auxiliary table is empty and it is explicitly dropped (empty indexes only).
- The associated base table for the auxiliary table is dropped.
- The base table space that contains the associated base table is dropped.

You can explicitly drop an empty auxiliary table with the DROP TABLE statement, unless the base table space is implicitly created. An empty or populated auxiliary table is implicitly dropped when:

- The associated base table for the auxiliary table is dropped.
- The base table space that contains the associated base table is dropped.

The following table shows which DROP statements implicitly or explicitly cause an auxiliary table and the index on that table to be dropped, as indicated by the 'D' in the column.

Table 116. Effect of various DROP statements on auxiliary tables and indexes that are in explicitly created table spaces

Statement	Auxiliary table		Index on auxiliary table	
	Populated	Empty	Populated	Empty
DROP TABLESPACE (base table space)	D	D	D	D
DROP TABLE (base table)	D	D	D	D
DROP TABLE (auxiliary table)		D		D
DROP INDEX (index on auxiliary table)				D

Note: D indicates that the table or index is dropped.

Dropping a migrated index or table space: Here, “migration” means migrated by the Hierarchical Storage Manager (DFSMSHsm). DB2 does not wait for any recall of the migrated data sets. Hence, recall is not a factor in the time it takes to execute the statement.

Dropping a trusted context: The drop of a trusted context takes effect after the DROP TRUSTED CONTEXT statement is committed. If the DROP TRUSTED CONTEXT statement results in an error or is rolled back, the trusted context is not dropped.

Deleting SYSLGRNG records for dropped table spaces: After dropping a table space, you cannot delete the associated records. If you want to remove the records, you must quiesce the table space, and then run the MODIFY RECOVERY utility *before* dropping the table space. If you delete the SYSLGRNG records and drop the table space, you cannot reclaim the table space.

Dependencies when dropping objects: Whenever an object is directly or indirectly dropped, other objects that depend on the dropped object might also be dropped. (The catalog stores information about the dependencies of objects on each other.) The following semantics determine what happens to a dependent object when the object that it depends on (the underlying object) is dropped:

Cascade (D)

Dropping the underlying object causes the dependent object to be dropped. However, if the dependent object cannot be dropped because it has a restrict dependency on another object, the drop of the underlying object fails.

Restrict (D)

The underlying object cannot be dropped if a dependent object exists.

Inoperative (O)

Dropping the underlying object causes the dependent object to become inoperative.

Invalidation (V)

Dropping the underlying object causes the dependent object to become invalidated.

For objects that directly depend on others, the following table uses the letter abbreviations above to summarize what happens to a dependent object when its underlying object is specified in a DROP statement. Additional objects can be indirectly affected, too.

To determine the indirect effects of a DROP statement, assess what happens to the dependent object and whether the dependent object has objects that depend on it. For example, assume that view B is defined on table A and view C is defined on view B. In the following table, the 'D' in the VIEW column of the DROP TABLE row indicates that view B is dropped when table A is dropped. Next, because view C is dependent on view B, check the VIEW column for DROP VIEW. The 'D' in the column indicates that view C will be dropped, too.

The letters in the following table have the following meanings:

- D** Dependent object is dropped.
- O** Dependent object is made inoperative.
- V** Dependent object is invalidated.
- R** DROP statement fails.

Table 117. Effect of dropping objects that have dependencies

	Type of object												
	A	L	I	A	S	E	N	X	E ¹	E	E	P	M
DROP ALIAS													
DROP DATABASE													
DROP FUNCTION													
DROP INDEX ^{2,6}													
DROP PACKAGE ⁷													
DROP PROCEDURE													
DROP ROLE													
DROP SEQUENCE													
DROP STOGROUP													
DROP SYNONYM													
DROP TABLE ^{9,10}													

Table 117. Effect of dropping objects that have dependencies (continued)

	Type of object												
DROP statement	A	D	F	I	P	R	S	S	T	A	B	T	V
	L	A	U	N	C	O	E	T	S	L	T		
	I	A	C	I	C	E	U	G	N	S	I		
	A	S	O	E	G	R	C	U	Y	A	G	T	I
	S	E	N	X	E ¹	E	E	P	M	E	R	E	W
DROP TABLESPACE ¹²				D	V					D			
DROP TRIGGER					V ¹⁶								
DROP TYPE			R ³	R ¹⁴		R ⁴	R			R			
DROP VIEW					V				D		D ¹⁵		D

Note:

1. The PACKAGE column represents packages for user-defined functions, procedures, and triggers, as well as other packages. The PACKAGE column also applies for plans.
2. The index space associated with the index is dropped.
3. If a function is dependent on the distinct type being dropped, the distinct type cannot be dropped unless the function is one of the cast functions that was created for the distinct type.
4. If the definition of a parameter of a stored procedure uses the distinct type, the distinct type cannot be dropped.
5. If other user-defined functions are sourced on the user-defined function being dropped, the function cannot be dropped.
6. An index on an auxiliary table cannot be explicitly dropped.
7. A trigger package cannot be explicitly dropped with DROP PACKAGE. A trigger package is implicitly dropped when the associated trigger or subject table is dropped.
8. A storage group cannot be dropped if it is used by any table space or index space.
9. An auxiliary table cannot be explicitly dropped with DROP TABLE. An auxiliary table is implicitly dropped when the associated base table is dropped.
10. If an implicit table space was created when the table was created, the table space is also dropped.
11. When a subject table is dropped, any associated triggers and related trigger packages are also dropped.
12. A LOB table space cannot be dropped until the base table with the LOB columns is dropped.
13. This restriction is only for SQL functions.
14. The index in this case must be defined on an expression.
15. When a subject view is dropped, any associated triggers and related trigger packages are also dropped.
16. Any packages that have a dependency on an INSTEAD OF trigger will be marked invalid.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- DATA TYPE or DISTINCT TYPE as a synonym for TYPE
- PROGRAM as a synonym for PACKAGE

Examples

Example 1: Drop table DSN8910.DEPT.

```
DROP TABLE DSN8910.DEPT;
```

Example 2: Drop table space DSN8S91D in database DSN8D91A.

```
DROP TABLESPACE DSN8D91A.DSN8S91D;
```

Example 3: Drop the view DSN8910.VPROJRE1:

```
DROP VIEW DSN8910.VPROJRE1;
```

Example 4: Drop the package DSN8CC0 with the version identifier VERSZZZZ. The package is in the collection DSN8CC61. Use the version identifier to distinguish the package to be dropped from another package with the same name in the same collection.

```
DROP PACKAGE DSN8CC61.DSN8CC0 VERSION VERSZZZZ;
```

Example 5: Drop the package DSN8CC0 with the version identifier "1994-07-14-09.56.30.196952". When a version identifier is generated by the VERSION(AUTO) precompiler option, delimit the version identifier.

```
DROP PACKAGE DSN8CC61.DSN8CC0 VERSION "1994-07-14-09.56.30.196952";
```

Example 6: Drop the distinct type DOCUMENT, if it is not currently in use:

```
DROP TYPE DOCUMENT;
```

Example 7: Assume that you are SMITH and that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Drop ATOMIC_WEIGHT.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT;
```

Example 8: Assume that you are SMITH and that you created the function CENTER in schema SMITH. Drop CENTER, using the function signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTER(INTEGER, FLOAT);
```

Example 9: Assume that you are SMITH and that you created another function named CENTER, which you gave the specific name FOCUS97, in schema JOHNSON. Drop CENTER, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION JOHNSON.FOCUS97;
```

Example 10: Assume that you are SMITH and that stored procedure OSMOSIS is in schema BIOLOGY. Drop OSMOSIS.

```
DROP PROCEDURE BIOLOGY.OSMOSIS;
```

Example 11: Assume that you are SMITH and that trigger BONUS is in your schema. Drop BONUS.

```
DROP TRIGGER BONUS;
```

Example 12: Drop the role CTXROLE:

```
DROP ROLE CTXROLE;
```

Example 13: Drop the trusted context CTX1:

```
DROP TRUSTED CONTEXT CTX1;
```

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of an SQL declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax

►►—END DECLARE SECTION—◄◄

Description

The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear in accordance with the rules of the host language. It is used to indicate the end of an SQL declare section. An SQL declare section starts with a BEGIN DECLARE SECTION statement described in “BEGIN DECLARE SECTION” on page 872.

The following rules are enforced by the precompiler only if the host language is C or the STDSQL(YES) precompiler option is specified:

- A variable referred to in an SQL statement must be declared within an SQL declare section of the source program.
- BEGIN DECLARE SECTION and END DECLARE SECTION statements must be paired and must not be nested.
- SQL declare sections can contain only host variable declarations, SQL INCLUDE statements that include host variable declarations, or DECLARE VARIABLE statements.

Notes

SQL declare sections are only required if the STDSQL(YES) option is specified or the host language is C. However, SQL declare sections can be specified for any host language so that the source program can conform to IBM SQL. If SQL declare sections are used, but not required, variables declared outside an SQL declare section should not have the same name as variables declared within an SQL declare section.

Example

```
EXEC SQL BEGIN DECLARE SECTION;

-- host variable declarations

EXEC SQL END DECLARE SECTION;
```

EXCHANGE

The EXCHANGE statement switches the content of a base table and its associated clone table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set that is defined below must include at least one of the following privileges:

- The INSERT and DELETE privileges on both the base table and the clone table
- Ownership of the both the base table and the clone table
- DBADM authority for the database
- SYSADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Syntax

►►—EXCHANGE DATA BETWEEN TABLE—*table-name1*—AND—*table-name2*—◄◄

Description

table-name1 and *table-name2*

Identifies the base table and the associated clone table for which the exchange of data will take place. Either *table-name1* or *table-name2* can identify the base table. The other table name must identify a clone table that is associated with the specified base table. The name of the base table and the name of the clone table remain unchanged after a data exchange.

Notes

Rules and restrictions: When a data exchange is done, real time statistics for the base table are invalidated. Data exchanges cannot be done for a subset of a table's partitions. There must be a commit between consecutive data exchanges using the EXCHANGE statement.

Examples

Example: Exchange the data of the EMPLOYEE table and its clone table, EMPCLONE.

```
EXCHANGE DATA BETWEEN TABLE EMPCLONE AND EMPLOYEE;
```

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

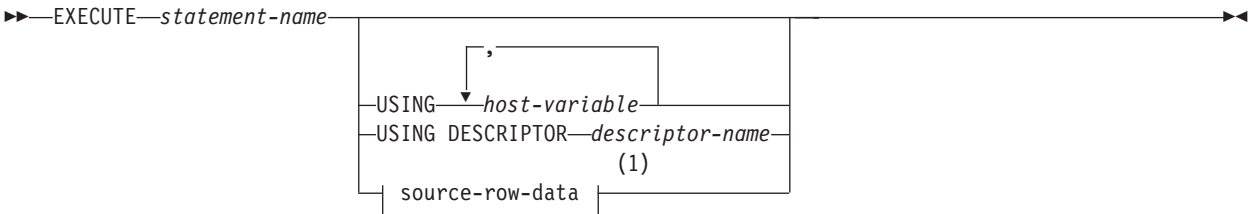
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See “PREPARE” on page 1405 for the authorization required to create a prepared statement.

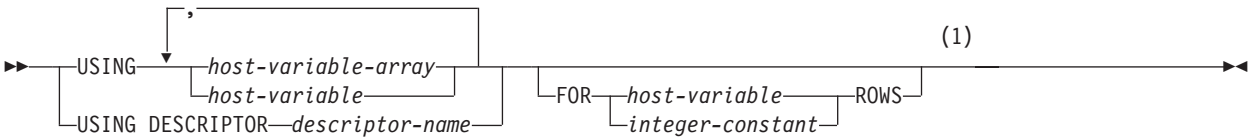
Syntax



Notes:

- 1 This option can be specified only when *statement-name* refers to a dynamic INSERT or MERGE statement that is prepared with FOR MULTIPLE ROWS and is specified as part of the ATTRIBUTES clause on the PREPARE statement.

source-row-data:



Notes:

- 1 The FOR n ROWS clause is required on the EXECUTE statement if it is not specified as part of the MERGE statement and a host variable array is specified. The FOR n ROWS clause is also required if MERGE is used with multiple rows of source data. For an INSERT statement, the FOR n ROWS clause can only be specified for a dynamic statement that contains only a single multiple-row INSERT statement.

Description

statement-name

Identifies the prepared statement to be executed. *statement-name* must identify a

statement that was previously prepared within the unit of work and the prepared statement must not be a SELECT statement.

USING

Introduces a list of host variables whose values are substituted for the parameter markers (question marks) in the prepared statement. (For an explanation of parameter markers, see “PREPARE” on page 1405.) If the prepared statement includes parameter markers, you must include USING in the EXECUTE statement. USING is ignored if there are no parameter markers.

For more on the substitution of values for parameter markers, see Parameter marker replacement.

host-variable,...

Identifies structures or variables that must be described in the application program in accordance with the rules for declaring host structures and variables. A reference to a structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable supplies the value for the *n*th parameter marker in the prepared statement.

USING DESCRIPTOR *descriptor-name*

Identifies an SQLDA that contains a valid description of the input host variables.

Before invoking the EXECUTE statement, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences that are provided in the SQLDA
A REXX SQLDA does not contain this field.
- SQLABC to indicate the number of bytes of storage that are allocated for the SQLDA
- SQLD to indicate the number of variables that are used in the SQLDA when processing the statement
- SQLVAR entries to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR entries. If an SQLVAR entry includes a LOB value or a distinct type based on a LOB, there must be additional SQLVAR entries for each parameter. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR entries, see “SQL descriptor area (SQLDA)” on page 1656.

SQLD must be set to a value that is greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement.

See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C.

source-row-data

The prepared statement must be an INSERT or MERGE statement for which the FOR MULTIPLE ROWS clause is specified as part of the ATTRIBUTES clause on the PREPARE statement.

USING *host-variable-array* **or** *host-variable*

Introduces a list of host variables or host variable arrays whose values are substituted for the parameter markers (question marks) in the prepared INSERT or MERGE statement. The number of columns specified in the INSERT

or MERGE statement must be less than or equal to the total number of host variables or host variable arrays that are specified.

host-variable-array

Identifies a host-variable array that must be defined in the application program in accordance with the rules for declaring a host variable array. A reference to a structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable supplies the value for the *n*th parameter marker in the prepared statement.

host-variable

Identifies a variable that must be described in the application program in accordance with the rules for declaring host variables.

USING DESCRIPTOR *descriptor-name*

Identifies an SQLDA that must contain a valid description of the host variable arrays or host variables that contain the values to insert.

Before invoking the EXECUTE statement for a dynamic INSERT or MERGE statement, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR entries that are provided in the SQLDA.
- SQLABC to indicate the number of bytes of storage that are allocated for the SQLDA.
- SQLD to indicate the number of variables, plus one, that are used in the SQLDA that provide values for columns that are the source of the INSERT or MERGE statement. SQLD must be set to a value that is greater than or equal to zero and less than or equal to SQLN.
- SQLVAR entries to indicate the attributes of an element of the host variable array for the SQLVAR entries that correspond to values that are provided for the source columns of the INSERT or MERGE statement. Within each SQLVAR, the following fields are set:
 - SQLTYPE indicates the data type of the elements of the host variable array.
 - SQLDATA points to the corresponding host variable array.
 - SQLEN and SQLLONGLEN indicate the length of a single element of the array.
- SQLNAME, the fifth and sixth bytes must contain a flag field and the seventh and eighth bytes must contain a binary small integer (halfword) that contains the dimension of the host-variable array and, if specified, the corresponding indicator array.

The SQLDA must have enough storage to contain a SQLVAR entry for each target column for which values are provided, plus an additional SQLVAR entry for the number of rows. The DB2 system generates code to fill in the required information for this extra SQLVAR entry. Each SQLVAR entry describes a host variable, host variable array, or buffer that contains the values for a column of the source table. The last SQLVAR entry contains the number of rows of data. For example, if the INSERT or MERGE statement is providing values for five columns of the target table, six SQLVAR entries must be provided. If any value is a LOB value, twice as many SQLVAR entries must be provided, and SQLN must be set to the number of SQLVAR entries. Thus, if the INSERT or MERGE statement is providing values for five columns of the source table, and some of the values to insert are LOB values, 12 SQLVAR entries must be provided.

The SQLVAR entry for the number of rows must also contain a flag value. See “Field descriptions of an occurrence of a base SQLVAR” on page 1662 for more information.

You set the SQLDATA and SQLIND pointers to the beginning of the corresponding arrays.

FOR *host-variable* or *integer-constant* ROWS

Specifies the number of rows of source data. The values for the insert or merge operation are specified in the USING clause.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with a scale of zero and must not include an indicator variable. *k* must be in the range 0 to 32767.

FOR *n* ROWS cannot be specified on the EXECUTE statement if the statement being processed is a dynamic INSERT or MERGE statement that includes a FOR *n* ROWS clause.

Notes

DB2 can stop the execution of a prepared SQL statement if the statement is taking too much processor time to finish. When this happens, an error occurs. The application that issued the statement is not terminated; it is allowed to issue another SQL statement.

Parameter marker replacement: Before the prepared statement is executed, each parameter marker in the statement is effectively replaced by its corresponding host variable. The replacement is an assignment operation in which the source is the value of the host variable and the target is a variable within DB2. The assignment rules are those described for assignment to a column in “Assignment and comparison” on page 102. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Parameter markers.

Let *V* denote a host variable that corresponds to parameter marker *P*. The value of *V* is assigned to the target variable for *P* in accordance with the rules for assigning a value to a column:

- *V* must be compatible with the target.
- If *V* is a string, its length must not be greater than the length attribute of the target.
- If *V* is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of *V* are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, *V* must not be null.

When the prepared statement is executed, the value used in place of *P* is the value of the target variable for *P*. For example, if *V* is CHAR(6) and the target is CHAR(8), the value used in place of *P* is the value of *V* padded on the right with two blanks.

Errors occurring on EXECUTE: In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/REOPT(ONCE) bind options can cause

some errors that are normally issued during PREPARE processing to be issued on EXECUTE.

Examples

Example 1: In this example, an INSERT statement with parameter markers is prepared and executed. S1 is a structure that corresponds to the format of DSN8910.DEPT.

```
EXEC SQL PREPARE DEPT_INSERT FROM
    'INSERT INTO DSN8910.DEPT VALUES(?,?,?,?)';
-- Check for successful execution and read values into S1
EXEC SQL EXECUTE DEPT_INSERT USING :S1;
```

Example 2: Assume that the IWH.PROGPARM table has 9 columns. Prepare and execute a dynamic INSERT statement that inserts 5 rows of data into the IWH.PROGPARM table. The values to be inserted are provided in arrays, where all the values for a column are provided in an host-variable-array with the EXECUTE statement.

```
STMT = 'INSERT INTO IWH.PROGPARM (IWHID, UPDATE_BY, UPDATE_TS, NAME,
                                SHORT_DESCRIPTION, ORDERNO, PARMDATA,
                                PARMDATALONG, VWPROGKEY)
VALUES ( ?, ?, ?, ?, ?, ?, ?, ?, ? )';
ATTRVAR = 'FOR MULTIPLE ROWS';
EXEC SQL PREPARE INS_STMT ATTRIBUTES :ATTRVAR FROM :STMT;
NROWS = 5;
EXEC SQL EXECUTE INS_STMT FOR :NROWS ROWS
    USING :V1, :V2, :V3, :V4, :V5, :V6, :V7, :V8, :V9;
```

In this example, each host variable in the USING clause represents an array of values for the corresponding column of the target of the INSERT statement.

Example 3: Using dynamically supplied values for an employee row, update the master EMPLOYEE table if the data is for an existing employee or insert a new row if the data is for a new employee.

```
hv_stmt =
    "MERGE INTO EMPLOYEE AS T
    USING (VALUES (CAST (? AS CHAR(6)), CAST (? AS VARCHAR(12)),
                  CAST (? AS CHAR(1)), CAST (? AS VARCHAR(15)),
                  CAST (? AS INTEGER)))
    AS S (EMPNO, FIRSTNAME, MI, LASTNAME, SALARY)
    ON T.EMPNO = S.EMPNO
    WHEN MATCHED THEN UPDATE
        SET SALARY = S.SALARY
    WHEN NOT MATCHED THEN INSERT (EMPNO, FIRSTNAME, MI, LASTNAME, SALARY)
        VALUES (S.EMPNO, S.FIRSTNAME, S.MI, S.LASTNAME, S.SALARY)
    NOT ATOMIC CONTINUE ON SQLEXCEPTION";
hv_attr = 'FOR MULTIPLE ROWS';
EXEC SQL
    PREPARE merge_stmt
    ATTRIBUTES :hv_attr FROM :hv_stmt;
hv_nrows = 5;
/* Initialize the hostvar array of hv_empno, hv_firstname... */
EXEC SQL
    EXECUTE merge_stmt
    USING :hv_empno, :hv_firstname, :hv_mi,
        :hv_lastname, :hv_salary
    FOR :hv_nrows ROWS;
```

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. It can be used to prepare and execute an SQL statement that contains neither host variables nor parameter markers.

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a string form of the statement
- Executes the SQL statement
- Destroys the executable form

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The authorization rules are those defined for the dynamic preparation of the SQL statement specified by EXECUTE IMMEDIATE. For example, see “INSERT” on page 1367 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

Syntax

►► EXECUTE IMMEDIATE host-variable string-expression ►►

Description

host-variable

host-variable must be specified. It must identify a host variable that is described in the application program in accordance with the rules for declaring character or graphic string variables. If the source string is over 32KB in length, the *host-variable* must be a CLOB or DBCLOB variable. The maximum source length is 2MB although the host variable can be declared larger than 2MB. An indicator variable must not be specified. In Assembler, C, COBOL, and PL/I, the host variable must be a varying-length string variable. In C, it must not be a NUL-terminated string. In SQL PL, an SQL variable is used in place of a host variable and the value must not be null.

string-expression

string-expression is any PL/I expression that yields a string. *string-expression* cannot be preceded by a colon. Variables that are within *string-expression* that include operators or functions should not be preceded by a colon. When *string-expression* is specified, the precompiler-generated structures for *string-expression* use an EBCDIC CCSID and an informational message is returned.

Notes

Allowable SQL statements: The value of the identified host variable or the specified *string-expression* is called the *statement string*.

The statement string must be one of the following SQL statements, and cannot be the *select-statement*:

ALLOCATE CURSOR	REVOKE
ALTER	ROLLBACK
ASSOCIATE LOCATORS	SAVEPOINT
COMMENT	SET CURRENT DEGREE
COMMIT	SET CURRENT DECFLOAT ROUNDING MODE
CREATE	SET CURRENT DEBUG MODE
DECLARE GLOBAL TEMPORARY	SET CURRENT LOCALE LC_CTYPE
TABLE	SET CURRENT MAINTAINED TABLE TYPES
DELETE	FOR OPTIMIZATION
DROP	SET CURRENT OPTIMIZATION HINT
EXPLAIN	SET CURRENT PRECISION
FREE LOCATOR	SET CURRENT REFRESH AGE
GRANT	SET CURRENT ROUTINE VERSION
HOLD LOCATOR	SET CURRENT RULES
INSERT	SET CURRENT SQLID
LABEL	SET ENCRYPTION PASSWORD
LOCK TABLE	SET PATH
MERGE	SET SCHEMA
REFRESH TABLE	SIGNAL
RELEASE SAVEPOINT	TRUNCATE
RENAME	UPDATE

The statement string must not:

- Begin with EXEC SQL
- End with END-EXEC or a semicolon
- Include references to variables
- Include parameter markers

Errors and error handling: When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is invalid, it is not executed and the error condition that prevents its execution is reported in the SQLCA. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the SQLCA.

DB2 can stop the execution of a prepared SQL statement if the statement is taking too much CPU time to finish. When this happens an error occurs. The application that issued the statement is not terminated; it is allowed to issue another SQL statement.

Performance considerations: If the same SQL statement is to be executed more than once, it is more efficient to use the PREPARE and EXECUTE statements rather than the EXECUTE IMMEDIATE statement.

Examples

Example 1: In this PL/I example, the EXECUTE IMMEDIATE statement is used to execute a DELETE statement in which the rows to be deleted are determined by a search-condition specified by the value of PREDS.

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM DSN8910.DEPT
WHERE' || PREDS;
```

Example 2: Use C to execute the SQL statement in the host variable Qstring.

```
EXEC SQL INCLUDE SQLCA;
void main ()
{
    EXEC SQL BEGINDECLARE SECTION;
    char Qstring[100M] =
        "INSERT INTO WORK TABLE SELECT * FROM EMPPROJACT WHERE ACTNO >= 100";
    EXEC SQL END DECLARE SECTION;
    .
    .
    .
    EXEC SQL EXECUTE IMMEDIATE :Qstring;
    return;
}
```

EXPLAIN

The EXPLAIN statement obtains information about access path selection for an *explainable statement*. A statement is explainable if it is a SELECT, MERGE, or INSERT statement, or the searched form of an UPDATE or DELETE statement. The information obtained is placed in a user-supplied *plan table*.

PSPI

Optionally, EXPLAIN can also obtain and place information in additional tables. A user-supplied *statement table* can be populated with information about the estimated cost of executing the explainable statement. A user-supplied *function table* can be populated with information about how DB2 resolves the user-defined functions that are referred to in the explainable statement. Other EXPLAIN tables can be populated with additional information about the execution of the explainable statement. For a complete list of EXPLAIN tables, see “EXPLAIN tables” on page 1929.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The authorization rules are those defined for the SQL statement specified in the EXPLAIN statement. For example, see the description of the DELETE statement for the authorization rules that apply when a DELETE statement is explained.

If the EXPLAIN statement is embedded in an application program, the authorization rules that apply are those defined for embedding the specified SQL statement in an application program. In addition, the owner of the plan or package must also have one of the following characteristics:

- Be the owner of a plan table named PLAN_TABLE
- Have an alias on a plan table named *owner*.PLAN_TABLE and have SELECT and INSERT privileges on the table

If the EXPLAIN statement is dynamically prepared, the authorization rules that apply are those defined for dynamically preparing the specified SQL statement. In addition, the SQL authorization ID of the process or the role this is associated with the process (if the EXPLAIN statement is running in a trusted context that specifies the ROLE AS OBJECT OWNER AND QUALIFIER clause) must also have one of the following characteristics:

- Be the owner of a plan table named PLAN_TABLE
- Have an alias on a plan table named *owner*.PLAN_TABLE and have SELECT and INSERT privileges on the table

The authorization rules are different if the STMTCACHE keyword is specified to have a cached statement explained. The privilege set must include at least one of the following:

- SYSADM authority
- The authority that is required to share the cached statement.

For the STMTCACHE ALL keyword, the privilege set must include at least one of the following:

- SYSADM authority to explain all statements in the dynamic statement cache

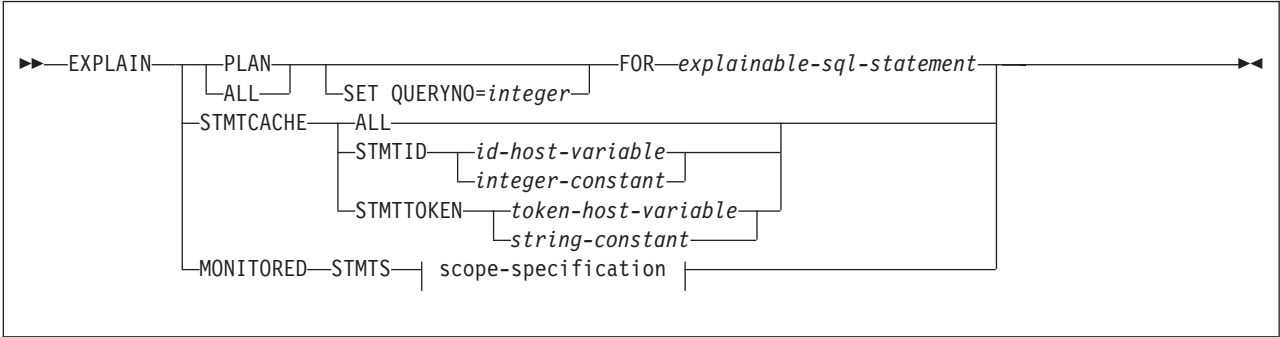
If the privilege set does not have the required authority, only those statements that have the same authorization ID as the privilege set are explained.

For the MONITOR STMTS keyword, the privilege set must include at least one of the following:

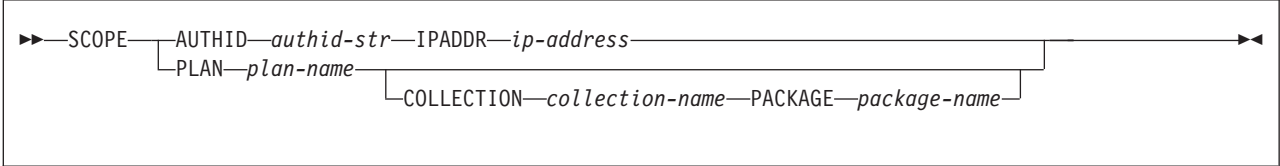
- SYSADM authority
- SYSOPR authority
- SYSCTRL authority

Privilege set: The privilege set comprises the union of authorities that are held by the authorization IDs of the process. If the process is running in a trusted context with a role, this role would be included as an authorization ID of the process.

Syntax



scope-specification:



Description

PLAN

Inserts one row into the plan table for each step used in executing *explainable-sql-statement*. The steps for enforcing referential constraints are not included. See “PLAN_TABLE” on page 1930 and for a description of the plan table.

If a statement table exists, one row that provides a cost estimate of processing the explainable statement is inserted into the statement table. If the explainable statement is a *SELECT FROM data-change-statement*, two rows are inserted into the statement table. See “DSN_STATEMNT_TABLE” on page 1985 for a description of the statement table.

If a function table exists, one row is inserted into the function table for each user-defined function that is referred to by the explainable statement. See “DSN_FUNCTION_TABLE” on page 1952 for a description of the function table.

If additional EXPLAIN tables exist, rows are also inserted into those tables. For a complete list of EXPLAIN tables, see “EXPLAIN tables” on page 1929.

ALL

Has the same effect as PLAN.

SET QUERYNO = *integer*

Associates *integer* with *explainable-sql-statement*. The column QUERYNO is given the value *integer* in every row inserted into the plan table, statement table, or function table by the EXPLAIN statement. If QUERYNO is not specified, DB2 itself assigns a number. For an embedded EXPLAIN statement, the number is the statement number that was assigned by the precompiler and placed in the DBRM.

FOR *explainable-sql-statement*

Specifies the SQL statement to be explained. *explainable-sql-statement* can be any explainable SQL statement. If EXPLAIN is embedded in a program, the statement can contain references to host variables. If EXPLAIN is dynamically prepared, the statement can contain parameter markers. Host variables that appear in the statement must be defined in the statement's program.

The statement must refer to objects at the current server.

explainable-sql-statement must not contain a QUERYNO clause. To specify the value of the QUERYNO column in plan table for the statement being explained, use the SET QUERYNO = clause of the EXPLAIN statement.

explainable-sql-statement cannot be a statement-name or a host-variable. To use EXPLAIN to get information about dynamic SQL statements, you must prepare the entire EXPLAIN statement dynamically.

To obtain information about an explainable SQL statement that references a declared temporary table, the EXPLAIN statement must be executed in the same application process in which the table was declared. For static EXPLAIN statements, the information is not obtained at bind-time but at runtime when the EXPLAIN statement is incrementally bound.

STMTCACHE

Specifies that statements in the dynamic statement cache are to be explained. In a data sharing environment, the statements in the dynamic statement cache of the data sharing member where EXPLAIN STMTCACHE is executed are explained.

ALL

Specifies that all of the cached statements are to be explained.

STMTCACHE ALL returns one row for each cached statement to the DSN_STATEMENT_CACHE_TABLE. These rows contain identifying information about the statements in the cache, as well as statistics that reflect the execution of the statements by all processes that have executed the statement. See “DSN_STATEMENT_CACHE_TABLE” on page 1981 for information about creating a statement cache table. See the Authorization

STMTID *id-host-variable* or *integer-constant*

Specifies that the cached statement with the specified statement ID is to be explained. The value contained *id-host-variable* or specified by *integer-constant* identifies the statement ID. The statement ID is an integer

that uniquely identifies a statement that has been cached in the dynamic statement cache. The statement ID of a cached statement can be retrieved through IFI monitor facilities from IFCID 316 or 124. Some diagnostic trace records, such as 173, 196, and 337, also show the statement ID.

For every row that the EXPLAIN statement inserts into the plan table, statement table, or function table, the QUERYNO column contains the value of the statement ID.

STMTTOKEN *id-host-variable* **or** *integer-constant*

Specifies that the cached statements with the specified statement token are to be explained. The value contained in *token-host-variable* or specified by *string-constant* identifies the statement token. The statement token must be a character string that is no longer than 240 bytes. The application program that originally prepares and inserts a statement into the cache associates a statement token with the cached statement. The program can make this association with the RRSAF SET_ID function, or the sqleseti API if the program is connected remotely.

For every row that the EXPLAIN statement inserts into the plan table, statement table, or function table, the STMTTOKEN column contains the value of the statement token, and the QUERYNO column contains the value of the statement ID for the cached statement with the statement token.

MONITORED STMTS

Specifies that monitored statements, either dynamic or static, are to be explained. The MONITORED STMTS clause is deprecated and might not be available in releases of DB2 for z/OS after Version 9.1.

SCOPE

Specifies the scope of the filter for the statements. Only statements that are within the specified scope or match the filter will be explained.

AUTHID *authid-string*

Specifies that only statements that are executed by an authorization ID that matches the authorization ID that is specified in *authid-string* are to be explained. *authid-string* is a string constant or a host variable that represents an authorization ID. Any authorization ID sees only the statements with the same authorization ID, except SYSADM. SYSADM sees all authorization IDs.

IPADDR *ip-address*

Specifies the IP address of the client that is used to connect to the DB2 server. *ip-address* is the IP address string or a host variable.

PLAN *plan-name*

Specifies that only statements under the specified *plan-name* are to be explained. *plan-name* is a string constant or a host variable that represents the plan name.

COLLECTION *collection-name*

Specifies that only statements under the specified *collection-name* are to be explained. *collection-name* is a string constant or a host variable representing the collection name.

PACKAGE *package-name*

Specifies that only statements under the specified *package-name* are to be explained. *package-name* is a string constant or a host variable representing the package name.

Notes

Output from EXPLAIN: DB2 inserts one or more rows of data into a plan table and other existing *EXPLAIN tables*. A plan table must exist before the operation that results in EXPLAIN output. For information about valid plan table formats see “PLAN_TABLE” on page 1930. You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESSC of the SDSNSAMP library.

Unless you need the information that is provided by the additional EXPLAIN tables, it is not necessary to create those tables to use EXPLAIN. However, a statement cache table is required when the STMTCACHE ALL keyword is specified as part of an EXPLAIN statement.

EXPLAIN tables: Each row in an EXPLAIN table describes some aspect of a step in the execution of a query or subquery in an explainable statement. The column values for the row identify, among other things, the query or subquery, the tables and other objects involved, the methods used to carry out each step, and cost information about those methods.

Instances of these tables might also be created and used by certain optimization tools. For information about the meanings of different values in plan table and other EXPLAIN tables, see .

Plan table

owner.PLAN_TABLE.

Function table

owner.DSN_FUNCTION_TABLE.

Statement table

owner.DSN_STATEMENT_TABLE.

Statement cache table

owner.DSN_STATEMENT_CACHE_TABLE.

Structure table

owner.DSN_STRUCT_TABLE.

Predicate table

owner.DSN_PREDICAT_TABLE.

Detailed cost table

owner.DSN_DETCOST_TABLE.

Sort table

owner.DSN_SORT_TABLE.

Sort key table

owner.DSN_SORTKEY_TABLE.

Filter table

owner.DSN_FILTER_TABLE.

Page range table

owner.DSN_PGRANGE_TABLE.

Parallel group table

owner.DSN_PGROUPTABLE.

Parallel task table

owner.DSN_PTASK_TABLE.

View reference table

owner.DSN_VIEWREF_TABLE.

Query table

owner.DSN_QUERY_TABLE.

Query information table

owner.DSN_QUERYINFO_TABLE.

Examples

Example 1: Determine the steps required to execute the query 'SELECT X.ACTNO...'. Assume that no set of rows in the PLAN_TABLE has the value 13 for the QUERYNO column.

```
EXPLAIN PLAN SET QUERYNO = 13
FOR SELECT X.ACTNO, X.PROJNO, X.EMPNO, Y.JOB, Y.EDLEVEL
FROM DSN8910.EMPPROJECT X, DSN8910.EMP Y
WHERE X.EMPNO = Y.EMPNO
AND X.EMPTIME > 0.5
AND (Y.JOB = 'DESIGNER' OR Y.EDLEVEL >= 12)
ORDER BY X.ACTNO, X.PROJNO;
```

Example 2: Retrieve the information returned in Example 1. Assume that a statement table exists, so also retrieve the estimated cost of processing the query. Use the following query, which joins the plan table and the statement table.

```
SELECT * FROM PLAN_TABLE A, DSN_STMTMNT_TABLE B
WHERE A.QUERYNO = 13 and B.QUERYNO = 13
ORDER BY A.QBLOCKNO, A.PLANNO, A.MIXOPSEQ;
```

Example 3: Have the cached statement with statement ID 124 explained. Assume that host variable SID contains 124.

```
EXPLAIN STMTCACHE STMTID :SID;
```

Example 4: Have one row of data for each statement in the dynamic statement cache written to the DSN_STATEMENT_CACHE_TABLE.

```
EXPLAIN STMTCACHE ALL;
```

Example 5: Assume that you want to use the plan table that was created by ADMF001 and your authorization ID is SYSADM. If you have an alias on ADMF001.PLAN_TABLE (CREATE ALIAS SYSADM.PLAN_TABLE FOR ADMF001.PLAN_TABLE) and sufficient INSERT and SELECT privileges on the table, the following EXPLAIN statement will execute and ADMF001.PLAN_TABLE will be populated.


```
EXPLAIN PLAN SET QUERYNO = 101
FOR SELECT * FROM DSN8910.EMP;
```




Related concepts


 Interpreting data access by using EXPLAIN (Performance Monitoring and Tuning Guide)

Related tasks

 Checking how DB2 resolves functions by using DSN_FUNCTION_TABLE (Application programming and SQL)


 Finding information about statements in the statement cache (Application programming and SQL)

 Installation step 18: Create and bind objects for the Optimization Service Center, Optimization Expert and IBM Data Studio Developer: DSNTIJOS (optional) (DB2 Installation Guide)

 Migration step 24: Create and bind objects for the Optimization Service Center, Optimization Expert, and IBM Data Studio Developer: DSNTIJOS (optional) (DB2 Installation Guide)

 Capturing EXPLAIN information (Performance Monitoring and Tuning Guide)

Related reference

 BIND and REBIND options (DB2 Commands)

 EXPLAIN tables (Performance Monitoring and Tuning Guide)

 DSNAEXP stored procedure (Performance Monitoring and Tuning Guide)

FETCH

The FETCH statement positions a cursor on a row of its result table. It can return zero, one, or multiple rows and assigns the values of the rows to host variables if there is a target specification.

There are two forms of this statement:

- **Single row fetch:** positions the cursor and, optionally, retrieves data from a single row of the result table.
- **Multiple row fetch:** positions the cursor on zero or more rows of the result table and, optionally, returns data if there is a target specification.

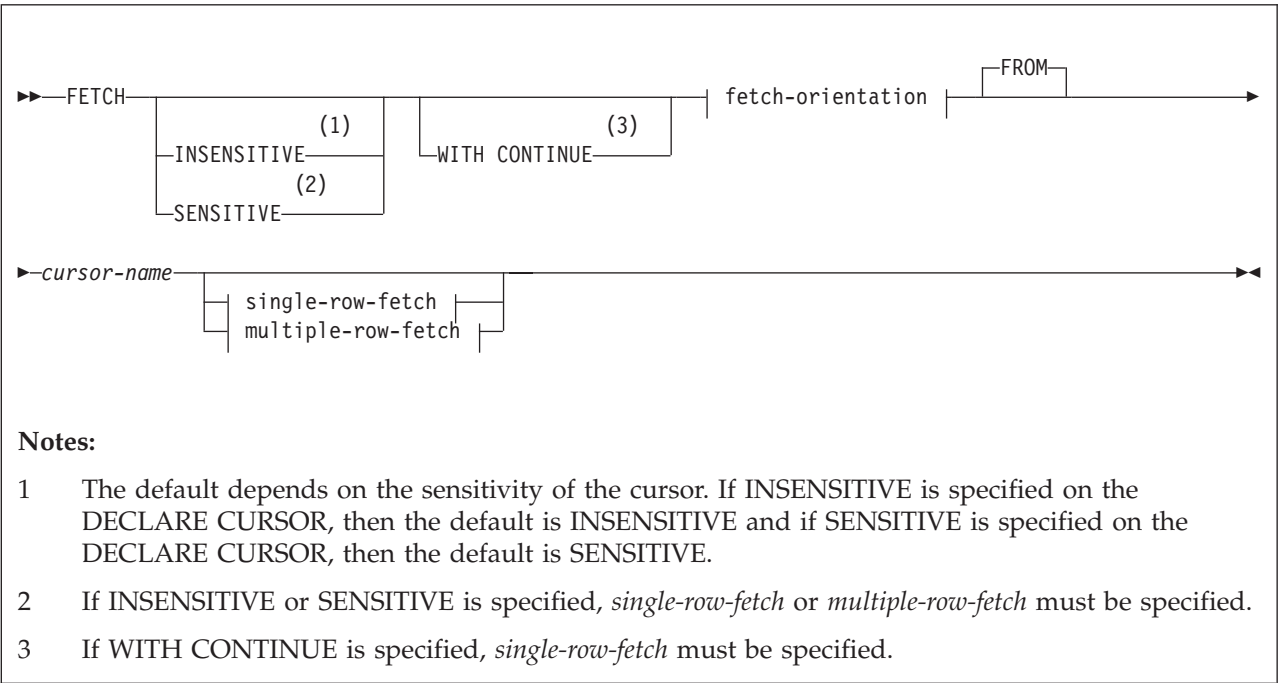
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. Multiple row fetch is not supported in REXX, Fortran³⁴, or SQL Procedure applications.

Authorization

See “DECLARE CURSOR” on page 1191 for an explanation of the authorization required to use a cursor.

Syntax



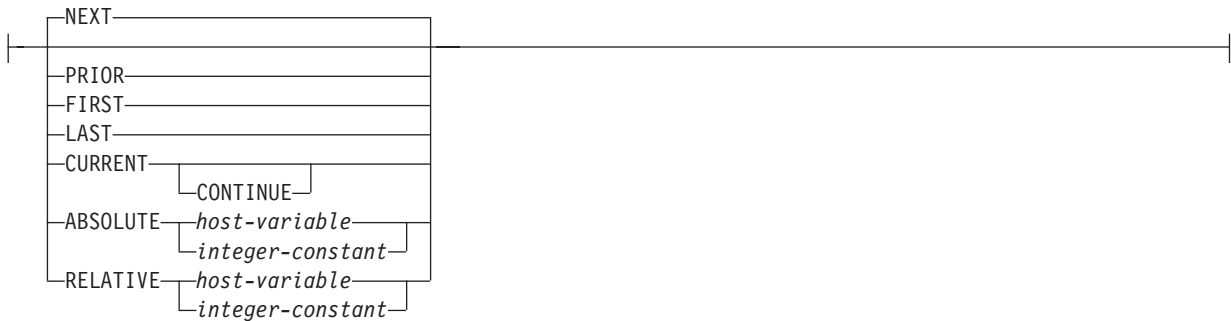
fetch-orientation

34. ASSEMBLER and other languages are supported, but this support is limited to statements that allow USING DESCRIPTOR. The precompiler does not recognize host-variable-arrays except in C/C++, COBOL, and PL/I.

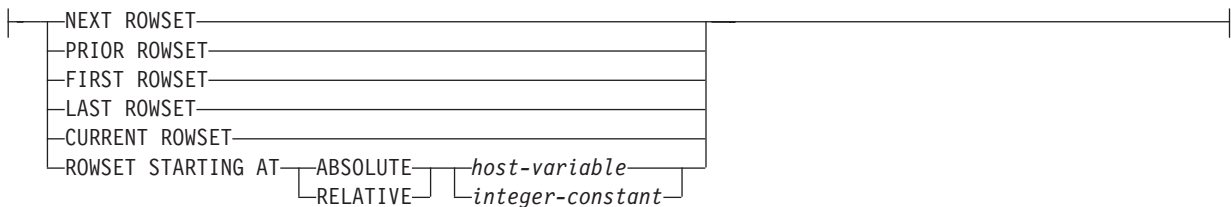
fetch-orientation:



row-positioned:



rowset-positioned:

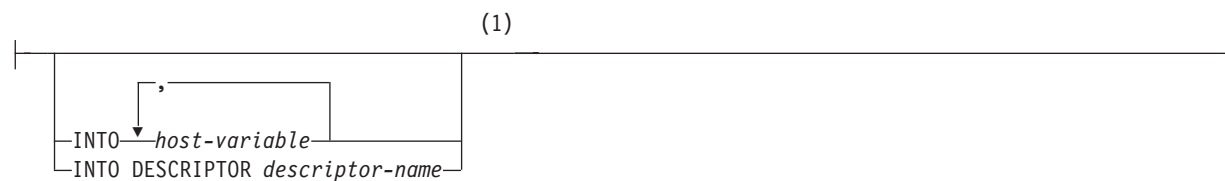


Notes:

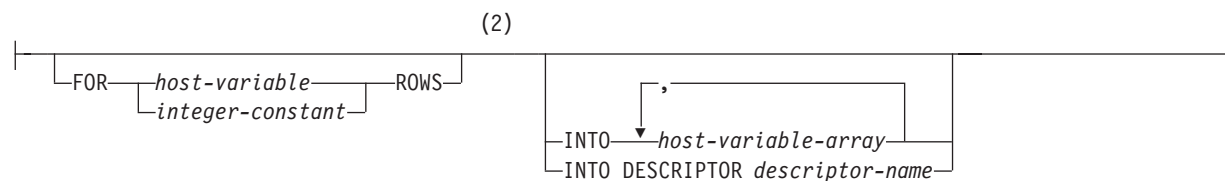
- 1 If BEFORE or AFTER is specified, SENSITIVE, INSENSITIVE, single-row-fetch, or multiple-row-fetch must not be specified.

fetch-type

single-row-fetch:



multiple-row-fetch:



Notes:

- 1 For single-row-fetch, a host-variable-array can be specified instead of a host variable and the descriptor can describe host-variable-arrays. In either case, data is returned only for the first entry of the host-variable-array.
- 2 This clause is optional. If this clause is not specified and either a rowset size has not been established yet or a row positioned FETCH statement was the last type of FETCH statement issued for this cursor, the rowset size is implicitly one. If the last FETCH statement issued for this cursor was a rowset positioned FETCH statement and this clause is not specified, the rowset size is the same size as the previous rowset positioned FETCH.

Description

INSENSITIVE

Returns the row from the result table as it is. If the row has been previously fetched with a FETCH SENSITIVE, it reflects changes made outside this cursor before the FETCH INSENSITIVE statement was issued. Positioned updates and deletes are reflected with FETCH INSENSITIVE if the same cursor was used for the positioned update or delete.

INSENSITIVE can only be specified for cursors declared as INSENSITIVE or SENSITIVE STATIC (or if the cursor is declared as ASENSITIVE and DB2 defaults to INSENSITIVE). Otherwise, if the cursor is declared as SENSITIVE DYNAMIC (or if the cursor is declared as ASENSITIVE and DB2 defaults to SENSITIVE DYNAMIC), an error occurs and the FETCH statement has no effect. For an INSENSITIVE cursor, specifying INSENSITIVE is optional because it is the default.

SENSITIVE

Updates the fetched row in the result table from the corresponding row in the base table of the cursor's SELECT statement and returns the current values. Thus, it reflects changes made outside this cursor. SENSITIVE can only be specified for a sensitive cursor. Otherwise, if the cursor is insensitive, an error occurs and the FETCH statement has no effect. For a SENSITIVE cursor, specifying SENSITIVE is optional because it is the default.

When the cursor is declared as SENSITIVE STATIC and a FETCH SENSITIVE is requested, the following steps are taken:

1. DB2 retrieves the row of the database that corresponds to the row of the result table that is about to be fetched.
2. If the corresponding row has been deleted, a "delete hole" occurs in the result table, a warning is issued, the cursor is repositioned on the "hole", and no data is fetched. (DB2 marks a row in the result table as a "delete hole" when the corresponding row in the database is deleted.)
3. If the corresponding row has not been deleted, the predicate of the underlying SELECT statement is re-evaluated. If the row no longer satisfies the predicate, an "update hole" occurs in the result table, a warning is issued, the cursor is repositioned on the "hole," and no data is fetched. (DB2 marks a row in the result table as an "update hole" when an update to the corresponding row in the database causes the row to no longer qualify for the result table.)
4. If the corresponding row does not result in a delete or an update hole in the result table, the cursor is repositioned on the row of the result table and the data is fetched.

WITH CONTINUE

Specifies that the DB2 subsystem should prepare to allow subsequent FETCH CURRENT CONTINUE operations to access any truncated LOB or XML result column following an initial FETCH operation that provides output variables that are not large enough to hold the entire LOB or XML columns. When the WITH CONTINUE clause is specified, the DB2 subsystem takes the following actions that can differ from the case where the FETCH statement does not include the WITH CONTINUE clause:

- If truncation occurs when returning an XML or LOB column, the DB2 subsystem will remember the truncation position and will not discard the remaining data.
- If truncation occurs when returning an XML or LOB column, the DB2 subsystem returns the total length that would have been required to hold all of the data of the LOB or XML column. This will either be in the first four bytes of the LOB host variable structure or in the 4 byte area that is pointed to by the SQLDATALEN pointer in the SQLVAR entry of the SQLDA for that host variable. What is returned depends on the programming method that is used. See "SQL descriptor area (SQLDA)" on page 1656 for details about the SQLDA contents.
- If returning XML data, the result column will be fully materialized in the database before the data is returned.

If the CURRENT CONTINUE clause is specified, the WITH CONTINUE behavior is assumed.

AFTER

Positions the cursor after the last row of the result table. Values are not assigned to host variables. The number of rows of the result table are returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA for cursors with an effective sensitivity of INSENSITIVE or SENSITIVE STATIC.

BEFORE

Positions the cursor before the first row of the result table. Values are not assigned to host variables.

row-positioned

Positioning of the cursor with row-positioned fetch orientations NEXT, PRIOR, CURRENT and RELATIVE is done in relation to the current cursor position.

Following a successful row-positioned `FETCH` statement, the cursor is positioned on a single row of data. If the cursor is enabled for rowsets, positioning is performed relative to the current row or the first row of the current rowset, and the cursor is positioned on a rowset consisting of a single row.

NEXT

Positions the cursor on the next row or rows of the result table relative to the current cursor position, and returns data if a target is specified. `NEXT` is the only row-positioned fetch operation that can be explicitly specified for cursors that are defined as `NO SCROLL`. `NEXT` is the default if no other cursor positioning is specified. If a specified row reflects a hole, a warning is issued and data values are not assigned to host variables for that row.

Table 118 lists situations for different cursor positions and the results when `NEXT` is used.

Table 118. Results when NEXT is used with different cursor positions

Current state of the cursor	Result of <code>FETCH NEXT</code>
Before the first row	Cursor is positioned on the first row (1) and data is returned if requested.
On the last row or after the last row	A warning occurs, values are not assigned to host variables, and the cursor position is unchanged.
Before a hole	For a <code>SENSITIVE STATIC</code> cursor, a warning occurs for a delete hole or an update hole, values are not assigned to host variables, and the cursor is positioned on the hole.
Unknown	An error occurs, values are not assigned to host variables, and the cursor position remains unknown.

Note:

(1) This row is not applicable in the case of a forward-only cursor (that is when `NO SCROLL` was specified implicitly or explicitly).

PRIOR

Positions the cursor on the previous row or rows of the result table relative to the current cursor position, and returns data if a target is specified. If a specified row reflects a hole, a warning is issued, and data values are not assigned to host variables for that row.

Table 119 lists situations for different cursor positions and the results when `PRIOR` is used.

Table 119. Results when PRIOR is used with different cursor positions

Current state of the cursor	Result of <code>FETCH PRIOR</code>
Before the first row or on the first row	A warning occurs, values are not assigned to host variables, and the cursor position is unchanged.
After a hole	For a <code>SENSITIVE STATIC</code> cursor, a warning occurs for a delete hole or an update hole, values are not assigned to host variables, and the cursor is positioned on the hole.
After the last row	Cursor is positioned on the last row.

Table 119. Results when PRIOR is used with different cursor positions (continued)

Current state of the cursor	Result of FETCH PRIOR
Unknown	An error occurs, values are not assigned to host variables, and the cursor position remains unknown.

FIRST

Positions the cursor on the first row of the result table, and returns data if a target is specified. For a SENSITIVE STATIC cursor, if the first row of the result table is a hole, a warning occurs for a delete hole or an update hole and values are not assigned to host variables.

LAST

Positions the cursor on the last row of the result table, and returns data if a target is specified. The number of rows of the result table is returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA for an insensitive or sensitive static cursor. For a SENSITIVE STATIC cursor, if the last row of the result table is a hole, a warning occurs for a delete hole or an update hole and values are not assigned to host variables.

CURRENT

The cursor position is not changed, data is returned if a target is specified. If the cursor was positioned on a rowset of more than one row, the cursor position is on the first row of the rowset.

Table 120 lists situations in which errors occur with the CURRENT clause.

Table 120. Situations in which errors occur with CURRENT

Current state of the cursor	Result of FETCH CURRENT
Before the first row or after the last row	A warning occurs, values are not assigned to host variables, and the cursor position is unchanged.
On a hole	For a SENSITIVE STATIC, a warning occurs for a delete hole or an update hole, values are not assigned to host variables, and the cursor is positioned on the hole. If the cursor is defined as a rowset cursor, with isolation level UR or a sensitive dynamic scrollable cursor, it is possible that a different row will be returned than the FETCH that established the most recent cursor position. This can occur while fetching a row again when it is determined to not be there anymore. In this case, fetching continues moving forward to get the row of data.
Unknown	An error occurs, values are not assigned to host variables, and the cursor position remains unknown.

CONTINUE

The cursor positioning is not changed, and data is returned if a target is specified. The FETCH CURRENT CONTINUE statement retrieves remaining data for any LOB or XML column result values that were truncated on a previous FETCH or FETCH CURRENT CONTINUE statement. It assigns the remaining data for those truncated columns to

the host variables that are referenced in the statement or pointed to by the descriptor. The data that is returned for previously-truncated result values begins at the point of truncation. This form of the CURRENT clause must only be used after a single-row FETCH WITH CONTINUE or FETCH CURRENT CONTINUE statement that has returned partial data for one or more LOB or XML columns. The cursor must be open and positioned on a row.

FETCH CURRENT CONTINUE must pass host variables entries for all columns in the SELECT list, even though the non-LOB columns or non-XML columns will not return any data.

ABSOLUTE

host-variable or *integer-constant* is assigned to an integral value *k*. If a *host-variable* is specified, it must be an exact numeric type with zero scale and must not include an indicator variable. The possible data types for the host variable are DECIMAL(*n*,0) or integer. The DECIMAL data type is limited to DECIMAL(18,0). An *integer-constant* can be up to 31 digits, depending on the application language.

If *k*=0, the cursor is positioned before the first row of the result table. Otherwise, ABSOLUTE positions the cursor to row *k* of the result table if *k*>0, or to *k* rows from the bottom of the table if *k*<0. For example, "ABSOLUTE -1" is the same as "LAST".

Data is returned if the specified position is within the rows of the result table, and a target is specified.

If an absolute position is specified that is before the first row or after the last row of the result table, a warning occurs, values are not assigned to host variables, and the cursor is positioned either before the first row or after the last row. If the resulting cursor position is after the last row for INSENSITIVE and SENSITIVE STATIC scrollable cursors, the number of rows of the result table are returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA. If row *k* of the result table is a hole, a warning occurs and values are not assigned to host variables.

FETCH ABSOLUTE 0 results in positioning before the first row and a warning is issued. FETCH BEFORE results in positioning before the first row and no warning is issued.

Table 121 lists some synonymous specifications.

Table 121. Synonymous scroll specifications for ABSOLUTE

Specification	Alternative
ABSOLUTE 0 (but with a warning)	BEFORE (without a warning)
ABSOLUTE +1	FIRST
ABSOLUTE -1	LAST
ABSOLUTE - <i>m</i> , 0< <i>m</i> ≤ <i>n</i>	ABSOLUTE <i>n</i> +1- <i>m</i>
ABSOLUTE <i>n</i>	LAST
ABSOLUTE - <i>n</i>	FIRST
ABSOLUTE <i>x</i> (with a warning)	AFTER (without a warning)
ABSOLUTE - <i>x</i> (with a warning)	BEFORE (without a warning)

Note: Assume: 0≤*m*≤*n*<*x* Where, *n* is the number of rows in the result table.

RELATIVE

host-variable or *integer-constant* is assigned to an integral value k . If a *host-variable* is specified, it must be an exact numeric type with zero scale and must not include an indicator variable. The possible data types for the host variable are DECIMAL($n,0$) or integer. The DECIMAL data type is limited to DECIMAL(18,0).

If the cursor is positioned before the first row, or after the last row of the result table, the cursor position is determined as follows:

- If n is 0, the cursor position is unchanged, values are not assigned to host variables, and a warning occurs
- If n is positive, and the cursor is positioned before the first row, the cursor is positioned on a rowset starting at row n
- If n is positive, and the cursor is positioned after the last row, a warning occurs
- If n is negative, and the cursor is positioned before the first row, a warning occurs
- If n is negative, and the cursor is positioned after the last row, the cursor is positioned on a rowset starting as row n from the end of the result table

An *integer-constant* can be up to 31 digits, depending on the application language.

Data is returned if the specified position is within the rows of the result table, and a target is specified.

RELATIVE positions the cursor to the row in the result table that is either k rows after the current row if $k > 0$, or $ABS(k)$ rows before the current row if $k < 0$. For example, "RELATIVE -1" is the same as "PRIOR". If $k = 0$, the position of the cursor does not change (that is, "RELATIVE 0" is the same as "CURRENT").

If a relative position is specified that results in positioning before the first row or after the last row, a warning is issued, values are not assigned to host variables, and the cursor is positioned either before the first row or after the last row. If the resulting cursor position is after the last row for INSENSITIVE and SENSITIVE STATIC scrollable cursors, the number of rows of the result table is returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA. If the cursor is positioned on a hole and RELATIVE 0 is specified or if the target row is a hole, a warning occurs and values are not assigned to host variables.

If the cursor is defined as a rowset cursor, with isolation level UR or a sensitive dynamic scrollable cursor, it is possible that a different row will be returned than the FETCH that established the most recent cursor position. This can occur while fetching a row again when it is determined to not be there anymore. In this case, fetching continues moving forward to get the row data.

If the cursor position is unknown and RELATIVE 0 is specified, an error occurs.

Table 122 lists some synonymous specifications.

Table 122. Synonymous Scroll Specifications for RELATIVE

Specification	Alternative
RELATIVE +1	NEXT

Table 122. Synonymous Scroll Specifications for *RELATIVE* (continued)

Specification	Alternative
RELATIVE -1	PRIOR
RELATIVE 0	CURRENT
RELATIVE +r (with a warning)	AFTER (without a warning)
RELATIVE -r (with a warning)	BEFORE (without a warning)
Note:	
<i>r</i> has to be large enough to position the cursor beyond either end of the result table.	

rowset-positioned

Positioning of the cursor with rowset-positioned fetch orientations NEXT ROWSET, PRIOR ROWSET, CURRENT ROWSET, and ROWSET STARTING AT RELATIVE is done in relation to the current cursor position. Following a successful row-positioned FETCH statement, the cursor is positioned on a rowset of data. The number of rows in the rowset is determined either explicitly or implicitly. The FOR *n* ROWS clause in the multiple-row-fetch clause is used to explicitly specify the size of the rowset. Positioning is performed relative to the current row or first row of the current rowset, and the cursor is positioned on all rows of the rowset.

A rowset-positioned fetch orientation must not be specified if the current cursor position is not defined to access rowsets. NEXT ROWSET is the only rowset-positioned fetch orientation that can be specified for cursors that are defined as NO SCROLL.

If a row of the rowset reflects a hole, a warning is returned, data values are not assigned to host variable arrays for that row (that is, the corresponding positions in the target host variable arrays are untouched), and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, an error occurs.

NEXT ROWSET

Positions the cursor on the next rowset of the result table relative to the current cursor position, and returns data if a target is specified. The next rowset is logically obtained by fetching the row that follows the current rowset and fetching additional rows until the number of rows that is specified implicitly or explicitly in the FOR *n* ROWS clause is obtained or the last row of the result table is reached.

If the cursor is positioned before the first row of the result table, the cursor is positioned on the first rowset.

If the cursor is positioned on the last row or after the last row of the result table, the cursor position is unchanged, values are not assigned to host variable arrays, and a warning occurs.

If a row of the rowset reflects a hole, the following actions occur:

- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the cursor is not positioned because of a prior error, values are not assigned to the host-variable-array, and an error is returned. If a row of the rowset would be after the last row of the result table, values are not assigned to host-variable-arrays for that row and any subsequent requested rows of the rowset, and a warning is returned.

NEXT ROWSET is the only rowset positioned fetch orientation that can be explicitly be specified for cursors that are defined as NO SCROLL.

PRIOR ROWSET

Positions the cursor on the previous rowset of the result table relative to the current position, and returns data if a target is specified.

The prior rowset is logically obtained by fetching the row that precedes the current rowset and fetching additional rows until the number of rows that is specified implicitly or explicitly in the FOR n ROWS clause is obtained or the last row of the result table is reached.

If the cursor is positioned after the last row of the result table, the cursor is positioned on the last rowset.

If the cursor is positioned before the first row or on the first row of the result table, the cursor position is unchanged, values are not assigned to host variable arrays, and a warning occurs.

If a row would be before the first row of the result table, the cursor is positioned on a partial rowset that consists of only those rows that are prior to the current position of the cursor starting with the first row of the result table, and a warning is returned. Values are not assigned to the host-variable-arrays for the rows in the rowset for which the warning is returned.

Although the rowset is logically obtained by fetching backwards from before the current rowset, the data is returned to the application starting with the first row of the rowset, to the end of the rowset.

If a row of the rowset reflects a hole, the following actions occur:

- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the cursor is not positioned because of a prior error, values are not assigned to the host-variable-array, and an error is returned.

FIRST ROWSET

Positions the cursor on the first rowset of the result table, and returns data if a target is specified.

If a row of the rowset reflects a hole, the following actions occur:

- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the result table contains fewer rows than specified implicitly or explicitly in the FOR *n* ROWS clause, values are not assigned to host-variable-arrays after the last row of the result table, and a warning is returned.

LAST ROWSET

Positions the cursor on the last rowset of the result table and returns data if a target is specified. The last rowset is logically obtained by fetching the last row of the result table and fetching prior rows until the number of rows in the rowset is obtained or the first row of the result table is reached. Although the rowset is logically obtained by fetching backwards from the bottom of the result table, the data is returned to the application starting with the first row of the rowset, to the end of the rowset, which is also the end of the result table.

If a row of the rowset reflects a hole, the following actions occur:

- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned.

If the result table contains fewer rows than specified implicitly or explicitly in the FOR *n* ROWS clause, the last rowset is the same as the first rowset, values are not assigned to host-variable-arrays after the last row of the result table, and a warning is returned.

CURRENT ROWSET

If the FOR *n* ROWS clause specifies a number different from the number of rows specified implicitly or explicitly in the FOR *n* ROWS clause on the most recent FETCH statement for this cursor, the cursor is repositioned on the specified number of rows, starting with the first row of the current rowset. If the cursor is positioned before the first row, or after the last row of the result table, the cursor position is unchanged, values are not assigned to host variable arrays, and a warning occurs. If the FOR *n* ROWS clause is not specified, it is possible that the FETCH statement will position the cursor on a partial rowset when the FETCH CURRENT ROWSET statement is processed. In this case, DB2 attempts to position the cursor on a full rowset starting with the first row of the current rowset. Otherwise, the position of the cursor on the current rowset is unchanged. Data is returned if a target is specified.

With isolation level UR or a sensitive dynamic scrollable cursor, it is possible that different rows will be returned than the FETCH that established the most recent rowset cursor position. This can occur while refetching the first row of the rowset when it is determined to not be there anymore. In this case, fetching continues moving forward to get the first row of data for the rowset. This can also occur when changes have been made to other rows in the current rowset such that they no longer exist or have been logically moved within (or out of) the result table of the cursor.

If the cursor is not positioned because of a prior error, values are not assigned to the host-variable-array, and an error occurs.

If the current rowset contains fewer rows than specified implicitly or explicitly in the FOR *n* ROWS clause, values are not assigned to host-variable-arrays after the last row, and a warning is returned.

ROWSET STARTING AT ABSOLUTE or RELATIVE *host-variable or integer-constant*

Positions the cursor on the rowset beginning at the row of the result table that is indicated by the ABSOLUTE or RELATIVE specification, and returns data if a target is specified.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, and must not include an indicator variable. The possible data types for the host variable are DECIMAL(*n*,0) or integer, where the DECIMAL data type is limited to DECIMAL(18,0). If a constant is specified, the value must be an integer.

If a row of the result table would be after the last row or before the first row of the result table, values are not assigned to host-variable-arrays for that row and a warning is returned.

ABSOLUTE

If *k*=0, an error occurs. If *k*>0, the first row of the rowset is row *k*. If *k*<0, the rowset is positioned on the ABS(*k*) rows from the bottom of the result table. Assume that ABS(*k*) is equal to the number of rows for the rowset and that there are enough row to return a complete rowset:

- FETCH ROWSET STARTING AT ABSOLUTE -*k* is the same as FETCH LAST ROWSET.
- FETCH ROWSET STARTING AT ABSOLUTE 1 is the same as FETCH FIRST ROWSET.

RELATIVE

If *k*=0 and the FOR *n* ROWS clause does not specify a number different from the number most recently specified implicitly or explicitly for this cursor, then the position of the cursor does not change (that is, "RELATIVE ROWSET 0" is the same as "CURRENT ROWSET"). If *k*≠0 and the FOR *n* ROWS clause specifies a number different from the number most recently specified implicitly or explicitly for this cursor, then the cursor is repositioned on the specified number of rows, starting with the first row of the current rowset.

If the cursor is positioned before the first row, or after the last row of the result table, the cursor position is determined as follows:

- If *n* is 0, the cursor position is unchanged, values are not assigned to host variables, and a warning occurs. This is the same as FETCH CURRENT ROWSET.
- If *n* is positive, and the cursor is positioned before the first row, the cursor is positioned on a rowset starting a row *n*.
- If *n* is positive, and the cursor is positioned after the last row, a warning occurs.
- If *n* is negative, and the cursor is positioned before the first row, a warning occurs.
- If *n* is negative, and the cursor is positioned after the last row, the cursor is positioned on a rowset starting at row *n* from the bottom of the result table.

Otherwise, RELATIVE repositions the cursor so that the first row of the new rowset cursor position is on the row in the result table that is either k rows after the first row of the current rowset cursor position if k>0, or ABS(k) rows before the first row of the current rowset cursor position if k<0. Assume that ABS(k) is equal to the number of rows for the resulting rowset

- FETCH ROWSET STARTING AT RELATIVE -k is the same as FETCH PRIOR ROWSET.
- FETCH ROWSET STARTING AT RELATIVE k is the same as FETCH NEXT ROWSET.
- FETCH ROWSET STARTING AT RELATIVE 0 is the same as FETCH CURRENT ROWSET.

When ROWSET STARTING AT RELATIVE -n is specified and there are not enough rows between the current position of the cursor and the beginning of the result table to return a complete rowset:

- A warning is returned.
- Values are not assigned to the host-variable-arrays.
- The cursor is positioned before the first row.

If a row of the rowset reflects a hole, If a row of the rowset reflects a hole, the following actions occur:

- A warning is returned.
- Data values are not assigned to the host-variable-arrays for that row (that is, the corresponding positions in the target host-variable-arrays are untouched).
- A value of -3 is returned in all of the indicator variables that are provided for the row.

If a hole is detected and at least one indicator variable is not provided, an error is returned. If a row of the rowset is unknown, values are not assigned to host variable arrays for that row, and an error is returned. If a row of the rowset would be after the last row or before the first row of the result table, values are not assigned to host-variable-arrays for that row, and a warning is returned.

cursor-name

Identifies the cursor to be used in the fetch operation. The cursor name must identify a declared cursor, as explained in the description of the DECLARE CURSOR statement in “DECLARE CURSOR” on page 1191, or an allocated cursor, as explained in “ALLOCATE CURSOR” on page 696. When the FETCH statement is executed, the cursor must be in the open state.

If a single-row-fetch or multiple-row-fetch clause is not specified, the cursor position is adjusted as specified, but no data is returned to the user.

single-row-fetch

When *single-row-fetch* is specified, SENSITIVE or INSENSITIVE can be specified though there is a default. The default depends on the sensitivity of the cursor. If the sensitivity of the cursor is INSENSITIVE, then the default is INSENSITIVE. If the effective sensitivity of the cursor is SENSITIVE DYNAMIC or SENSITIVE STATIC, then the default is SENSITIVE. The single-row-fetch or multiple-row-fetch clause must not be specified when the FETCH BEFORE or FETCH AFTER option is specified. They are required when FETCH BEFORE or FETCH AFTER is not specified. If an individual fetch operation causes the cursor to be positioned or to remain positioned on a row if there is a target specification, the values of the result table are assigned to host variables as specified by the single-fetch-clause.

INTO *host-variable*,...

Specifies a list of host variables. Each *host-variable* must identify a structure or variable that is described in the application program in accordance with the rules for declaring host structures and variables. A reference to a structure is replaced by a reference to each of its variables. The first value in the result row is assigned to the first host variable, the second value to the second host variable, and so on.

INTO DESCRIPTOR *descriptor-name*

Identifies an SQLDA that contains a valid description of the host output variables. Result values from the associated SELECT statement are returned to the application program in the output host variables.

Before the FETCH statement is processed, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
A REXX SQLDA does not contain this field.
- SQLABC to indicate the number of bytes of storage allocated in the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR occurrences. Each SQLVAR occurrence describes a host variable or buffer into which a value in the result table is to be assigned. If LOBs are present in the results, there must be additional SQLVAR entries for each column of the result table. If the result table contains only base types and distinct types, multiple SQLVAR entries are not needed for each column. However, extra SQLVAR entries are needed for distinct types as well as for LOBs in DESCRIBE and PREPARE INTO statements. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see “SQL descriptor area (SQLDA)” on page 1656.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C.

multiple-row-fetch

Retrieves multiple rows of data from the result table of a query. The FOR *n* ROWS clause of the FETCH statement controls how many rows are returned on a single FETCH statement. The fetch orientation determines whether the resulting cursor position (for example, on a single row, rowset, before, or after the result table). Fetching stops when an error is returned, all requested rows are fetched, or the end of data condition is reached.

Fetching multiple rows of data can be done with scrollable or non-scrollable cursors. The operations used to define, open, and close a cursor used for fetching multiple rows of data are the same as for those used for single row FETCH statements.

If the BEFORE or AFTER option is specified, neither single-row-fetch or multiple-row-fetch can be specified.

FOR *host-variable* **or** *integer-constant* **ROWS**

host-variable or *integer-constant* is assigned to an integral value *k*. If a host variable is specified, it must be an exact numeric type with a scale of zero and must not include an indicator variable. Furthermore, *k* must be in the range, $0 < k \leq 32767$.

This clause must not be specified if a row-positioned fetch-orientation clause was specified. This clause must also not be specified for a cursor that is defined without rowset access.

If a rowset fetch orientation is specified and this clause is not specified, the number of rows in the resulting rowset is determined as follows:

- If the most recent FETCH statement for this cursor was a rowset-positioned FETCH, the number of rows of the rowset is implicitly determined by the number of rows that was most recently specified (implicitly or explicitly) for this cursor.
- When the most recent FETCH statement for this cursor was either FETCH BEFORE or FETCH AFTER and the most recent FETCH statement for this cursor prior to that was a rowset-positioned FETCH, the number of rows of the rowset is implicitly determined by the number of rows that were most recently specified (implicitly or explicitly) for this cursor.
- Otherwise, the rowset consists of a single row.

For result set cursors, the number of rows for a rowset cursor position, established in the procedure that defined the rowset, is not inherited by the caller when the rowset is returned. Use the FOR *n* ROWS clause on the first rowset FETCH statement for the result set in the calling program to establish the number of rows for the cursor. Otherwise, the rowset consists of a single row.

The cursor is positioned on the row or rowset that is specified by the orientation clause (for example, NEXT ROWSET), and those rows are fetched if a target is specified. After the cursor is positioned on the first row being fetched, the next *k*-1 rows are fetched. Fetching moves forward from the cursor position in the result table and continues until the end of data condition is returned, *k*-1 rows have been fetched, or an assignment error is returned.

The resulting cursor position depends on the fetch orientation that is specified:

- For a row-positioned fetch orientation, the cursor is positioned at the last row successfully retrieved.
- For a rowset-positioned fetch orientation, the cursor is positioned on all the rows retrieved.

The values from each individual fetch are placed in data areas that are described in the INTO or USING clause. If a target specification is provided for a rowset-positioned FETCH, the host variable arrays must be specified as the target specification, and the arrays must be defined with a dimension of 1 or greater. The target specification must be defined as an array for a rowset-positioned FETCH even if the number of rows that is specified implicitly or explicitly is one. See Diagnostics information for rowset positioned FETCH statements.

INTO *host-variable-array*

Identifies for each column of the result table a host-variable-array to receive the data that is retrieved with this FETCH statement. If the number

of host-variable-arrays is less than the number of columns of the result table, the SQLWARN3 field of the SQLCA is set to 'W'. No warning is given if there are more host-variable-arrays than the number of columns in the result table.

Each host-variable-array must be defined in the application program in accordance with the rules for declaring an array. A host-variable-array is used to return the values for a column of the result table. The number of rows to be fetched must be less than or equal to the dimension of each of the host-variable-arrays.

An optional indicator array can be specified for a host-variable-array. It should be specified if the SQLTYPE of any SQLVAR occurrence indicates that the column of the result table is nullable. Additionally, if an operation may result in null values, such as an UPDATE operation that results in a hole, is performed in the application, an indicator array should be specified. Otherwise an error occurs if null values are encountered. The indicators are returned as small integers.

INTO DESCRIPTOR *descriptor-name*

Identifies an SQLDA that must contain a valid description of zero or more host-variable-arrays or buffers into which the values for a column of the result table are to be returned.

Before the FETCH statement is processed, you must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of an element of the host-variable-array. Within each SQLVAR representing an array:
 - SQLTYPE indicates the data type of the elements of the host-variable-array.
 - SQLDATA field points to the first element of the host-variable-array.
 - The length fields (SQLLEN and SQLLONGLEN) are set to indicate the maximum length of a single element of the array.
 - SQLNAME - The length of SQLNAME must be set to 8, and the first two bytes of the data portion of SQLNAME must be initialized to X'0000'. The fifth and sixth bytes must contain a flag field and the seventh and eighth bytes must be initialized to a binary small integer (half word) representation of the dimension of the host-variable-array, and the corresponding indicator array, if one is specified.

The SQLVAR entry for the number of rows must also contain a flag value. The number of rows to be fetched must be less than or equal to the dimension of each of the host variable arrays.

You set the SQLDATA and SQLIND pointers to the beginning of the corresponding arrays. The SQLDA must have enough storage to contain all SQLVAR occurrences. Each SQLVAR occurrence describes a host-variable-array or buffer into which the values for a column in the result table are to be returned. If any column of the result table is a LOB, two SQLVAR entries must be provided for each SQLVAR, and SQLN must

be set to two times the number of SQLVARs. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

Notes

Assignment to host variables: The *n*th variable identified by the INTO clause or described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of a host variable must be compatible with its corresponding value. If the value is numeric, the variable must have the capacity to represent the whole part of the value. For a datetime value, the variable must be a character string variable of a minimum length as defined in “String representations of datetime values” on page 89. If the value is null, an indicator variable must be specified.

Assignments are made in sequence through the list. Each assignment to a variable is made according to the rules described in Chapter 2, “Language elements,” on page 47. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to W. There is no warning if there are more variables than the number of result columns. If the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the variable and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to host variables with other data types. For more information on using locators, see *DB2 Application Programming and SQL Guide*.

Restrictions on using the WITH CONTINUE and CURRENT CONTINUE clauses: When using the WITH CONTINUE clause, the DB2 system will only reserve truncated data for result set columns of the BLOB, CLOB, DBCLOB, or XML data type, and only when the output host variable data type is the appropriate LOB data type.

If an application uses FETCH WITH CONTINUE, and truncated data remains after the FETCH operation, the application cannot perform any intervening operation on that cursor before performing the FETCH CURRENT CONTINUE. If intervening operations on that cursor are performed, the truncated data is lost.

FETCH CURRENT CONTINUE is not supported with multi-row fetch. Also, FETCH CURRENT CONTINUE is not supported for non-LOB and non-XML columns that have been truncated. If truncation occurs for these non-LOB and non-XML columns, the truncated data will be discarded as usual.

Result column evaluation considerations: If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (division by zero, or overflow) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.) If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered. No value is assigned to the host variable or to later variables, though any values that have already been assigned to variables remain assigned.

If the specified host variable is not large enough to contain the result, a warning is returned and W is assigned to SQLWARN1 in the SQLCA. The actual length of the result is returned in the indicator variable associated with the host-variable, if an indicator is provided. It is possible that a warning may not be returned on a FETCH operation. This occurs as a result of optimizations, such as the use of system temporary tables or blocking. It is also possible that the returned warning applies to a previously fetched row. When a datetime value is returned, the length of the variable must be large enough to store the complete value. Otherwise, a warning or an error is returned.

Cursor positioning: An open cursor has three possible positions:

- Before a row
- On a row or rowset
- After the last row

When a scrollable or non-scrollable cursor is opened, it is positioned before the first row in the result table. If a cursor is on a row, that row is called the current row of the cursor. If a cursor is on a rowset, the rows are called the current rowset of the cursor.

A cursor referred to in an UPDATE or DELETE statement must be positioned on a row or rowset. A cursor can only be on a row or rowset as a result of a FETCH statement.

If the cursor was declared SENSITIVE STATIC SCROLL, a row may be a *hole*, from which no values may be fetched, updated, or deleted. Holes do not exist with sensitive dynamic cursors because there is no temporary result table. For information about holes in the result table of a cursor, see *DB2 Application Programming and SQL Guide*.

For scrollable cursors, the cursor position after an error varies depending on the type of error:

- When an operation is attempted against an update or delete hole, or when an update or delete hole is detected, the cursor is positioned on the hole.
- When a FETCH operation is attempted past the end of file, the cursor is positioned after the last row.
- When a FETCH operation is attempted before the beginning of file, the cursor is positioned before the first row.
- When an error causes the cursor position to be invalid such as when a single row positioned update or positioned delete error occurs that causes a rollback, the cursor is closed.

Cursor position after exception condition: If an error occurs during the execution of a fetch operation, the position of the cursor and the result of any later fetch is unpredictable. It is possible for an error to occur that makes the position of the cursor invalid, in which case the cursor is closed.

If an individual fetch operation specifies a destination that is outside the range of the cursor, a warning is issued (except for FETCH BEFORE or FETCH AFTER), the cursor is positioned before or after the result table, and values are not assigned to host variables.

Concurrency and scrollability: The current row of a cursor cannot be updated or deleted by another application process if it is locked. Unless it is already locked

because it was inserted or updated by the application process during the current unit of work, the current row of a cursor is not locked if:

- The isolation level is UR, or
- The isolation level is CS, and
 - The result table of the cursor is read-only
 - The bind option CURRENTDATA(NO) is in effect

A dynamic scrollable cursor is useful when it is more important to the application to see updated rows and newly inserted rows and there is no need to see deleted rows. The isolation level of CS should be used for maximum concurrency with dynamic scrollable cursors. Specifying an isolation level of RR or RS severely restricts the update of the table, thus defeating the purpose of a SENSITIVE DYNAMIC scrollable cursor. If the application needs a constant result table, a SENSITIVE STATIC scrollable cursor with an isolation level of CS should be used.

Sensitivity of SENSITIVE STATIC SCROLL cursors to database changes: When SENSITIVE STATIC SCROLL has been declared, the following rules apply:

- For the result of an update operation to be visible within a cursor after "open," the update operation must be a positioned update executed against the cursor, or a FETCH SENSITIVE in a STATIC cursor must be executed against a row which has been updated by some other means (that is, a searched update, committed updates of others, or an update with another cursor in the same process).
- Another process can update the base table of the SELECT statement so that the current values no longer satisfy the WHERE clause. In this case, an "update hole" effectively exists during the time the values in the base table do not satisfy the WHERE clause, and the row is no longer accessible through the cursor. When an attempt is made to fetch a row that has been identified as an update hole, no values are returned, and a warning is issued.

Under SENSITIVE STATIC SCROLL cursors, update holes are only identified during positioned update, positioned delete, and FETCH SENSITIVE operations. Each positioned update, positioned delete, and FETCH SENSITIVE operation does the necessary tests to determine if an update hole exists.

- For the result of a delete operation to be visible within a SENSITIVE STATIC SCROLL cursor, the delete operation must be a positioned delete executed against the cursor or a FETCH SENSITIVE in a STATIC cursor must be executed against a row that has been deleted by some other means (that is, a searched delete, committed deletes of others, or a delete with another cursor in the same process).
- Another process, or the even the same process, may delete a row in the base table of the SELECT statement so that a row of the cursor no longer has a corresponding row in the base table. In this case, a "delete hole" effectively exists, and that row is no longer accessible through the cursor. When an attempt is made to fetch a row that has been identified as a delete hole, no values are returned, and a warning is issued.

Under SENSITIVE STATIC SCROLL cursors, delete holes are identified during positioned update, positioned delete, and FETCH SENSITIVE operations.

- Inserts into the base table or tables of SENSITIVE STATIC SCROLL cursors are not seen after the cursor is opened.

LOB locators: When information is retrieved into LOB locators and it is not necessary to retain the locator across FETCH statements, it is a good practice to issue a FREE LOCATOR statement before issuing another FETCH statement because locator resources are limited.

Isolation level considerations: The isolation level of the statement (specified implicitly or explicitly) can affect the result of a rowset-positioned FETCH statement. This is possible when changes are made to the tables underlying the cursor when isolation level UR is used with a dynamic scrollable cursor, or with other isolation levels when rows have been added by the application fetching from the cursor. These situations can occur with the following fetch orientations:

PRIOR ROWSET

With a dynamic scrollable cursor and isolation level UR, the content of a prior rowset can be affected by other activity within the table. It is possible that a row that previously qualified for the cursor, and was included as a member of the "prior" rowset, has since been deleted or modified before it is actually returned as part of the rowset for the current statement. To avoid this behavior, use an isolation level other than UR.

CURRENT ROWSET

With a dynamic scrollable cursor, additional rows can be added between rows that form the rowset that was returned to the user. With isolation level RR, these rows can only be added by the application fetching from the cursor. For isolation levels other than RR, other applications can insert rows that can affect the results of a subsequent FETCH CURRENT ROWSET. To avoid this behavior, use a static scrollable cursor instead of a dynamic scrollable cursor.

LAST ROWSET

With a dynamic scrollable cursor and isolation level UR, the content of the last rowset can be affected by other activity within the table. It is possible that a row that previously qualified for the cursor, and was included as a member of the "last" rowset, has since been deleted or modified before it is actually returned as part of the rowset for the current statement. To avoid this behavior, use an isolation level other than UR.

ROWSET STARTING AT RELATIVE *-n* (where *-n* is a negative number)

With a dynamic scrollable cursor and isolation level UR, the content of a prior rowset can be affected by other activity within the table. It is possible that a row that previously qualified for the cursor, and was included as a member of the "prior" rowset, has since been deleted or modified before it is actually returned as part of the rowset for the current statement. To avoid this behavior, use an isolation level other than UR.

Row positioned and rowset positioned FETCH statement interaction: Table 123 on page 1310 demonstrates the interaction between row positioned and rowset positioned FETCH statements. The table is based on the following assumptions:

- TABLE T1 has 15 rows
- CURSOR CS1 is declared as follows:

```
DECLARE CS1 SCROLL CURSOR WITH ROWSET POSITIONING FOR
SELECT * FROM T1;
```
- An OPEN CURSOR statement has been successfully executed for CURSOR CS1 and the FETCH statements in the table are executed in the order that they appear in the table.

Table 123. Interaction between row positioned and rowset positioned FETCH statements

FETCH Statement	Cursor Position
FETCH FIRST	Cursor is positioned on row 1.
FETCH FIRST ROWSET	Cursor is positioned on a rowset of size 1, consisting of row 1.
FETCH FIRST ROWSET FOR 5 ROWS	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH CURRENT ROWSET	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH CURRENT	Cursor is positioned on row 1
FETCH FIRST ROWSET FOR 5 ROWS	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH or FETCH NEXT	Cursor is positioned on row 2.
FETCH NEXT ROWSET	Cursor is positioned on a rowset of size 1, consisting of row 3.
FETCH NEXT ROWSET FOR 3 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 4,5, and 6.
FETCH NEXT ROWSET	Cursor is positioned on a rowset of size 3, consisting of rows 7,8, and 9.
FETCH LAST	Cursor is positioned on row 15.
FETCH LAST ROWSET FOR 2 ROWS	Cursor is positioned on a rowset of size 2, consisting of rows 14 and 15.
FETCH PRIOR ROWSET	Cursor is positioned on a rowset of size 2, consisting of rows 12 and 13.
FETCH ABSOLUTE 2	Cursor is positioned on row 2.
FETCH ROWSET STARTING AT ABSOLUTE 2 FOR 3 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 2, 3, and 4.
FETCH RELATIVE 2	Cursor is positioned on row 4.
FETCH ROWSET STARTING AT ABSOLUTE 2 FOR 4 ROWS	Cursor is positioned on a rowset of size 4, consisting of rows 2, 3, 4, and 5.
FETCH RELATIVE -1	Cursor is positioned on row 1.
FETCH ROWSET STARTING AT ABSOLUTE 3 FOR 2 ROWS	Cursor is positioned on a rowset of size 2, consisting of rows 3 and 4.
FETCH ROWSET STARTING AT RELATIVE 4	Cursor is positioned on a rowset of size 2, consisting of rows 7 and 8.
FETCH PRIOR	Cursor is positioned on row 6.
FETCH ROWSET STARTING AT ABSOLUTE 13 FOR 5 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 13, 14, and 15.
FETCH FIRST ROWSET	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5. Note: Even though the previous FETCH statement returned only 3 rows because EOF was encountered, DB2 will remember that 5 rows were requested by the previous FETCH statement.

Considerations for using the FOR n ROWS clause with the FETCH FIRST n ROWS ONLY clause: A clause specifying the desired number of rows can be specified in the SELECT statement of a cursor or the FETCH statement for a cursor, or both. However, these clauses have different effects:

- In the `SELECT` statement, a `FETCH FIRST n ROWS ONLY` clause controls the maximum number of rows that can be accessed with the cursor. When a `FETCH` statement attempts to retrieve a row beyond the number specified in the `FETCH FIRST n ROWS ONLY` clause of the `SELECT` statement, an end-of-data condition occurs.
- In a `FETCH` statement, a `FOR n ROWS` clause controls the number of rows that are returned for a single `FETCH` statement.

Both of these clauses can be specified.

Diagnostics information for rowset positioned `FETCH` statements: A single `FETCH` statement from a rowset cursor might encounter zero, one, or more conditions. If the current cursor position is not valid for the fetch orientation, a warning occurs and the statement terminates. If a warning or non-terminating error (such as a bind out error) occurs during the fetch of a row, processing continues. In this case, a summary message is returned for the `FETCH` statement, and additional information about each fetched row is available with the `GET DIAGNOSTICS` statement. Use the `GET DIAGNOSTICS` statement to obtain information about all of the conditions that are encountered for one of these `FETCH` statements. See “`GET DIAGNOSTICS`” on page 1317 for more information.

The `SQLCA` returns some information about errors and warnings that are found while fetching from a rowset cursor. Processing stops when the end of data is encountered, or when a terminating condition occurs. After each `FETCH` statement from a rowset cursor, information is returned to the program through the `SQLCA`. The `SQLCA` is set as follows:

- `SQLCODE` contains the `SQLCODE`.
- `SQLSTATE` contains the `SQLSTATE`.
- `SQLERRD1` and `SQLERRD2` contain the number of rows of the result table if the cursor is positioned on the last row of the result table.
- `SQLERRD3` contains the actual number of rows returned. If `SQLERRD3` is less than the number of rows requested, an error or end-of-data condition occurred.
- `SQLWARN` flags are set to represent all the warnings that were accumulated while processing the `FETCH` statement.

Consider the following examples, where 10 rows are fetched with a single `FETCH` statement.

- **Example 1:** Assume that an error is detected on the 5th row. `SQLERRD3` is set to 4 for the 4 returned rows, `SQLSTATE` is set to 22537, and `SQLCODE` is set to -354. This information is also available from the `GET DIAGNOSTICS` statement (the information that is returned is generated from connected server, which may differ across different servers). For example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 4 and num_cond = 1 (1 condition)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 22537, sqlcode = -354, and row_num = 5
```

- **Example 2:** Assume that an end-of-data condition is detected on the 6th row and that the cursor does not have immediate sensitivity to updates. `SQLERRD3` is set to 5 for the 5 returned rows, `SQLSTATE` is set to 02000, and `SQLCODE` is set to +100. This information is also available from the `GET DIAGNOSTICS` statement. For example:

```

GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 5 and num_cond = 1 (1 condition)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02000, sqlcode = 100, and row_num = 6

```

- **Example 3:** Assume that an bind out error condition is detected on the 5th row, the condition is recorded, and processing continues. Also, assume that an end-of-data condition is detected on the 8th row. SQLERRD3 is set to 7 for the 7 returned rows, SQLSTATE is set to 02000, and SQLCODE is set to +100. Processing to complete the FETCH statement is performed, and the bind out error that occurred is noted. An additional SQLCODE is recorded for the bind out error. SQLCODE is set to -354, and SQLSTATE is set to 01668. Use the GET DIAGNOSTICS statement to determine what went on. For example:

```

GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 7 and num_cond = 3 (3 conditions)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 01668, sqlcode = -354, and row_num = 0

GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02000, sqlcode = 100, and row_num = 0

GET DIAGNOSTICS CONDITION 3 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 22003, sqlcode = -302, and row_num = 5

```

In some cases, DB2 returns a warning if indicator variables are provided, or an error if indicator variables are not provided. These errors can be thought of as data mapping errors that result in a warning if indicator variables are provided.

- If indicator variables are provided, DB2 returns all rows to the user, marking the errors in the indicator variables. The SQLCODE and SQLSTATE contain the warning from the last data mapping error. The GET DIAGNOSTICS statement can be used to retrieve information about all the data mapping errors that have occurred.
- If some or no indicator variables are provided, all rows are returned as above until the first data mapping error that does not have indicator variables is detected. The rows successfully fetched are returned and the SQLSTATE, SQLCODE, and SQLWARN flags are set, if necessary. (The SQLCODE may be 0 or a positive value).

It is possible, if a data mapping error occurs, for the positioning of the cursor to be successful. In this case, the cursor is positioned on the rowset that encountered the data mapping error.

Consider the following examples, which try to fetch 10 rows with a single FETCH statement.

- **Example 1:** Assume that indicators have been provided for values returned for column 1, but not for column 2. The 5th row has a data mapping error (+802) for column 1, and the 7th row has a data mapping error for column 2 (-802 is returned because an indicator was not provided for column 2). SQLERRD3 is set to 6 for the 6 returned rows, SQLSTATE and SQLCODE are set to the error from the 7th row fetched. The indicator variable for the 5th row column 1 indicates

that a data mapping error was found. This information is also available from the GET DIAGNOSTICS statement, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 6 and num_cond = 2 (2 conditions)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 01519, sqlcode = +802, and row_num = 5

GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 22003, sqlcode = -802, and row_num = 7
```

The resulting cursor position is unknown.

- **Example 2:** Assume that null indicators are provided, that rows 3 and 5 are holes, and that data exists for the other requested rows. SQLERRD3 is set to 10 to reflect that 10 fetches were completed and that information has been returned for the 10 requested rows. Eight rows actually contain data. For two rows, indicator variables are set to indicate no data was returned for those rows. SQLSTATE is set to 02502, SQLCODE is set to +222, and all null indicators for rows 3 and 5 are set to -3 to indicate that a hole was detected. This information is also available from the GET DIAGNOSTICS statement, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
-- Results of the statement:
-- num_rows = 10 and num_cond = 2 (2 conditions)

GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02502, sqlcode = +222, and row_num = 3

GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
:sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
-- Results of the statement:
-- sqlstate = 02502, sqlcode = +222, and row_num = 5
```

If a null indicator was not provided for any variable in a row that was a hole, an error occurs.

SQLCA usage summary: For multiple-row-fetch, the fields of the SQLCA are set as follows:

Condition		Action: Resulting Values Stored in the SQLCA Fields		
Errors	Data	SQLSTATE	SQLCODE	SQLERRD3
No ¹	Return all requested rows	00000	0	Number of rows requested
No ¹	Return data for subset of requested rows, end of data	02000	+100	Number of rows rows
No ¹	Return all requested rows	sqlstate(2)	sqlcode(2)	Number of rows requested
Yes ¹	Return successfully fetched rows	sqlstate(3)	sqlcode(3)	Number of rows
Yes ¹	Return successfully fetched rows	sqlstate(4)	sqlcode(4)	Number of rows

Condition		Action: Resulting Values Stored in the SQLCA Fields		
Errors	Data	SQLSTATE	SQLCODE	SQLERRD3
Notes:				
1. SQLWARN flags may be set in all cases, even if there are no other warnings or errors indicated. The warning flags are an accumulation of all warning flags set while processing the multiple-row-fetch.				
2. sqlcode is the last positive SQLCODE, and sqlstate is the corresponding SQLSTATE value.				
3. Database Server detected error. sqlcode is the first negative SQLCODE encountered, sqlstate is the corresponding SQLSTATE value.				
4. Client detected error. sqlcode is the first negative SQLCODE encountered, sqlstate is one of the following SQLSTATES: 22002, 22008, 22509, 22518, or 55021.				

Providing indicator variables for error conditions: If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (division by zero or overflow) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.)

If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered. No value is assigned to the host variable or to later variables, though any values that have already been assigned to variables remain assigned. Additionally, a -3 is returned in all indicators provided by the application when a hole was detected for the row on a rowset positioned FETCH, and values were not returned for the row. Processing of the statement terminates if a hole is detected and at least one indicator variable was not provided by the application.

Alternative syntax and synonyms: USING DESCRIPTOR can be specified as a synonym for INTO DESCRIPTOR.

Example

Example 1: The FETCH statement fetches the results of the SELECT statement into the application program variables DNUM, DNAME, and MNUM. When no more rows remain to be fetched, the not found condition is returned.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM DSN8910.DEPT
  WHERE ADMRDEPT = 'A00';
EXEC SQL OPEN C1;
DO WHILE (SQLCODE = 0);
  EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM;
END;
EXEC SQL CLOSE C1;
```

Example 2: For an example of FETCH statements with a dynamic scrollable cursor, see Example 8.

Example 3: Fetch the last 5 rows of the result table C1 using cursor C1:

```
FETCH ROWSET STARTING AT ABSOLUTE -5
FROM C1 FOR 5 ROWS INTO DESCRIPTOR :MYDESCR;
```

Example 4: Fetch 6 rows starting at row 10 for cursor CURS1, and fetch the data into three host-variable-arrays:

```
FETCH ROWSET STARTING AT ABSOLUTE 10
FROM CURS1 FOR 6 ROWS
INTO :hav1, :hva2, :hva3;
```

Alternatively, a descriptor could have been specified in an INTO DESCRIPTOR clause where the information in the SQLDA reflects the data types of the host-variable-arrays:

```
FETCH ROWSET STARTING AT ABSOLUTE 10
FROM CURS1 FOR 6 ROWS
INTO DESCRIPTOR :MYDESCR;
```

FREE LOCATOR

The FREE LOCATOR statement removes the association between a LOB locator variable and its value.

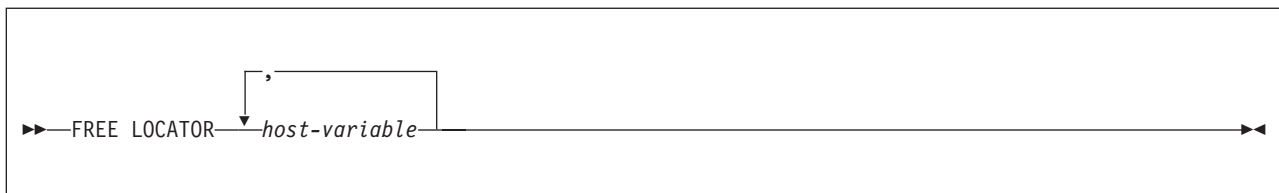
Invocation

This statement can only be embedded in an application program. It cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. FREE LOCATOR cannot be used with the EXECUTE IMMEDIATE statement. It must not be specified in Java.

Authorization

None required.

Syntax



Description

host-variable, ...

Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables. The locator variable type must be a binary large object locator, a character large object locator, or a double-byte character large object locator.

The *host-variable* must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a FETCH, SELECT INTO, assignment statement, SET *host-variable* statement, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is returned.

If more than one locator is specified and an error is returned on one of the locators, it is possible that some locators have been freed and others have not been freed.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the column values. Free the CLOB locator variables LOCRES and LOCHIST, and the BLOB locator variable LOCPIC.

```
EXEC SQL FREE LOCATOR :LOCRES, :LOCHIST, :LOCPIC
```

GET DIAGNOSTICS

The GET DIAGNOSTICS statement provides diagnostic information about the last SQL statement (other than a GET DIAGNOSTICS statement) that was executed. This diagnostic information is gathered as the previous SQL statement is executed. Some of the information available through the GET DIAGNOSTICS statement is also available in the SQLCA.

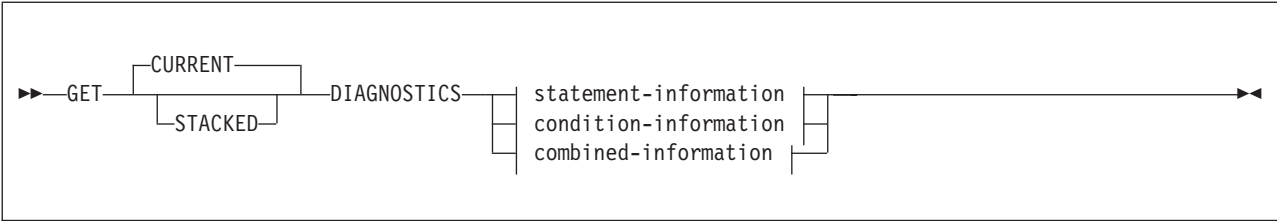
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

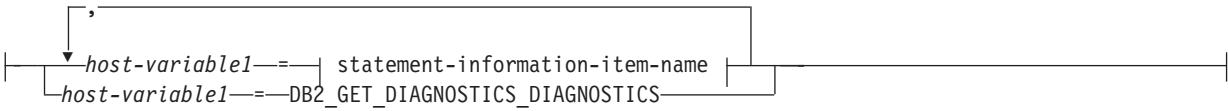
None required.

Syntax

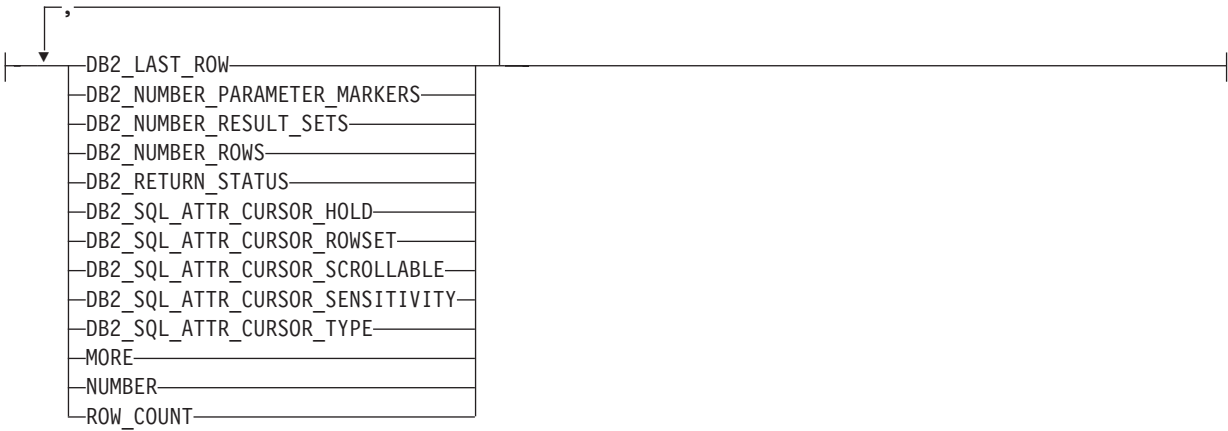


statement-information:

statement-information:

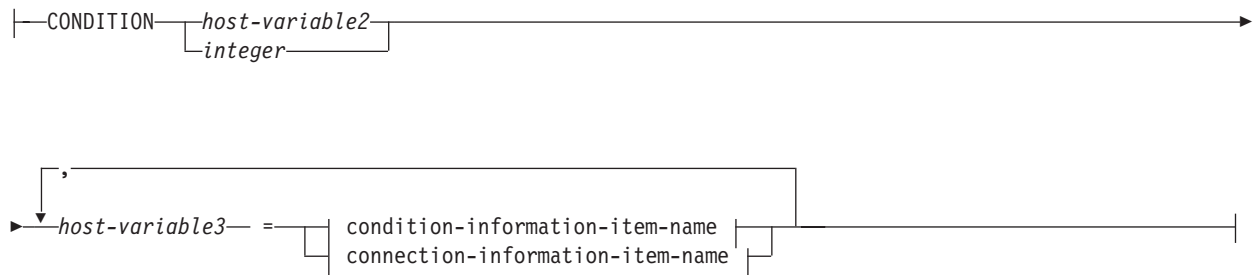


statement-information-item-name:

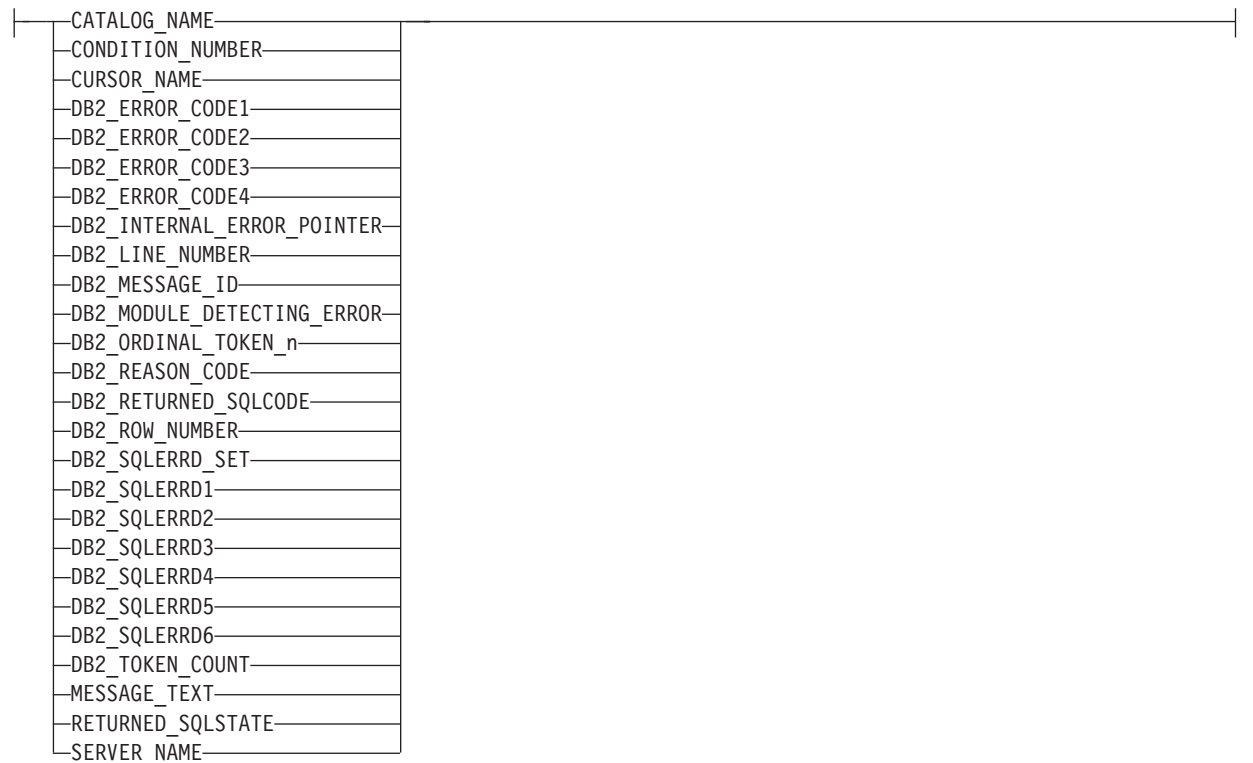


condition-information:

condition-information:



condition-information-item-name:

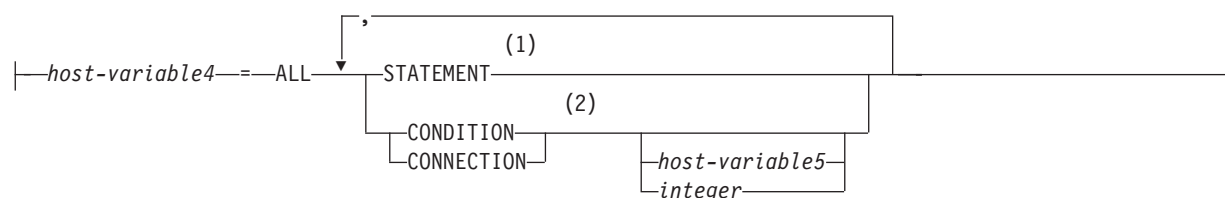


connection-information-item-name:



combined-information:

combined-information:



Notes:

- 1 STATEMENT can only be specified once.
- 2 CONDITION and CONNECTION can only be specified once if host-variable5 or integer is not also specified.

Description

Diagnostic information is provided in three main areas: statement information, condition information, and combined information. After the execution of an SQL statement, information about the execution of the statement is provided as statement information, and at least one instance of condition information is provided. The number of instances of the condition information is indicated by the NUMBER item that is available in the statement information. Combined information contains a text representation of all the information gathered about the execution of the SQL statement.

The diagnostic information that is provided is specific to the server. If you are connected to a server other than DB2 for z/OS, see that product's documentation for the diagnostic information that is returned.

CURRENT

Specifies that information is to be returned from the first diagnostics area. It corresponds to the previous SQL statement that was executed that was not a GET DIAGNOSTICS or compound statement. CURRENT is the default.

STACKED

Specifies that information is to be returned from the stacked diagnostics area. The stacked diagnostics area is only available within a handler in a native SQL procedure. The stacked diagnostics area corresponds to the previous SQL statement (that was not a GET DIAGNOSTICS or compound statement) that was executed before the handler was entered. If the GET DIAGNOSTICS statement is the first statement within a handler, the current diagnostics area and the stacked diagnostics area contain the same diagnostics information.

statement-information

Provides information about the last SQL statement executed.

host-variable1

Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be the data type as specified in Data types for GET DIAGNOSTICS items.

The host variable is assigned the value of the specified statement information item. If the value is truncated when assigning it to the host variable, a warning is returned and the GET_DIAGNOSTICS_DIAGNOSTICS item of the

diagnostics area is updated with the details of this condition. If a DIAGNOSTICS item is not set, then the host variable is set to a default value, based on its data type: 0 for an exact numeric field, an empty string for a VARCHAR field, and blanks for a CHAR field.

DB2_GET_DIAGNOSTICS_DIAGNOSTICS

Contains textual information about errors or warnings that might have occurred in the execution of the GET DIAGNOSTICS statement. The format of the information is similar to what would be returned by a GET DIAGNOSTICS :hv = ALL statement.

statement-information-item-name:

DB2_LAST_ROW

For a multiple-row FETCH statement, contains a value of +100 if the last row currently in the table is in the set of rows that have been fetched. For cursors that are not sensitive to updates, there would be no need to do a subsequent FETCH, because the result would be an end-of-data indication. For cursors that are sensitive to updates, a subsequent FETCH may return more data if a row had been inserted before the FETCH was executed. For statements other than multiple-row FETCH statements, or for multiple-row FETCH statements that do not contain the last row, this variable contains the value 0.

An end of data warning might not occur and DB2_LAST_ROW might not contain +100 when the number of rows returned is equal to the number of rows requested and the last row of data returned is the last row of data.

DB2_NUMBER_PARAMETER_MARKERS

For a PREPARE statement, contains the number of parameter markers in the prepared statement. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_NUMBER_RESULT_SETS

For a CALL statement, contains the actual number of result sets returned by the procedure. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_NUMBER_ROWS

If the previous SQL statement was an OPEN or a FETCH that caused the size of the result table to be known, returns the number of rows in the result table. For SENSITIVE DYNAMIC cursors, this value can be thought of as an approximation because rows that are inserted and deleted will affect the next retrieval of this value. If the previous SQL statement was a PREPARE statement, returns the estimated number of rows in the result table for the prepared statement. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_RETURN_STATUS

Identifies the status value returned from the stored procedure associated with the previously executed SQL statement, provided that the statement was a CALL statement that invoked a procedure that returns a status. Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_SQL_ATTR_CURSOR_HOLD

For an ALLOCATE or OPEN statement, indicates whether a cursor can be held open across multiple units of work.

- N indicates that this cursor does not remain open across multiple units of work.

- Y indicates that this cursor remains open across multiple units of work. Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_ROWSET

For an ALLOCATE or OPEN statement, indicates whether or not a cursor can be accessed using rowset positioning.

- N indicates that this cursor supports only row positioned operations.
- Y indicates that this cursor supports rowset positioned operations.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_SCROLLABLE

For an ALLOCATE or OPEN statement, indicates whether or not a cursor can be scrolled forward and backward.

- N indicates that this cursor is not scrollable.
- Y indicates that this cursor is scrollable.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_SENSITIVITY

For an ALLOCATE or OPEN statement, indicates whether or not a cursor does or does not show updates to cursor rows made by other connections.

- I indicates insensitive.
- S indicates sensitive.

Otherwise, a blank is returned.

DB2_SQL_ATTR_CURSOR_TYPE

For an ALLOCATE or OPEN statement, indicates the type of cursor, whether a cursor type is forward-only, static, or dynamic.

- F indicates a forward cursor.
- D indicates a dynamic cursor.
- S indicates a static cursor.

Otherwise, a blank is returned.

MORE

Indicates whether some of the warning and errors from the previous SQL statement were stored or discarded.

- N indicates that all the warnings and errors from the previous SQL statement are stored in the diagnostic area.
- Y indicates that some of the warnings and errors from the previous SQL statement were discarded because the amount of storage needed to record warnings and errors exceeded 65535 bytes.

NUMBER

Returns the number of errors and warnings detected by the execution of the previous SQL statement, other than a GET DIAGNOSTICS statement, that have been stored in the diagnostics area. If the previous SQL statement returned an SQLSTATE of 00000 or no previous SQL statement has been executed, the number returned is one.

The GET DIAGNOSTICS statement itself may return information via the SQLSTATE parameter, but does not modify the previous contents of the diagnostics area, except for the DB2_GET_DIAGNOSTICS_DIAGNOSTICS item.

ROW_COUNT

Identifies the number of rows associated with the previous SQL statement that was executed.

If the previous SQL statement is a DELETE, INSERT, UPDATE, or MERGE statement, ROW_COUNT indicates the number of rows that qualified to be deleted, inserted, or updated by that statement, excluding rows affected by either triggers or referential integrity constraints.

For the OPEN of a cursor for a SELECT with a data change statement, or a SELECT INTO statement, SQLERRD(3) contains the number of rows affected by the embedded data change statement. The value is 0 if the SQL statement fails, indicating that all changes made in executing the statement canceled.

A value of -1 indicates a mass delete from a table in a segmented table space and the DELETE statement did not include selection criteria, or a truncate operation. If the delete was against a view, then neither the DELETE statement nor the definition of the view included selection criteria.

For a REFRESH TABLE statement, SQLERRD(3) contains the number of rows inserted into the materialized query table.

If the previous SQL statement is a multiple-row FETCH, ROW_COUNT identifies the number of rows fetched.

Otherwise, or if the server only returns an SQLCA, the value zero is returned.

condition-information

Assigns the values of the specified condition information to the associated host variables. The host variable specified must be of the data type that is compatible with the data type of the specified diagnostic-ID or an error occurs. If the value of the condition is truncated when assigning it to the host variable, an error occurs. If an indicator variable was provided, the length of the value is returned in the indicator variable.

If a DIAGNOSTICS item is not set, then the host variable is set to a default value, based on the data type of the item. The specific value will be 0 for a numeric field, an empty string for a VARCHAR field, and blanks for a CHAR field.

***host-variable2* or integer**

Identifies the diagnostic for which information is requested. Each diagnostic that occurs while executing an SQL statement is assigned an integer. The value 1 indicates the first diagnostic, 2 indicates the second diagnostic, and so on. If the value is 1, the diagnostic information that is retrieved corresponds to the condition that is indicated by the SQLSTATE value actually returned by the execution of the previous SQL statement (other than a GET DIAGNOSTICS statement). The host variable specified must be an integer data type or an error occurs. An indicator variable is not allowed for this host variable. If a value is specified that is less than or equal to zero or greater than the number of available diagnostics, an error occurs.

host-variable3

Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be the data type as specified in Data types for GET DIAGNOSTICS items for the indicated condition-information item.

condition-information-item-name

CATALOG_NAME

If the returned SQLSTATE is any one of the following values, the constraint that caused the error is a referential, check, or unique constraint. The location (RDB) name of the server that generated the condition is returned.

- Class 09 (Triggered Action Exception),
- Class 23 (Integrity Constraint Violation)
- Class 27 (Triggered Data Change Violation)
- 40002 (Transaction Rollback - Integrity Constraint Violation)
- 40004 (Transaction Rollback - Triggered Action Exception)

If the returned SQLSTATE is class 42 (Syntax Error or Access Rule Violation), the server name of the table that caused the error is returned.

If the returned SQLSTATE is class 44 (WITH CHECK OPTION Violation), the server name of the view that caused the error is returned.

Otherwise, the empty string is returned.

The actual server name may be different than the server name specified, either implicitly or explicitly, on the CONNECT statement because of the use of aliases or synonyms.

CONDITION_NUMBER

Returns the number of the diagnostic returned.

CURSOR_NAME

If the returned SQLSTATE is class 24 (Invalid Cursor State), the name of the cursor is returned. Otherwise, the empty string is returned.

DB2_ERROR_CODE1

Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_ERROR_CODE2

Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_ERROR_CODE3

Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_ERROR_CODE4

Returns an internal error code. Otherwise, or if the server only returns an SQLCA, the value 0 is returned.

DB2_INTERNAL_ERROR_POINTER

For some errors, this is a negative value that is an internal error pointer. Otherwise, the value 0 is returned.

DB2_LINE_NUMBER

Returns the line number where an error is encountered in parsing a dynamic statement. Also returns the line number where an error is encountered in parsing, binding, or executing a CREATE or ALTER statement for a native SQL procedure. DB2_LINE_NUMBER also returns the line number when a CALL statement invokes a native SQL procedure and the procedure returns with an error. This information is not returned for an external SQL procedure.

This value will only be meaningful if the statement source contains new line control characters.

DB2_MESSAGE_ID

Corresponds to the message that is contained in the MESSAGE_TEXT diagnostic item (for example, DSNT102I or DSNU180I).

DB2_MODULE_DETECTING_ERROR

Returns an identifier indicating which module detected the error. For a SIGNAL statement that is issued from a routine, the value 'ROUTINE' is returned. Otherwise, the string 'DSN' is returned.

DB2_ORDINAL_TOKEN_n

Returns the *n*th token. *n* must be a value from 1 to 100. For example, DB2_ORDINAL_TOKEN_1 would return the value of the first token, DB2_ORDINAL_TOKEN_2 the second token, and so on. A numeric value for a token is converted to characters before being returned. If there is no value for the token, or if the server only returns an SQLCA, an empty string is returned.

DB2_REASON_CODE

Contains the reason code for errors that have a reason code token in the message text. Otherwise, the value zero is returned.

DB2_RETURNED_SQLCODE

Returns the SQLCODE for the specified diagnostic.

DB2_ROW_NUMBER

Returns the number of the row where the condition was encountered, when such information is available and applicable. If SQLCODE +1– or +20237 is returned, DB2_ROW_NUMBER returns a value of 0.

DB2_SQLERRD_SET

A value of Y indicates that the DB2_SQLERRD1 through DB2_SQLERRD items might be set. These items are set only when communicating with a server that returns the SQLCA SQL communications area and not the new diagnostics area. Otherwise, a blank is returned.

DB2_SQLERRD1

Returns the value of sqlerrd(1) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD2

Returns the value of sqlerrd(2) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD3

Returns the value of sqlerrd(3) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD4

Returns the value of sqlerrd(4) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD5

Returns the value of sqlerrd(5) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_SQLERRD6

Returns the value of sqlerrd(6) from the SQLCA that is returned by the server. Otherwise, the value zero is returned.

DB2_TOKEN_COUNT

Returns the number of tokens available for the specified diagnostic ID.

MESSAGE_TEXT

Returns the message text that is associated with the SQLCODE. This is the short text, including substituted tokens. The message text does not contain the message number. When the SQLCODE is 0, the empty string is returned, even if the RETURNED_SQLSTATE value indicates a warning condition.

RETURNED_SQLSTATE

Returns the SQLSTATE for the specified diagnostic.

SERVER_NAME

If the previous SQL statement is a CONNECT, DISCONNECT, or SET CONNECTION statement, returns the name of the server specified in the previous statement is returned. Otherwise, the name of the server where the statement executes is returned.

connection-information-item-name

Provides information about the last SQL statement executed if it was a CONNECT statement.

DB2_AUTHENTICATION_TYPE

Contains an authentication type value of:

- 'S' for a server authentication
- 'C' for client authentication
- 'T' for trusted server authentication
- Otherwise, or if the server only returns an SQLCA, a blank is returned

DB2_AUTHORIZATION_ID

Authorization ID used by connected server. Because of user ID translation and authorization exits, the local user ID may not be the authorized ID used by the server.

DB2_CONNECTION_STATE

Contains the connection state:

- -1 if the connection is unconnected
- 1 if the connection is connected

Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_CONNECTION_STATUS

Contains a value of:

- 1 if committable updates can be performed on the connection for this unit of work
- 2 if no committable updates can be performed on the connection for this unit of work

Otherwise, or if the server only returns an SQLCA, the value zero is returned.

DB2_SERVER_CLASS_NAME

For a CONNECT or SET CONNECTION statement, contains one of the following values:

- QAS for DB2 for iSeries
- QDB2 for DB2 for OS/390® and z/OS
- QDB2/2 for DB2 for OS/2
- QDB2/6000 for DB2 for AIX®
- QDB2/6000 PE for DB2 for AIX Parallel Edition

- QDB2/AIX64 for DB2 for AIX 64-bit
- QDB2/HPUX for DB2 for HP-UX
- QDB2/HP64 for DB2 for HP-UX 64-bit
- QDB2/LINUX for DB2 for Linux
- QDB2/LINUX390 for DB2 for Linux
- QDB2/LINUXIA64 for DB2 for Linux
- QDB2/LINUXPPC for DB2 for Linux
- QDB2/LINUXPPC64 for DB2 for Linux
- QDB2/LINUXZ64 for DB2 for Linux
- QDB2/NT for DB2 for NT
- QDB2/NT64 for DB2 for NT 64-bit
- QDB2/PTX for DB2 for NUMA-Q®
- QDB2/SCO for DB2 for SCO UnixWare
- QDB2/SGI for DB2 for Silicon Graphics
- QDB2/SNI for DB2 for Siemens Nixdorf
- QDB2/SUN for DB2 for SUN Solaris
- QDB2/SUN64 for DB2 for SUN Solaris 64-bit
- QDB2/Windows 95 for DB2 for Windows 95 or Windows 98
- QSQLDS/VM for DB2 for VM and VSE
- QSQLDS/VSE for DB2 for VM and VSE

Otherwise, the empty string is returned.

DB2_ENCRYPTION_TYPE

The level of encryption for the connection:

- A indicates only the authentication tokens (authid and password) are encrypted.
- D indicates all data is encrypted for the connection.
- Otherwise, a blank is returned.

DB2_PRODUCT_ID

Returns a product signature. If the application server is an IBM relational database product, the form is *pppvrrm*, where:

- *ppp* identifies the product as follows:
 - ARI for DB2 Server for VSE & VM
 - DSN for DB2 for z/OS
 - QSQ for DB2 for i
 - SQL for all other DB2 products
- *vv* is a two-digit version identifier such as '09'
- *rr* is a two-digit release identifier such as '01'
- *m* is a one-digit maintenance level identifier such as '5' (Values 0, 1, 2, 3, and 4 are for maintenance levels in compatibility and enabling-new-function mode. Values 5, 6, 7, 8, and 9 are for maintenance levels in new-function mode.)

For example, if the application server is Version 9 of DB2 for z/OS in new-function mode with the latest maintenance, the value would be 'DSN09015'.

combined-information

Provides a text representation of all the information gathered about the execution of the SQL statement.

ALL

Indicates that all diagnostic items that are set for the last SQL statement executed are to be combined into one string. The format of the string is a semicolon separated list of all of the available diagnostic information in the form: <item-name>[(<condition-number>)]=<value-converted-to-character>;... as shown in the following example:

```
NUMBER=1;RETURNED_SQLSTATE=02000;DB2_RETURNED_SQLCODE=+100;
```

host-variable4

Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be VARCHAR. If the length of *host-variable4* is not sufficient to hold the full returned diagnostic string, the string is truncated, a warning is returned, and the GET_DIAGNOSTICS_DIAGNOSTICS item of the diagnostics area is updated with the details of this condition.

STATEMENT

Indicates that all statement-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. The format is the same as described for the ALL option.

CONDITION

Indicates that all condition-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. If *host-variable5* or integer is supplied after CONDITION, the format is the same as described above for the ALL option. If *host-variable5* or integer is not supplied, the format includes a condition number entry at the beginning of the information for that condition in the form:

CONDITION_NUMBER=X;<item-name>=<value-converted-to-character>;... where X is the number of the condition, as shown in the following example:

```
CONDITION_NUMBER=1;RETURNED_SQLSTATE=02000;RETURNED_SQLCODE=100;
CONDITION_NUMBER=2;RETURNED_SQLSTATE=01004;
```

CONNECTION

Indicates that all connection-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. If *host-variable5* or integer is supplied after CONNECTION, the format is the same as described for the ALL option. If *host-variable5* or integer is not supplied, then the format includes a condition number entry at the beginning of the information for that condition in the form:

CONNECTION_NUMBER=X;<item-name>=<value-converted-to-character>;... where X is the number of the condition, as shown in the following example:

```
CONNECTION_NUMBER=1;CONNECTION_NAME=SVL1;DB2_PRODUCT_ID=DSN08010;
```

host-variable5 or integer

Identifies the diagnostic for which ALL CONDITION or ALL CONNECTION information is requested. The host variable specified must be an integer data type or an error occurs. An indicator variable is not allowed for this host variable or an error occurs. If a value is specified that is less than or equal to zero or greater than the number of available diagnostics, an error occurs.

Notes

Effect of the statement in a native SQL procedure: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area (SQLCA), or the SQLSTATE and SQLCODE SQL variables. If the SQLSTATE and SQLCODE SQL variables are declared in an SQL procedure, they will contain the same values as would be returned from issuing the GET DIAGNOSTICS statement specifying RETURNED_SQLSTATE and DB2_RETURNED_SQLCODE.

If information is desired about an error, the GET DIAGNOSTICS statement must be the first executable statement specified in the handler that will handle the error condition.

If information is desired about a warning and a handler will get control for the warning condition, the GET DIAGNOSTICS statement must be the first executable statement specified in that handler.

If information is desired about a warning and a handler will not get control for the warning condition, the GET DIAGNOSTICS statement must be the next statement executed after that previous statement.

Data types for items: When a diagnostic item is assigned to a host variable, SQL variable, or SQL parameter, the data type of the target must be compatible with the data type of the requested diagnostic item.

Data types for GET DIAGNOSTICS items

Table 124. Data types for GET DIAGNOSTICS items

Type of information	Item	Data type
Statement Information	DB2_GET_DIAGNOSTICS_DIAGNOSTICS	VARCHAR(32672)
	DB2_LAST_ROW	INTEGER
	DB2_NUMBER_PARAMETER_MARKERS	INTEGER
	DB2_NUMBER_RESULT_SETS	INTEGER
	DB2_NUMBER_ROWS	DECIMAL(31,0)
	DB2_RETURN_STATUS	INTEGER
	DB2_SQL_ATTR_CURSOR_HOLD	CHAR(1)
	DB2_SQL_ATTR_CURSOR_ROWSET	CHAR(1)
	DB2_SQL_ATTR_CURSOR_SCROLLABLE	CHAR(1)
	DB2_SQL_ATTR_CURSOR_SENSITIVITY	CHAR(1)
	DB2_SQL_ATTR_CURSOR_TYPE	CHAR(1)
	MORE	CHAR(1)
	NUMBER	INTEGER
	ROW_COUNT	DECIMAL(31,0)

Table 124. Data types for GET DIAGNOSTICS items (continued)

Type of information	Item	Data type
Condition Information	CATALOG_NAME	VARCHAR(128)
	CONDITION_NUMBER	INTEGER
	CURSOR_NAME	VARCHAR(128)
	DB2_ERROR_CODE1	INTEGER
	DB2_ERROR_CODE2	INTEGER
	DB2_ERROR_CODE3	INTEGER
	DB2_ERROR_CODE4	INTEGER
	DB2_INTERNAL_ERROR_POINTER	INTEGER
	DB2_LINE_NUMBER	INTEGER
	DB2_MESSAGE_ID	CHAR(10)
	DB2_MODULE_DETECTING_ERROR	CHAR(8)
	DB2_ORDINAL_TOKEN_n	VARCHAR(515)
	DB2_REASON_CODE	INTEGER
	DB2_RETURNED_SQLCODE	INTEGER
	DB2_ROW_NUMBER	DECIMAL(31,0)
	DB2_TOKEN_COUNT	INTEGER
	MESSAGE_TEXT	VARCHAR(32672)
	RETURNED_SQLSTATE	CHAR(5)
	SERVER_NAME	VARCHAR(128)
Connection Information	DB2_AUTHENTICATION_TYPE	CHAR(1)
	DB2_AUTHORIZATION_ID	VARCHAR(128)
	DB2_CONNECTION_STATE	INTEGER
	DB2_CONNECTION_STATUS	INTEGER
	DB2_ENCRYPTION_TYPE	CHAR(1)
	DB2_PRODUCT_ID	VARCHAR(8)
	DB2_SERVER_CLASS_NAME	CHAR(128)
Combined Information	ALL	VARCHAR(32672)

DRDA considerations The GET DIAGNOSTICS statement is supported from a current DB2 for z/OS client, regardless of the level of the server (a DB2 for z/OS Version 7 or a DB2 for Windows Version 7, for example). When the application is connected to servers that do not support the Open Group Version 3 DRDA standard, the diagnostic information that is returned by the servers is available in the condition information.

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports the following keywords:

- RETURN_STATUS as a synonym for DB2_RETURN_STATUS
- EXCEPTION as a synonym for CONDITION

Examples

Example 1: In an application, use the GET DIAGNOSTICS statement to determine how many rows were updated.

```
long rcount;
EXEC SQL UPDATE T1 SET C1 = C1 + 1;
EXEC SQL GET DIAGNOSTICS :rcount = ROW_COUNT;
```

After execution of this code segment, rcount will contain the number of rows that were updated.

Example 2: In an application, use the GET DIAGNOSTICS statement to handle multiple SQL Errors.

```
long numerrors, counter;
char retsqlstate[5];
long hva[5];
EXEC SQL INSERT INTO T1 FOR 5 ROWS VALUES (:hva) NOT ATOMIC
CONTINUE ON SQLEXCEPTION;
EXEC SQL GET DIAGNOSTICS :numerrors = NUMBER;
for ( i=1; i < numerrors; i++)
{
    EXEC SQL GET DIAGNOSTICS CONDITION :i :retsqlstate = RETURNED_SQLSTATE;
```

Execution of this code segment sets and prints retsqlstate with the SQLSTATE for each error that was encountered in the previous SQL statement.

Example 3: Retrieve information about a connection.

```
EXEC SQL GET DIAGNOSTICS :HV_PRODUCT_ID = DB2_PRODUCT_ID;
```

Example 4: Use the GET DIAGNOSTICS statement to retrieve information that is similar to what is returned in the SQLCA:

```
EXEC SQL GET DIAGNOSTICS CONDITION 1
:dasqlcode   = DB2_RETURNED_SQLCODE,
:datokencnt  = DB2_TOKEN_COUNT,
:datoken1    = DB2_ORDINAL_TOKEN_1,
:datoken2    = DB2_ORDINAL_TOKEN_2,
:datoken3    = DB2_ORDINAL_TOKEN_3,
:datoken4    = DB2_ORDINAL_TOKEN_4,
:datoken5    = DB2_ORDINAL_TOKEN_5,
:dasqlerrd1b = DB2_MESSAGE_ID,
:damsqtext   = MESSAGE_TEXT,
:dasqlerrp   = DB2_MODULE_DETECTING_ERROR,
:dasqlstate  = RETURNED_SQLSTATE;
```

Example 5: Specify the **STACKED** keyword on a GET DIAGNOSTICS statement that is used within a handler to access information in the diagnostics area that caused the handler to be activated:

```
CREATE PROCEDURE divide2 ( IN numerator INTEGER,
                          IN denominator INTEGER,
                          OUT divide_result INTEGER,
                          OUT divide_error VARCHAR(70))
    LANGUAGE SQL
BEGIN
    DECLARE msg_text      CHAR(70) DEFAULT '';
    DECLARE divide_error  CHAR(70) DEFAULT '';

    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
```

```

|          BEGIN
|          INSERT .....;  -- insert row into a log table
|
|          -- get diagnostic information for the INSERT statement
|          GET CURRENT DIAGNOSTICS CONDITION 1 msg_text = MESSAGE_TEXT;
|
|          -- get information about condition that activated the handler
|          GET STACKED DIAGNOSTICS CONDITION 1 divide_error = MESSAGE_TEXT;
|          END;
|
|          SET divide_result = numerator/denominator;
|          END;

```

| The first GET DIAGNOSTICS statement obtains diagnostic information about the
| INSERT statement.

| The second GET DIAGNOSTICS statement specifies the STACKED keyword. The
| use of the STACKED keyword allows access the stacked diagnostics area which
| contains the diagnostic information for the condition that caused the handler to be
| activated. The information about the original condition is still accessible within the
| handler even after another statement has been issued, such as the INSERT
| statement in the example.

GRANT

The DB2 GRANT statement grants privileges to authorization IDs. There is a separate form of the statement for each of these classes of privilege:

- Collection
- Database
- Distinct type
- Function or stored procedure
- Package
- Plan
- Schema
- Sequence
- System
- Table or view
- Use

The applicable objects are always at the current server. The grants are recorded in the current server's catalog.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

If the authorization mechanism was not activated when the DB2 subsystem was installed, an error condition occurs.

Authorization

To grant a privilege P, the privilege set must include one of the following:

- The privilege P WITH GRANT OPTION
- Ownership of the object on which P is a privilege
- SYSADM authority

The presence of SYSCTRL authority in the privilege set allows the granting of all authorities except:

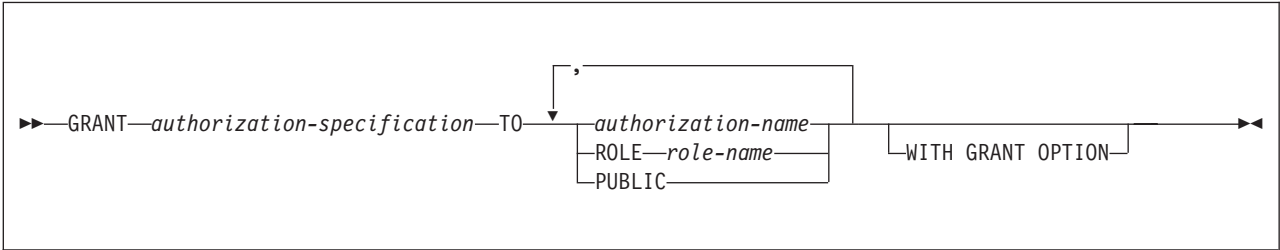
- DBADM on databases
- DELETE, INSERT, SELECT, and UPDATE on user tables or views
- EXECUTE on plans, packages, functions, or stored procedures
- PACKADM on collections
- SYSADM authority
- USAGE on distinct types, JARs, and sequences

Except for views, the GRANT option for privileges on a table is also inherent in DBADM authority for its database, provided DBADM authority was acquired with the GRANT option. See “CREATE VIEW” on page 1184 for a description of the rules that apply to views.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. The owner can be a

role. If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. However, if the process is running in a trusted context that is defined with the `ROLE AS OBJECT OWNER` `CLAUSE`, the privilege set is the privileges that are held by the role in effect.

Syntax



Description

authorization-specification
Specifies one or more privileges for the class of privilege. The same privilege must not be specified more than once.

TO Specifies to what authorization IDs the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

ROLE *role-name*
Lists one or more role names. Each name must identify a role that exists at the current server.

The value of the `CURRENT RULES` special register determines whether you can use the ID or role of the `GRANT` statement itself (to grant privileges to yourself). When `CURRENT RULES` is:

DB2 You cannot use the ID or role of the `GRANT` statement.

STD
You can use the ID or role of the `GRANT` statement.

PUBLIC
Grants the privileges to all users at the current server, including database requesters using DRDA access.

WITH GRANT OPTION
Allows the named users to grant the privileges to others. Granting an administrative authority with this option allows the user to specifically grant any privilege belonging to that authority. If you omit `WITH GRANT OPTION`, the named users cannot grant the privileges to others unless they have that authority from some other source.

`GRANT` authority cannot be passed to `PUBLIC`. When `WITH GRANT OPTION` is used with `PUBLIC`, a warning is issued, and the named privileges are granted, but without `GRANT` authority.

Notes

For more on DB2 privileges, read *DB2 Administration Guide*.

A *grant* is the granting of a specific privilege by a specific grantor to a specific grantee. The grantor for a given `GRANT` statement is the authorization ID for the

| privilege set; that is, the SQL authorization ID of the process or a role, or the
| authorization ID of the owner of the plan or package. Grant statements that are
| made in a trusted context that is defined with the ROLE AS OBJECT OWNER
| clause result in the grantor being the role that is in effect. If the statement is
| prepared dynamically, the grantor is the role that is associated with the ID that is
| running the statement. If the statement is embedded in an application program
| that was bound in a trusted context that was defined with the ROLE AS OBJECT
| OWNER clause the owner of the plan or package is a role which is the grantor. If
| the ROLE AS OBJECT OWNER clause is not specified for the trusted context, the
| grantor is the authorization ID of the process.

| The grantee, as recorded in the catalog, is an authorization ID or PUBLIC.

Duplicate grants from the same grantor are not recorded in the catalog. Otherwise,
the result of executing a GRANT statement is recorded as one or more grants in
the current server's catalog.

If more than one privilege or *authorization-name* is specified after the TO keyword
and one of the grants is in error, execution of the statement is stopped and no
grants are made. The status of the privilege or privileges granted is recorded in the
catalog for each *authorization-name*.

Different grantors can grant the same privilege to a single grantee. The grantee
retains that privilege as long as one or more of those grants are recorded in the
catalog. Privileges that imply other privileges are also termed *authorities*. Grants are
removed from the catalog by executing SQL REVOKE statements.

Whenever a grant is made for a database, distinct type, package, plan, schema,
stored procedure, table, trigger, user-defined function, view, or USE privilege for
an object that does not exist, an SQL return code is issued and the grant is not
made.

| **PUBLIC AT ALL LOCATIONS:** PUBLIC AT ALL LOCATIONS can continue to be
| specified as an alternative to PUBLIC as in prior releases. PUBLIC AT ALL
| LOCATIONS specifies that the privileges are to be granted to all users in the
| network. The PUBLIC AT ALL LOCATIONS clause applies to table privileges only,
| excluding ALTER, INDEX, REFERENCES, and TRIGGER.

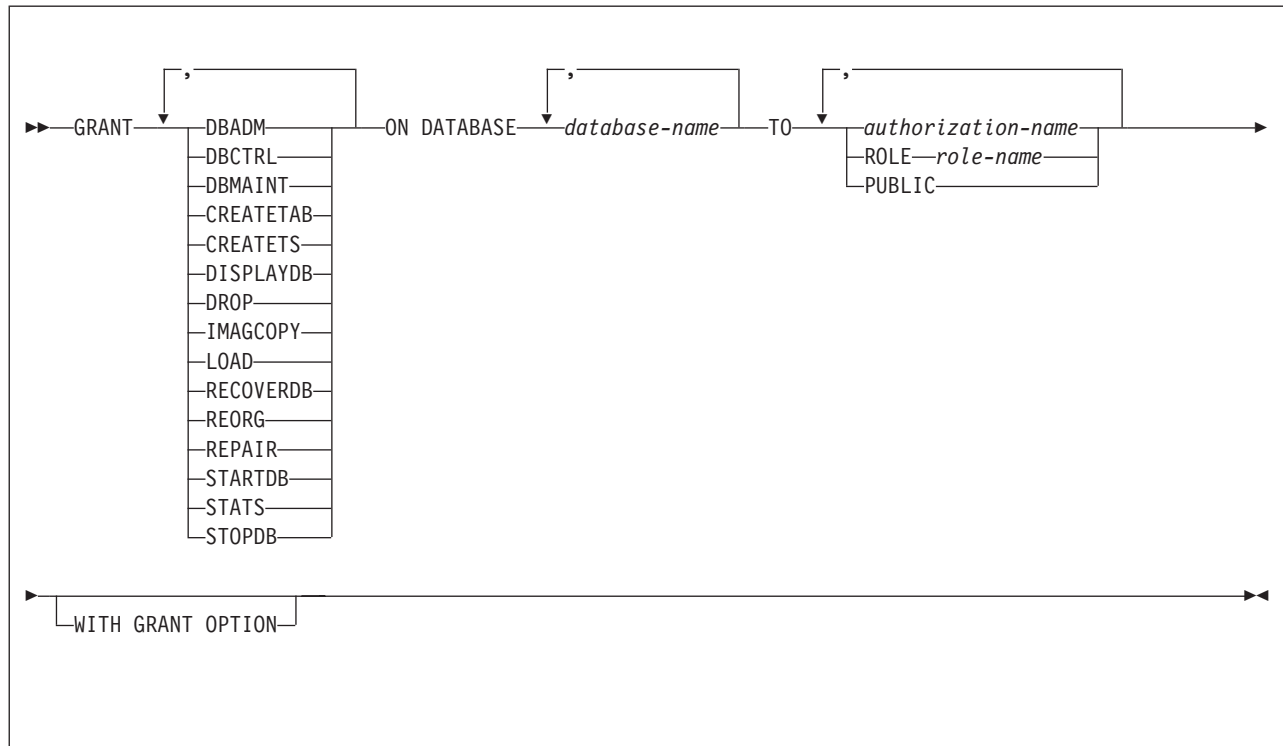
| PUBLIC AT ALL LOCATIONS applies to DB2 private protocol access only.

| The grantee, as recorded in the catalog for PUBLIC AT ALL LOCATIONS is
| PUBLIC*.

GRANT (database privileges)

This form of the GRANT statement grants privileges on databases.

Syntax



Description

Each keyword listed grants the privilege described, but only as it applies to or within the databases named in the statement.

DBADM

Grants the database administrator authority.

DBCTRL

Grants the database control authority.

DBMAINT

Grants the database maintenance authority.

CREATETAB

Grants the privilege to create new tables. To create tables in an implicitly created database, CREATETAB privileges are needed on the DSNDB04 database. For a work file database, PUBLIC implicitly has the CREATETAB privilege (without GRANT authority) to define declared temporary tables; this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

CREATETS

Grants the privilege to create new table spaces.

DISPLAYDB

Grants the privilege to issue the DISPLAY DATABASE command.

DROP

Grants the privilege to issue the DROP or ALTER DATABASE statements for the designated databases.

IMAGCOPY

Grants the privilege to run the COPY, MERGECOPY, and QUIESCE utilities against table spaces of the specified databases, and to run the MODIFY RECOVERY utility.

LOAD

Grants the privilege to use the LOAD utility to load tables.

RECOVERDB

Grants the privilege to use the RECOVER and REPORT utilities to recover table spaces and indexes.

REORG

Grants the privilege to use the REORG utility to reorganize table spaces and indexes.

REPAIR

Grants the privilege to use the REPAIR and DIAGNOSE utilities.

STARTDB

Grants the privilege to issue the START DATABASE command.

STATS

Grants the privilege to use the RUNSTATS utility to update statistics, the CHECK utility to test whether indexes are consistent with the data they index, and the MODIFY STATISTICS utility to delete unwanted statistics history records from the corresponding catalog tables.

STOPDB

Grants the privilege to issue the STOP DATABASE command.

ON DATABASE *database-name,...*

Identifies databases on which privileges are to be granted. For each named database, the grantor must have all the specified privileges with the GRANT option. Each name must identify a database that exists at the current server. DSNDB01 must not be identified; however, a grant of a privilege on DSNDB06 implies the granting of the same privilege on DSNDB01 for utility operations only.

Database privileges granted on DSNDB04 are applicable to all implicitly created databases. This means that a user with the STOPDB privilege on DSNDB04 can also stop database objects in any implicitly created database. Similarly, having DBADM on DSNDB04 allows access to all tables in all implicitly created databases. However, having a database privilege on DSNDB04 does not allow granting of this privilege on an implicitly created database to others.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant drop privileges on database DSN8D91A to user PEREZ.

```
GRANT DROP
  ON DATABASE DSN8D91A
  TO PEREZ;
```

Example 2: Grant repair privileges on database DSN8D91A to all local users.

```
GRANT REPAIR
  ON DATABASE DSN8D91A
  TO PUBLIC;
```

Example 3: Grant authority to create new tables and load tables in database DSN8D91A to users WALKER, PIANKA, and FUJIMOTO, and give them grant privileges.

```
GRANT CREATETAB,LOAD
  ON DATABASE DSN8D91A
  TO WALKER,PIANKA,FUJIMOTO
  WITH GRANT OPTION;
```

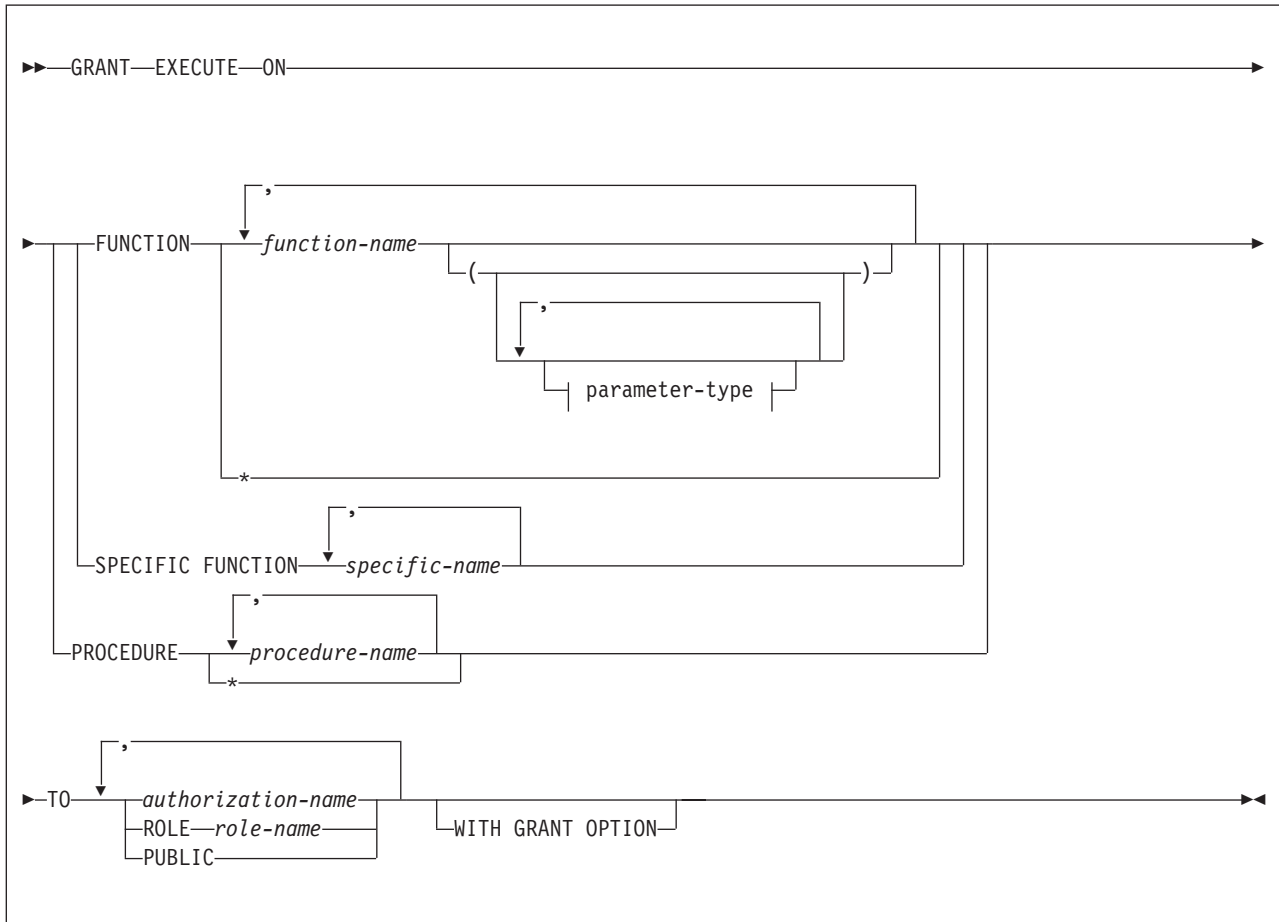
Example 4: Grant load privileges to database DSN9D91A to role ROLE1:

```
GRANT LOAD
  ON DATABASE DSN9D91A
  TO ROLE ROLE1;
```

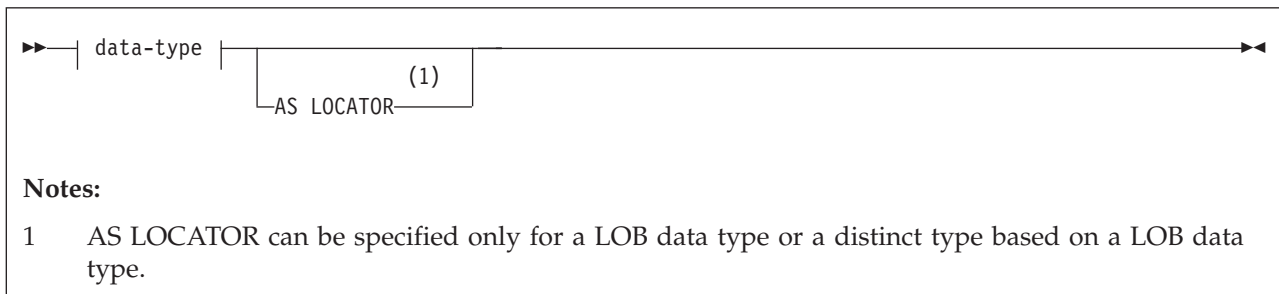
GRANT (function or procedure privileges)

This form of the GRANT statement grants privileges on user-defined functions, cast functions that are generated for distinct types, and stored procedures.

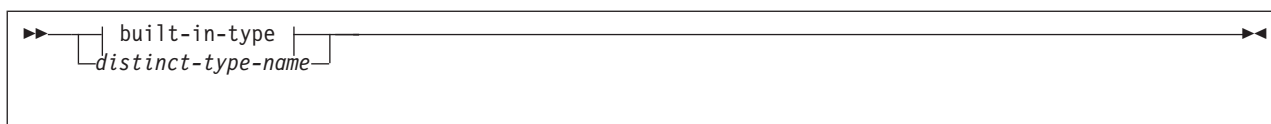
Syntax



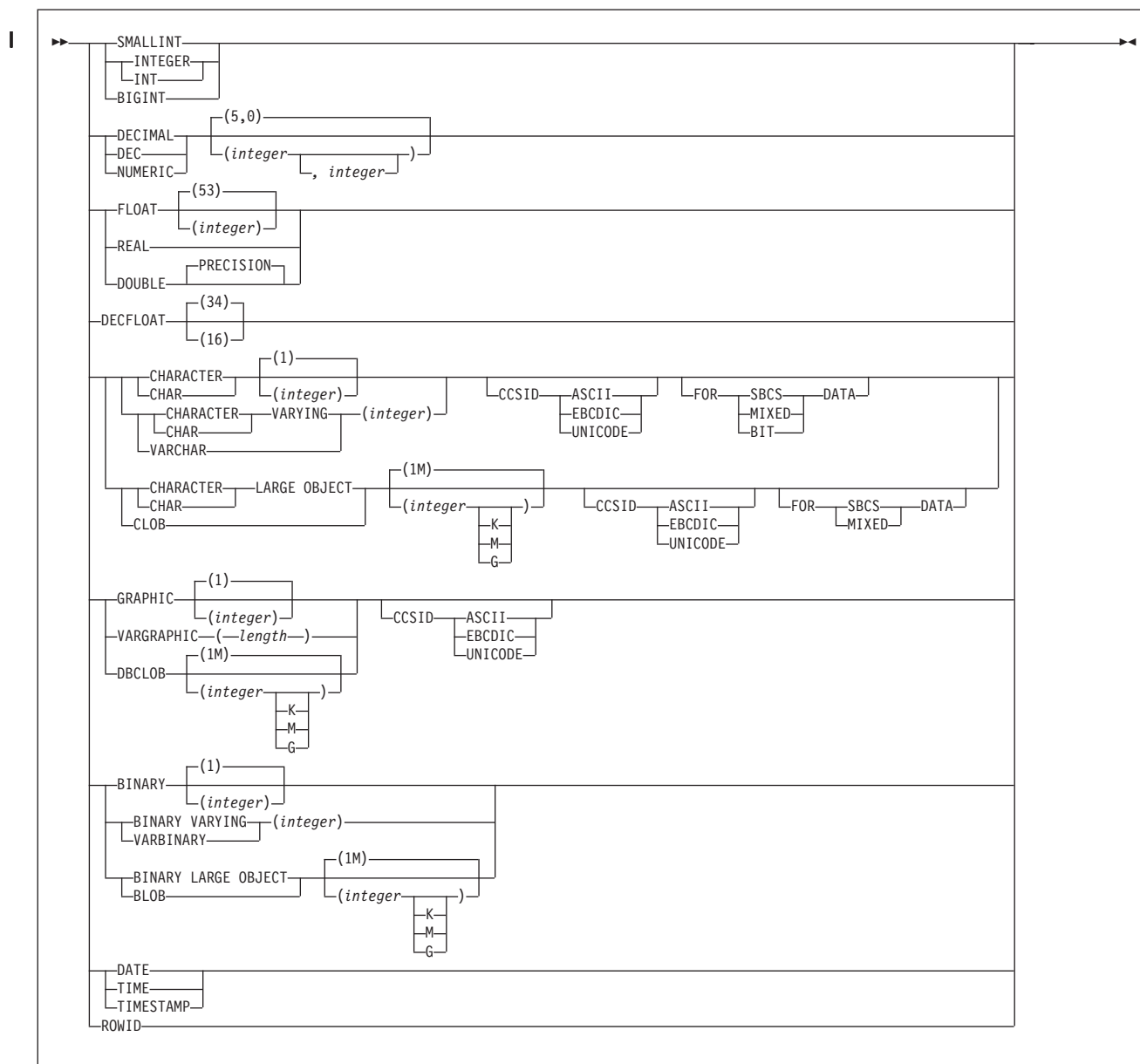
parameter-type:



data-type:



built-in-type:



Description

EXECUTE

Grants the privilege to run the identified user-defined function, cast function that was generated for a distinct type, or stored procedure.

FUNCTION or SPECIFIC FUNCTION

Identifies the function on which the privilege is granted. The function must exist at the current server, and it must be a function that was defined with the

CREATE FUNCTION statement or a cast function that was generated by a CREATE TYPE statement. The function can be identified by name, function signature, or specific name.

If the function was defined with a table parameter (the LIKE TABLE was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to identify the function. Instead, identify the function with its function name, if unique, or with its specific name.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function can have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

An * can be specified for a qualified or unqualified *function-name*. An * (or *schema-name.**) indicates that the privilege is granted on all the functions in the schema including those that do not currently exist. SYSADM authority is required if an * (or *schema-name.**) is specified. Specifying an * does not affect any EXECUTE privileges that are already granted on a function.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be granted. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function. If you do not explicitly qualify the function name with a schema name, the function name is implicitly qualified with a schema name as described in the preceding description for FUNCTION *function-name*.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly, DECFLOAT() will be considered a match for DECFLOAT(16) or

DECFLOAT(34). However, FLOAT cannot be specified with empty parentheses because its parameter value indicates a specific data type (REAL or DOUBLE).

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the FOR *subtype* DATA clause or the CCSID clause is optional. Omission of either clause indicates that DB2 ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE *procedure-name*

Identifies a stored procedure that is defined at the current server. The name, including the implicit or explicit schema name, must identify a stored procedure that exists at the current server.

An * can be specified for a qualified or unqualified *procedure-name*. An * (or *schema-name.**) indicates that the privilege is granted on all the stored procedures in the schema including those that do not currently exist. Specifying an * does not affect any EXECUTE privileges that are already granted on a stored procedure.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant the EXECUTE privilege on function CALC_SALARY to user JONES. Assume that there is only one function in the schema with function name CALC_SALARY.

```
GRANT EXECUTE ON FUNCTION CALC_SALARY TO JONES;
```

Example 2: Grant the EXECUTE privilege on procedure VACATION_ACCR to all users at the current server.

```
GRANT EXECUTE ON PROCEDURE VACATION_ACCR TO PUBLIC;
```

Example 3: Grant the EXECUTE privilege on function DEPT_TOTALS to the administrative assistant and give the assistant the ability to grant the EXECUTE privilege on this function to others. The function has the specific name DEPT85_TOT. Assume that the schema has more than one function that is named DEPT_TOTALS.

```
GRANT EXECUTE ON SPECIFIC FUNCTION DEPT85_TOT TO ADMIN_A
WITH GRANT OPTION;
```

Example 4: Grant the EXECUTE privilege on function NEW_DEPT_HIRES to HR (Human Resources). The function has two input parameters with data types of INTEGER and CHAR(10), respectively. Assume that the schema has more than one function that is named NEW_DEPT_HIRES.

```
GRANT EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10))
TO HR;
```

You can also code the CHAR(10) data type as CHAR().

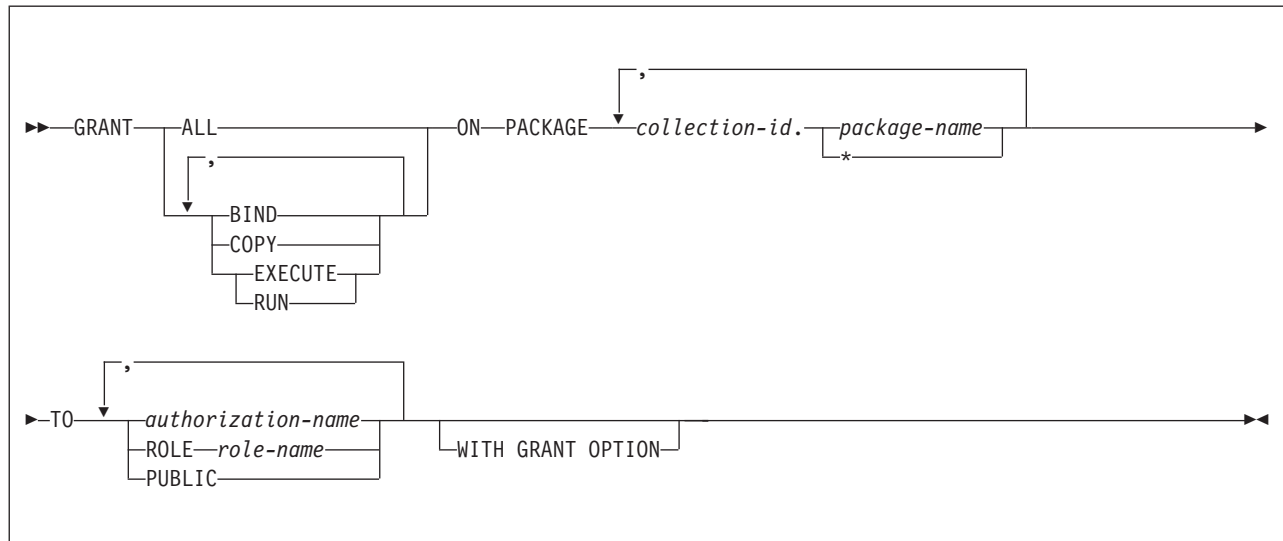
Example 5: Grant the EXECUTE privilege on function FIND_EMPDEPT to role ROLE1:

```
GRANT EXECUTE ON FUNCTION FIND_EMPDEPT TO ROLE ROLE1;
```

GRANT (package privileges)

This form of the GRANT statement grants privileges on packages.

Syntax



Description

BIND

Grants the privilege to use the **BIND** and **REBIND** subcommands for the designated packages.

The **BIND** package privilege can also be used to allow a user to add a new version of an existing package. For details on the authorization required to create new packages and new versions of existing packages, see “Notes” on page 1346.

COPY

Grants the privilege to use the **COPY** option of the **BIND** subcommand for the designated packages.

EXECUTE

Grants the privilege to run application programs that use the designated packages and to specify the packages following **PKLIST** for the **BIND PLAN** and **REBIND PLAN** commands. **RUN** is an alternate name for the same privilege.

ALL

Grants all package privileges for which you have **GRANT** authority for the packages named in the **ON** clause.

ON PACKAGE *collection-id.package-name,...*

Identifies packages for which you are granting privileges. The granting of a package privilege applies to all versions of a package. The list can simultaneously contain items of the following two forms:

- *collection-id.package-name* explicitly identifies a single package. The name must identify a package that exists at the current server.
- *collection-id.** applies to every package in the indicated collection. This includes packages that currently exist and future packages. The grant applies

to a collection at the current server, but the *collection-id* does not have to identify a collection that exists when the grant is made.

To grant a privilege in this form requires PACKADM with the WITH GRANT OPTION over the collection or all collections, SYSADM, or SYSCTRL authority. Because of this fact, WITH GRANT OPTION, if included in the statement, is ignored for grants of this form, but not for grants for specific packages.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Notes

The authorization required to add a new package or a new version of an existing package depends on the value of field BIND NEW PACKAGE on installation panel DSNTIPP. The default value is BINDADD.

If the value of BIND NEW PACKAGE is BINDADD, the owner must have one of the following to add a new package or a new version of an existing package to a collection:

- The BINDADD system privilege and either the CREATE IN privilege or PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority

If the value of BIND NEW PACKAGE is BIND, the owner must have one of the following to add a new package or a new version of an existing package to a collection:

- The BINDADD system privilege and either the CREATE IN privilege or PACKADM authority for the collection or for all collections
- SYSADM or SYSCTRL authority
- PACKADM authority for the collection or for all collections
- Users with the BIND package privilege can also add a new version of an existing package

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports specifying PROGRAM as a synonym for PACKAGE.

Examples

Example 1: Grant the privilege to copy all packages in collection DSN8CC61 to LEWIS.

```
GRANT COPY ON PACKAGE DSN8CC61.* TO LEWIS;
```

Example 2: You have the BIND privilege with GRANT authority over the package CLCT1.PKG1. You have the EXECUTE privilege with GRANT authority over the package CLCT2.PKG2. You have no other privileges with GRANT authority over any package in the collections CLCT1 AND CLCT2. Hence, the following statement, when executed by you, grants LEWIS the BIND privilege on CLCT1.PKG1 and the EXECUTE privilege on CLCT2.PKG2, and makes no other grant. The privileges granted include no GRANT authority.

```
GRANT ALL ON PACKAGE CLCT1.PKG1, CLCT2.PKG2 TO JONES;
```

| *Example 3:* Grant the privileges to run all packages in collection DSN9CC13 to role
| ROLE1:

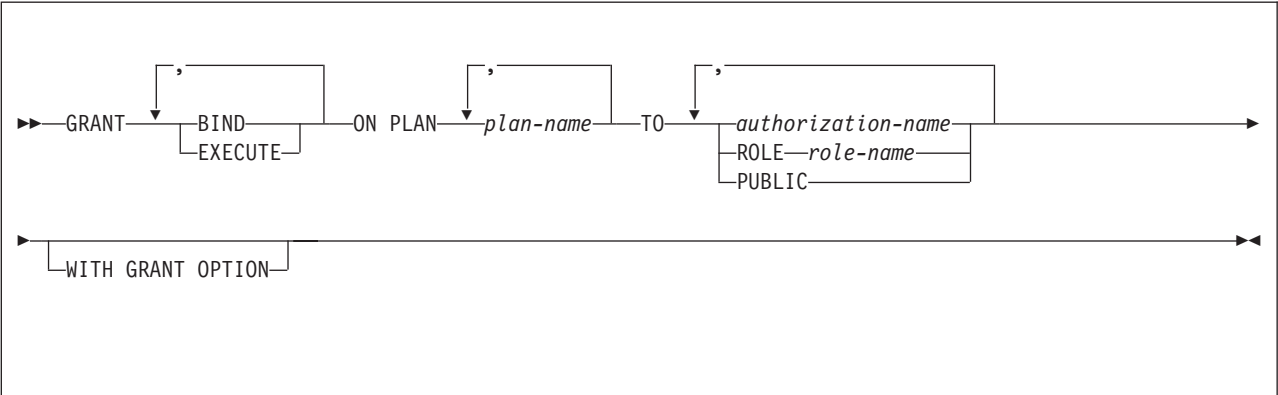
| GRANT EXECUTE ON PACKAGE DSN9CC13.* TO ROLE ROLE1;

|

GRANT (plan privileges)

This form of the GRANT statement grants privileges on plans.

Syntax



Description

BIND

Grants the privilege to use the BIND, REBIND, and FREE subcommands for the identified plans. (The authority to create new plans using BIND ADD is a system privilege.)

EXECUTE

Grants the privilege to run programs that use the identified plans.

ON PLAN *plan-name*,...

Identifies the application plans on which the privileges are granted. For each identified plan, you must have all specified privileges with the GRANT option.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant the privilege to bind plan DSN8IP91 to user JONES.

```
GRANT BIND ON PLAN DSN8IP91 TO JONES;
```

Example 2: Grant privileges to bind and execute plan DSN8CP91 to all users at the current server.

```
GRANT BIND,EXECUTE ON PLAN DSN8CP91 TO PUBLIC;
```

Example 3: Grant the privilege to execute plan DSN8CP91 to users ADAMSON and BROWN with grant option.

```
GRANT EXECUTE ON PLAN DSN8CP91 TO ADAMSON,BROWN WITH GRANT OPTION;
```

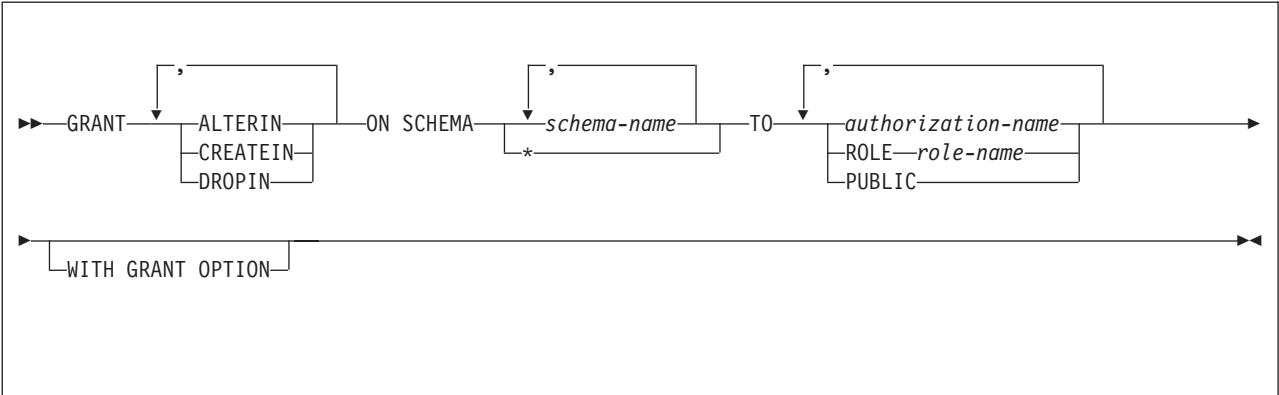
Example 4: Grant the privileges to bind the DSN91PLN plan to role ROLE1:

```
GRANT BIND ON PLAN DSN91PLN TO ROLE ROLE1;
```

GRANT (schema privileges)

This form of the GRANT statement grants privileges on schemas.

Syntax



Description

ALTERIN

Grants the privilege to alter stored procedures and user-defined functions, or specify a comment for distinct types, cast functions that are generated for distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

CREATEIN

Grants the privilege to create distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

DROPIN

Grants the privilege to drop distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

SCHEMA *schema-name*

Identifies the schemas on which the privilege is granted. The schemas do not need to exist when the privilege is granted.

SCHEMA *

Indicates that the specified privilege is granted on all schemas including those that do not currently exist.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant the CREATEIN privilege on schema T_SCORES to user JONES.

```
GRANT CREATEIN ON SCHEMA T_SCORES TO JONES;
```

Example 2: Grant the CREATEIN privilege on schema VAC to all users at the current server.

```
GRANT CREATEIN ON SCHEMA VAC TO PUBLIC;
```

Example 3: Grant the ALTERIN privilege on schema DEPT to the administrative assistant and give the grantee the ability to grant ALTERIN privileges on this schema to others.

```
GRANT ALTERIN ON SCHEMA DEPT TO ADMIN_A  
WITH GRANT OPTION;
```

Example 4: Grant the CREATEIN, ALTERIN, and DROPIN privileges on schemas NEW_HIRE, PROMO, and RESIGN to HR (Human Resources).

```
GRANT CREATEIN, ALTERIN, DROPIN ON SCHEMA NEW_HIRE, PROMO, RESIGN TO HR;
```

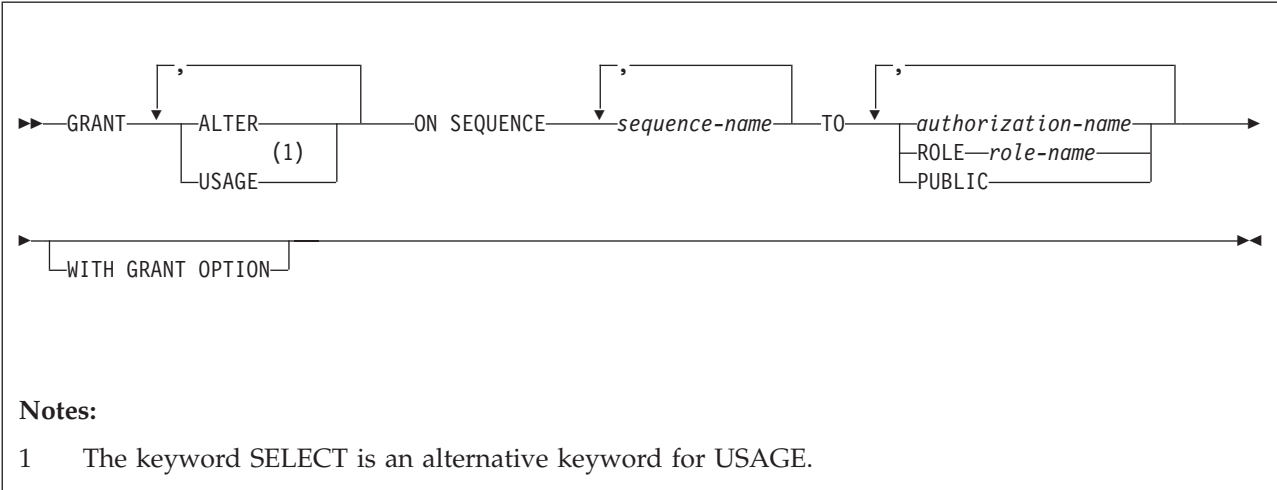
Example 5: Grant the ALTERIN privileges on the EMPLOYEE schema to role ROLE1:

```
GRANT ALTERIN ON SCHEMA EMPLOYEE TO ROLE ROLE1;
```


GRANT (sequence privileges)

This form of the GRANT statement grants privileges on a user-defined sequence.

Syntax



Description

ALTER

Grants the privilege to alter a sequence or record a comment on a sequence.

USAGE

Grants the **USAGE** privilege to use a sequence. This privilege is needed when the **NEXT VALUE** or **PREVIOUS VALUE** expression is invoked for a sequence name.

SEQUENCE *sequence-name*

Identifies the sequence. The name, including the implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error occurs. *sequence-name* must not be the name of an internal sequence object that is used by DB2.

TO Refer to “GRANT” on page 1333 for a description of the **TO** clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the **WITH GRANT OPTION** clause.

Examples

Example 1: Grant **USAGE** privilege on sequence **MYNUM** to user **JONES**.

```
GRANT USAGE
  ON SEQUENCE MYNUM
  TO JONES;
```

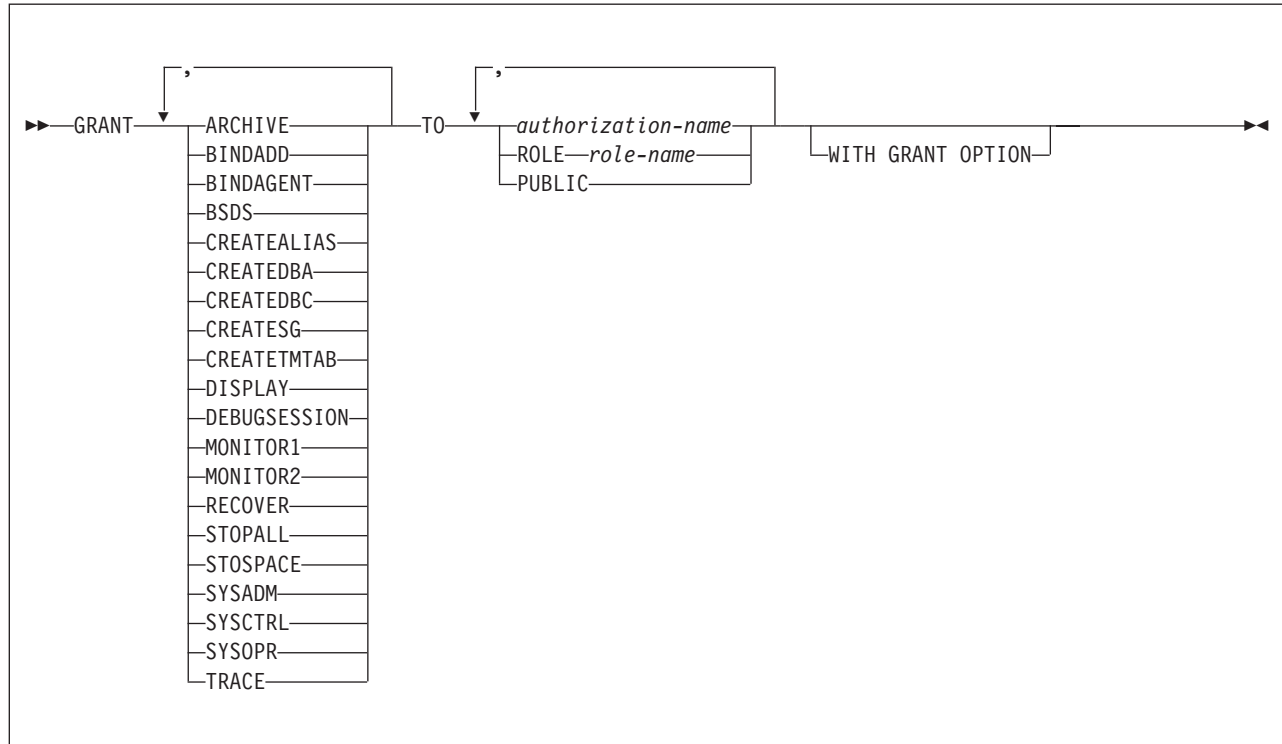
Example 2: Grant **USAGE** privileges on sequence **ORDER_SEQ** to role **ROLE1**:

```
GRANT USAGE ON SEQUENCE ORDER_SEQ TO ROLE ROLE1;
```

GRANT (system privileges)

This form of the GRANT statement grants system privileges.

Syntax



Description

ARCHIVE

Grants the privilege to use the ARCHIVE LOG and SET LOG commands.

BINDADD

Grants the privilege to create plans and packages by using the BIND subcommand with the ADD option.

BINDAGENT

Grants the privilege to issue the BIND, FREE PACKAGE, or REBIND subcommands for plans and packages and the DROP PACKAGE statement on behalf of the grantor. The privilege also allows the holder to copy and replace plans and packages on behalf of the grantor.

A warning is issued if WITH GRANT OPTION is specified when granting this privilege.

BSDS

Grants the privilege to issue the RECOVER BSDS command.

CREATEALIAS

Grants the privilege to use the CREATE ALIAS statement.

CREATEDBA

Grants the privilege to issue the CREATE DATABASE statement and acquire DBADM authority over those databases.

CREATEDBC

Grants the privilege to issue the CREATE DATABASE statement and acquire DBCTRL authority over those databases.

CREATESG

Grants the privilege to create new storage groups.

CREATETMTAB

Grants the privilege to use the CREATE GLOBAL TEMPORARY TABLE statement.

DISPLAY

Grants the privilege to use the following commands:

- The DISPLAY ARCHIVE command for archive log information
- The DISPLAY BUFFERPOOL command for the status of buffer pools
- The DISPLAY DATABASE command for the status of all databases
- The DISPLAY LOCATION command for statistics about threads with a distributed relationship
- The DISPLAY LOG command for log information, including the status of the offload task
- The DISPLAY THREAD command for information on active threads within DB2
- The DISPLAY TRACE command for a list of active traces

DEBUGSESSION

Grants the privilege to attach a debug client to the current application process connection, which enables client application debugging of native SQL or Java procedures that are executed within the session.

MONITOR1

Grants the privilege to obtain IFC data classified as serviceability data, statistics, accounting, and other performance data that does not contain potentially secure data.

MONITOR2

Grants the privilege to obtain IFC data classified as containing potentially sensitive data such as SQL statement text and audit data. Users with MONITOR2 privileges have MONITOR1 privileges.

RECOVER

Grants the privilege to issue the RECOVER INDOUBT command.

STOPALL

Grants the privilege to issue the STOP DB2 command.

STOSPACE

Grants the privilege to use the STOSPACE utility.

SYSADM

Grants all DB2 privileges except for a few reserved for installation SYSADM authority. The privileges the user possesses are all grantable, including the SYSADM authority itself. The privileges the user lacks restrict what the user can do with the directory and the catalog. Using WITH GRANT OPTION when granting SYSADM is redundant but valid. For more on SYSADM and install SYSADM authority, see *DB2 Administration Guide*.

SYSCTRL

Grants the system control authority, which allows the user to have most of the privileges of a system administrator but excludes the privileges to read or

change user data. Using WITH GRANT OPTION when granting SYSCTRL is redundant but valid. For more information on SYSCTRL authority, see *DB2 Administration Guide*.

SYSOPR

Grants the privilege to have system operator authority.

TRACE

Grants the privilege to issue the MODIFY TRACE, START TRACE, and STOP TRACE commands.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

If you grant the SYSADM or SYSCTRL system privilege, WITH GRANT OPTION is valid but unnecessary. It is unnecessary because whoever is granted SYSADM or SYSCTRL has that authority and all the privileges it implies, with the GRANT option.

Examples

Example 1: Grant DISPLAY privileges to user LUTZ.

```
GRANT DISPLAY  
  TO LUTZ;
```

Example 2: Grant BSDS and RECOVER privileges to users PARKER and SETRIGHT, with the WITH GRANT OPTION.

```
GRANT BSDS,RECOVER  
  TO PARKER,SETRIGHT  
  WITH GRANT OPTION;
```

Example 3: Grant TRACE privileges to all local users.

```
GRANT TRACE  
  TO PUBLIC;
```

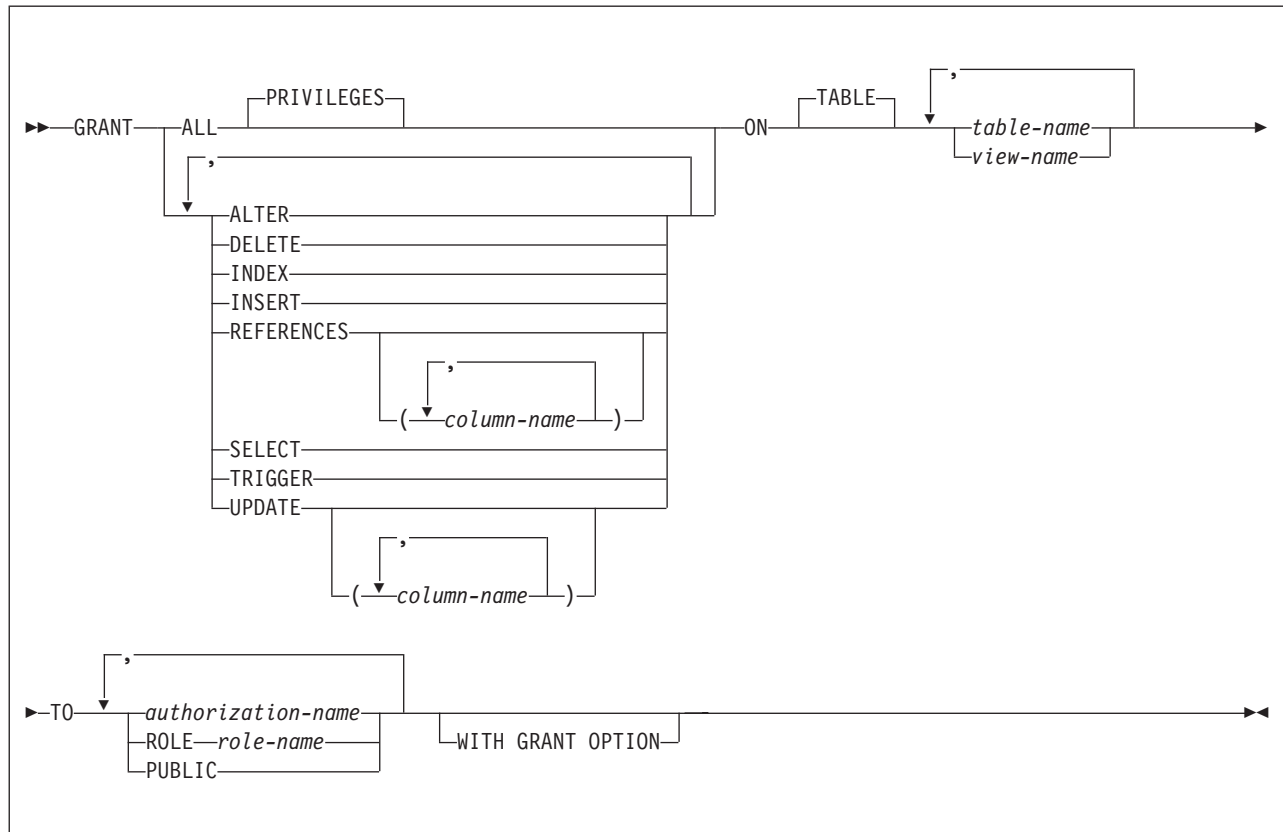
Example 4: Grant ARCHIVE privileges to role ROLE1:

```
GRANT ARCHIVE TO ROLE ROLE1;
```

GRANT (table or view privileges)

This form of the GRANT statement grants privileges on tables and views.

Syntax



Description

ALL or ALL PRIVILEGES

Grants all table or view privileges for which you have GRANT authority, for the tables and views named in the ON clause.

If you do not use **ALL**, you must use one or more of the keywords in the following list. For each keyword that you use, you must have GRANT authority for that privilege on every table or view identified in the ON clause.

ALTER

Grants the privilege to alter the specified table or create a trigger on the specified table. **ALTER** cannot be used if the statement identifies an auxiliary table or a view.

DELETE

Grants the privilege to delete rows in the specified table or view. **DELETE** cannot be granted on an auxiliary table.

INDEX

Grants the privilege to create an index on the specified table. **INDEX** cannot be granted on a view.

INSERT

Grants the privilege to insert rows into the specified table or view. INSERT cannot be granted on an auxiliary table.

REFERENCES

Grants the privilege to add a referential constraint in which the specified table is a parent. If a list of column names is not specified or if REFERENCES is granted via the specification of ALL PRIVILEGES, the grantee can define referential constraints using all columns of the table as a parent key, even those added later via the ALTER TABLE statement. This privilege cannot be granted on a view or auxiliary table.

REFERENCES(*column-name*,...)

Grants the privilege to add or drop a referential constraint in which the specified table is a parent using only those columns that are specified in the column list as a parent key. Each *column-name* must be an unqualified name that identifies a column of the table identified in the ON clause. This privilege cannot be granted on a view or auxiliary table.

SELECT

Grants the privilege to create a view or read data from the specified table or view. SELECT cannot be granted on an auxiliary table.

TRIGGER

Grants the privilege to create a trigger on the specified table. TRIGGER cannot be granted on an auxiliary table or a view.

UPDATE

Grants the privilege to update rows in the specified table or view. UPDATE cannot be granted on an auxiliary table.

UPDATE(*column-name*,...)

Grants the privilege to update only the columns named. Each *column-name* must be the unqualified name of a column of every table or view identified in the ON clause. Each *column-name* must not identify a column of an auxiliary table.

ON *table-name* or *view-name*

Specifies the tables or views on which you are granting the privileges. The list can be a list of table names or view names, or a combination of the two. A declared temporary table and a table that is implicitly created for an XML column must not be identified.

If you use GRANT ALL, then for each named table or view, the privilege set (described in ““Authorization” on page 1333” in “GRANT” on page 1333) must include at least one privilege with the GRANT option.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Notes

The REFERENCES privilege does not replace the ALTER privilege. It was added to conform to the SQL standard. To define a foreign key that references a parent table, you must have either the REFERENCES or the ALTER privilege, or both.

For a created temporary table or a view of a created temporary table, only ALL or ALL PRIVILEGES can be granted. Specific table or view privileges cannot be granted. In addition, only the ALTER, DELETE, INSERT, and SELECT privileges apply to a created temporary table.

To grant table privileges on a created temporary table, the privilege set must include one of the following:

- SYSADM
- DBADM on DSNDB06
- Ownership of the created temporary table

To grant table privileges on a view of a created temporary table, the privilege set must include one of the following:

- SYSADM
- ownership of the created temporary table

For a declared temporary table, no privileges can be granted. When a declared temporary table is defined, PUBLIC implicitly receives all table privileges (without GRANT authority) for the table. These privileges are not recorded in the DB2 catalog, and they cannot be revoked.

For an auxiliary table, only the INDEX privilege can be granted. DELETE, INSERT, SELECT, and UPDATE privileges on the base table that is associated with the auxiliary table extend to the auxiliary table.

PUBLIC AT ALL LOCATIONS: PUBLIC AT ALL LOCATIONS can be specified as an alternative to PUBLIC. PUBLIC AT ALL LOCATIONS is intended for use only with DB2 private protocol access.

The following privileges cannot be granted to PUBLIC AT ALL LOCATIONS:

- ALTER
- INDEX
- REFERENCES
- TRIGGER

In addition, ALL or ALL PRIVILEGES does not include ALTER, INDEX, REFERENCES, or TRIGGER privileges for a grant to PUBLIC AT ALL LOCATIONS.

Examples

Example 1: Grant SELECT privileges on table DSN8910.EMP to user PULASKI.

```
GRANT SELECT ON DSN8910.EMP TO PULASKI;
```

Example 2: Grant UPDATE privileges on columns EMPNO and WORKDEPT in table DSN8910.EMP to all users at the current server.

```
GRANT UPDATE (EMPNO,WORKDEPT) ON TABLE DSN8910.EMP TO PUBLIC;
```

Example 3: Grant all privileges on table DSN8910.EMP to users KWAN and THOMPSON, with the WITH GRANT OPTION.

```
GRANT ALL ON TABLE DSN8910.EMP TO KWAN,THOMPSON WITH GRANT OPTION;
```

Example 4: Grant the SELECT and UPDATE privileges on the table DSN8910.DEPT to every user in the network.

```
GRANT SELECT, UPDATE ON TABLE DSN8910.DEPT  
TO PUBLIC;
```

Even with this grant, it is possible that some network users do not have access to the table at all, or to any other object at the subsystem where the table exists. Controlling access to the subsystem involves the communications databases at the subsystems in the network. The tables for the communication databases are described in “DB2 catalog tables” on page 1677. Controlling access is described in *DB2 Administration Guide*.

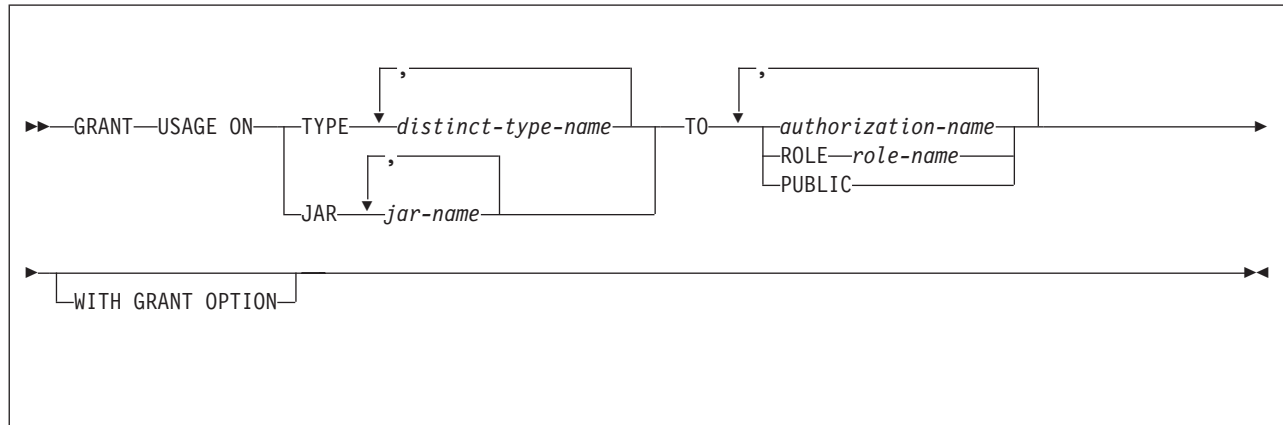
Example 5: Grant ALTER privileges on table DSN9910.EMP to role ROLE1:

```
GRANT ALTER ON TABLE DSN9910.EMP TO ROLE ROLE1;
```


GRANT (type or JAR file privileges)

This form of the GRANT statement grants the privilege to use distinct types (user-defined data types) or JAR files.

Syntax



Description

USAGE

Grants the privilege to use the distinct type in tables, functions procedures, or the privilege to use the JAR file.

TYPE *distinct-type-name*

Identifies the distinct type. The name, including the implicit or explicit schema name, must identify a unique distinct type that exists at the current server.

JAR *jar-name*

Identifies the JAR file. The name, including the implicit or explicit schema name, must identify a unique JAR file that exists at the current server.

T0 Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Notes

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports DATA TYPE or DISTINCT TYPE as a synonym for TYPE.

Examples

Example 1: Grant the USAGE privilege on distinct type SHOE_SIZE to user JONES. This GRANT statement does not give JONES the privilege to execute the cast functions that are associated with the distinct type SHOE_SIZE.

GRANT USAGE ON TYPE SHOE_SIZE TO JONES;

Example 2: Grant the USAGE privilege on distinct type US_DOLLAR to all users at the current server.

GRANT USAGE ON TYPE US DOLLAR TO PUBLIC;

Example 3: Grant the USAGE privilege on distinct type CANADIAN_DOLLAR to the administrative assistant (ADMIN_A), and give this user the ability to grant the USAGE privilege on the distinct type to others. The administrative assistant cannot grant the privilege to execute the cast functions that are associated with the distinct type CANADIAN_DOLLAR because WITH GRANT OPTION does not give the administrative assistant the EXECUTE authority on these cast functions.

```
GRANT USAGE ON TYPE CANADIAN_DOLLAR TO ADMIN_A
    WITH GRANT OPTION;
```

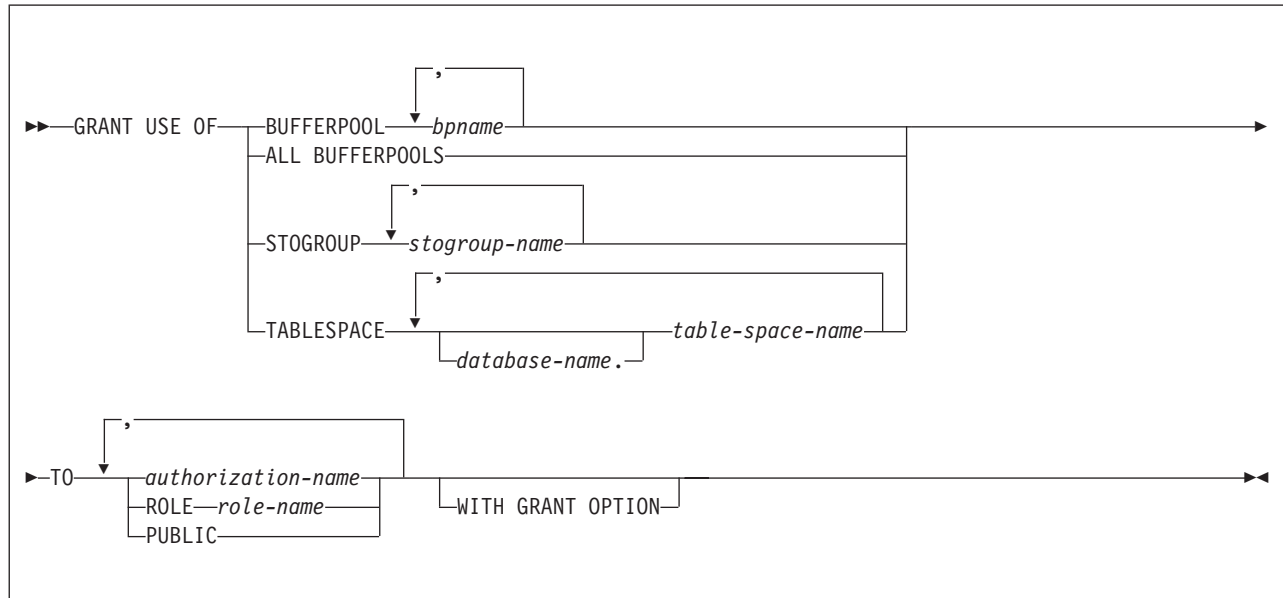
Example 4: Grant the USAGE privilege on the distinct type MILES to role ROLE1 at the current server:

```
GRANT USAGE ON TYPE MILES
    TO ROLE ROLE1;
```

GRANT (use privileges)

This form of the GRANT statement grants authority to use particular buffer pools, storage groups, or table spaces.

Syntax



Description

BUFFERPOOL *bpname*,...

Grants the privilege to refer to any of the identified buffer pools in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement. See “Naming conventions” on page 51 for more details about *bpname*.

ALL BUFFERPOOLS

Grants the privilege to refer to any buffer pool in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

STOGROUP *stogroup-name*,...

Grants the privilege to refer to any of the identified storage groups in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

TABLESPACE *database-name.table-space-name*,...

Grants the privilege to refer to any of the identified table spaces in a CREATE TABLE statement. The default for *database-name* is DSNDB04.

You cannot grant the privilege for table spaces that are for declared temporary tables (table spaces in a work file database). For these table spaces, PUBLIC implicitly has the TABLESPACE privilege (without GRANT authority); this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

TO Refer to “GRANT” on page 1333 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 1333 for a description of the WITH GRANT OPTION clause.

Notes

You can grant privileges for only one type of object with each statement. Thus, you can grant the use of several table spaces with one statement, but not the use of a table space and a storage group. For each object you identify, you must have the USE privilege with GRANT authority.

Examples

Example 1: Grant authority to use buffer pools BP1 and BP2 to user MARINO.

```
GRANT USE OF BUFFERPOOL BP1,BP2  
TO MARINO;
```

Example 2: Grant to all local users the authority to use table space DSN8S91D in database DSN8D91A.

```
GRANT USE OF TABLESPACE  
DSN8D91A.DSN8S91D  
TO PUBLIC;
```

Example 3: Grant authority to use storage group SG1 to role ROLE1:

```
GRANT USE OF STOGROUP SG1  
TO ROLE ROLE1;
```

HOLD LOCATOR

The HOLD LOCATOR statement allows a LOB locator variable to retain its association with a value beyond a unit of work.

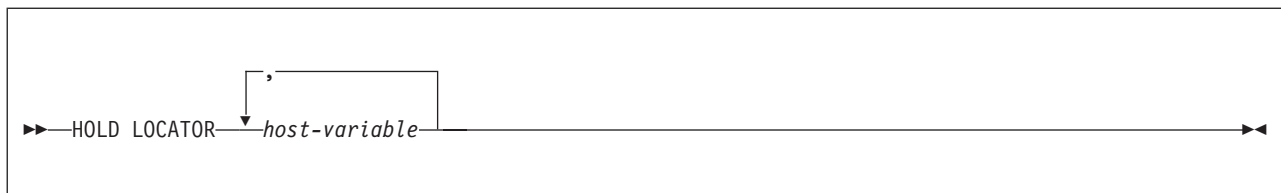
Invocation

This statement can only be embedded in an application program. It cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. HOLD LOCATOR cannot be used with the EXECUTE IMMEDIATE statement.

Authorization

None required.

Syntax



Description

host-variable, ...

Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables. The locator variable type must be a binary large object locator, a character large object locator, or a double-byte character large object locator.

The *host-variable* must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a FETCH, SELECT INTO, assignment statement, SET *host-variable* statement, or VALUES INTO statement); otherwise, an error is returned.

If more than one locator is specified and an error is returned on one of the locators, it is possible that some locators have been held and others have not been held.

Notes

A host-variable LOB locator variable that has the hold property is freed (has its association between it and its value removed) when:

- The SQL FREE LOCATOR statement is executed for the locator variable.
- The SQL ROLLBACK statement is executed.
- The SQL session is terminated.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the

values represented by the columns. Give the CLOB locator variables LOCRES and LOCHIST, and the BLOB locator variable LOCPIC the hold property.

```
EXEC SQL HOLD LOCATOR :LOCRES, :LOCHIST, :LOCPIC
```

INCLUDE

The INCLUDE statement inserts application code, including declarations and statements, into a source program.

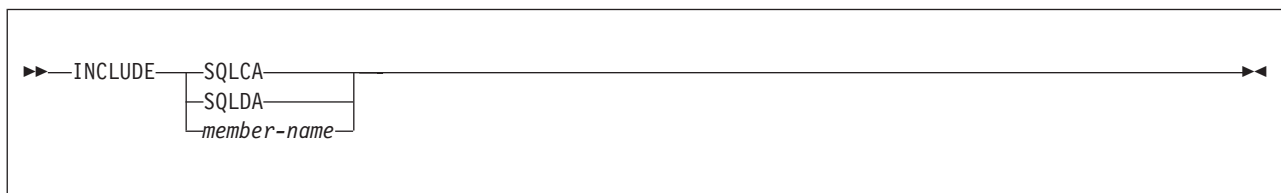
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

SQLCA

Indicates that the description of an SQL communication area (SQLCA) is to be included. INCLUDE SQLCA must not be specified more than once in the same application program. In COBOL, INCLUDE SQLCA must be specified in the Working-Storage Section or the Linkage Section. INCLUDE SQLCA must not be specified if the program is precompiled with the STDSQL(YES) option.

For a description of the SQLCA, see “SQL communication area (SQLCA)” on page 1646.

SQLDA

Indicates that the description of an SQL descriptor area (SQLDA) is to be included. It must not be specified in a Fortran. For a description of the SQLDA, see “SQL descriptor area (SQLDA)” on page 1656.

member-name

Names a member of the partitioned data set to be the library input when your application program is precompiled. It must be an SQL identifier.

The member can contain any host language source statements and any SQL statements other than an INCLUDE statement. In COBOL, INCLUDE *member-name* must not be specified in other than the Data Division or the Procedure Division.

Notes

When your application program is precompiled, the INCLUDE statement is replaced by source statements. Thus, the INCLUDE statement must be specified at a point in your application program where the resulting source statements are acceptable to the compiler.

The INCLUDE statement cannot refer to source statements that themselves contain INCLUDE statements.

The declarations that are generated by DCLGEN can be used in an application program by specifying the same member in the INCLUDE statement as in the DCLGEN LIBRARY parameter.

Example

Include an SQL communications area in a PL/I program.

```
EXEC SQL INCLUDE SQLCA;
```

INSERT

The INSERT statement inserts rows into a table or view or activates the INSTEAD OF INSERT trigger. The table or view can be at the current server or any DB2 subsystem with which the current server can establish a connection. Inserting a row into a view inserts the row into the table on which the view is based if no INSTEAD OF INSERT trigger is defined on the specified view. If an INSTEAD OF INSERT trigger is defined, the trigger is activated instead of the INSERT statement.

There are three forms of this statement:

- The INSERT via VALUES form is used to insert a single row into the table or view using the values provided or referenced.
- The INSERT via SELECT form is used to insert one or more rows into the table or view using values from other tables, or views, or both.
- The INSERT via FOR *n* ROWS form is used to insert multiple rows into the table or view using values provided or referenced. Although not required, the values can come from *host-variable arrays*. This form of INSERT is supported in SQL procedure applications. However, since host-variable arrays are not supported in SQL procedure applications, the support is limited to insertion of scalar values.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

Authority requirements depend on whether the object identified in the statement is a user-defined table, a catalog table for which inserts are allowed, or a view:

When a user-defined table is identified: The privilege set must include at least one of the following:

- The INSERT privilege on the table
- Ownership of the table
- DBADM authority on the database that contains the table
- SYSADM authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

When a catalog table is identified: The privilege set must include at least one of the following:

- DBADM authority on the catalog database
- SYSCtrl authority
- SYSADM authority

When a view is identified: The privilege set must include at least one of the following:

- The INSERT privilege on the view
- SYSADM authority

The owner of a view, unlike the owner of a table, might not have INSERT authority on the view (or can have INSERT authority without being able to grant it

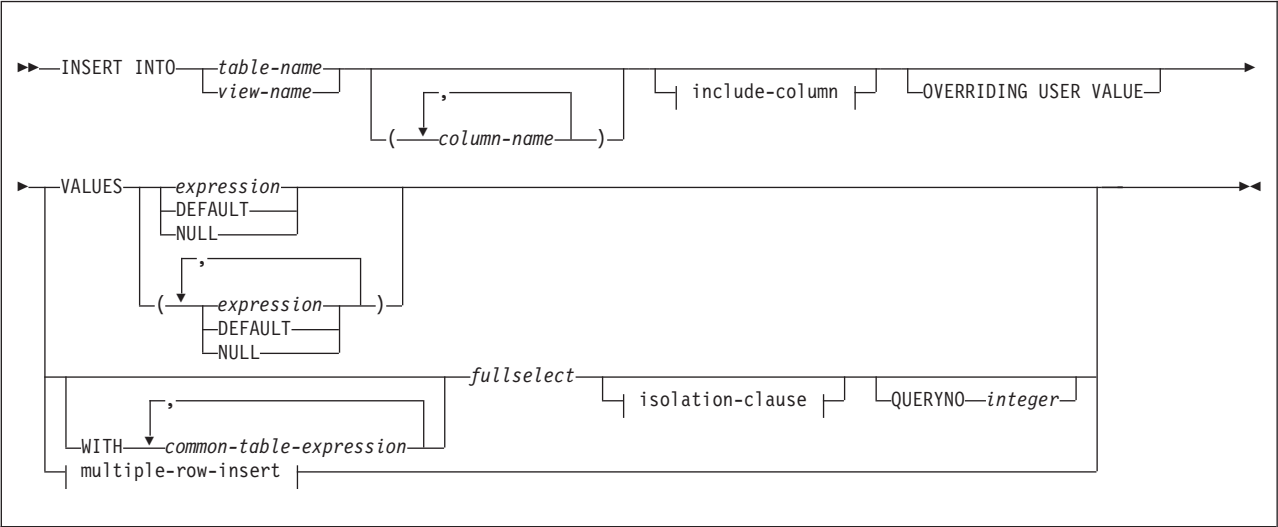
to others). The nature of the view itself can preclude its use for INSERT. For more information, see the discussion of authority in “CREATE VIEW” on page 1184.

If the INSERT statement is embedded in a SELECT statement, the privilege set must include the SELECT privilege on the table or view.

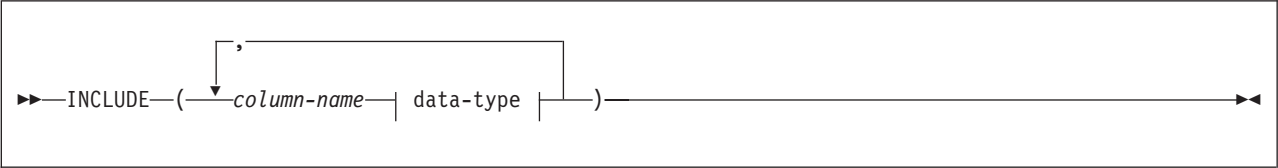
If a fullselect is specified, the privilege set must include authority to execute the fullselect. For more information about the authorization rules, see “Authorization” on page 630.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)

Syntax



include-column:



data-type:

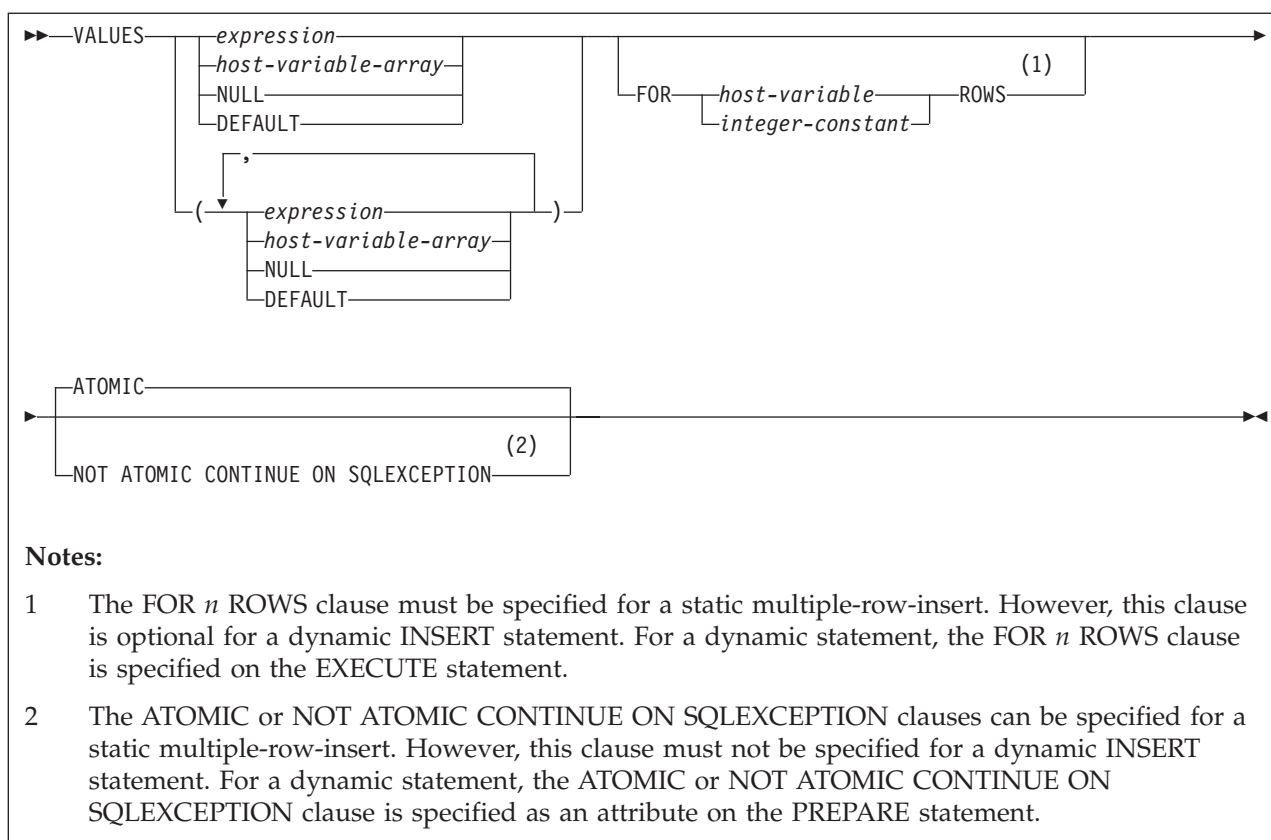


1



7





Description

INTO *table-name* or *view-name*

Identifies the object of the INSERT statement. The name must identify a table or view that exists at the current server. The name must not identify:

- An auxiliary table
- A catalog table
- A read-only view unless an instead of trigger is defined for the insert operation on the view. (For a description of a read-only view, see "CREATE VIEW" on page 1184.)
- A view column that is derived from a constant, expression, or scalar function
- A view column that is derived from the base table column as some other column of the view
- A materialized query table
- A table that is implicitly created for an XML column

In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must be a remote server that supports two-phase commit.

column-name,...

Specifies the columns for which insert values are provided. Each name must identify a column of the table or view. The columns can be identified in any order, but the same column must not be identified more than one time. A view column that cannot accept insert values must not be identified. If the object of the INSERT statement is a view with such columns, a list of column names must be specified, and the list must not identify these columns. If a qualifier is specified, it must be valid (that is, the table name must be the table or view

name specified after the INTO keyword, and if a qualifier is specified for the table name, it must match the default qualifier).

Omission of the column list is an implicit specification of a list in which every column of the table (that is not defined as implicitly hidden) or view is identified in left-to-right order. This list is established when the statement is prepared and therefore does not include columns that were added to the table after the statement was prepared.

The effect of a rebind on INSERT statements that do not include a column list is that the implicit list of names is re-established. Therefore, the number of columns into which data is inserted can change and cause an error.

include-column

Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the result table of the INSERT statement when it is nested in the FROM clause of the outer fullselect that is used in a subselect, a SELECT statement, or in a SELECT INTO statement. The included columns are appended to the end of the list of columns that is identified by *table-name* or *view-name*.

INCLUDE

Introduces a list of columns that is to be included in the result table of the INSERT statement. The included columns are only available if the INSERT statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

column-name

Specifies the name for a column of the result table of the INSERT statement that is not the same name as another included column nor a column in the table or view that is specified in *table-name* or *view-name*.

data-type

Specifies the data type of the included column. The included columns are nullable.

built-in-type

Specifies a built-in data type. See “CREATE TABLE” on page 1079 for a description of each built-in type.

distinct-type

Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

OVERRIDING USER VALUE

Specifies that the value specified in the VALUES clause or produced by a fullselect for a column that is defined as either GENERATED ALWAYS or GENERATED BY DEFAULT is ignored. Instead, a system-generated value is inserted, overriding the user-specified value.

If OVERRIDING USER VALUE is specified, the implicit or explicit list of column must include a column that is defined as either GENERATED ALWAYS or GENERATED BY DEFAULT. For example, a ROWID column, an identity column, or a row change timestamp column.

VALUES

Specifies one new row in the form of a list of values. The number of values in the VALUES clause must be equal to the number of names in the column list and the columns that are identified in the INCLUDE clause. The first value is inserted in the first column in the list, the second value in the second column,

and so on. If more than one value is specified, the list of values must be enclosed in parentheses. Assignments to included columns are only processed when the INSERT statement is nested in the FROM clause in a SELECT statement or a SELECT INTO statement.

expression

Any expression of the type described in “Expressions” on page 180. The expression must not include a column name. If *expression* is a host variable, the host variable can identify a structure. Any host variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables.

DEFAULT

The default value that is assigned to the column. If the column is a ROWID column or an identity column, DB2 will generate a value for the column. You can specify DEFAULT only for columns that have an assigned default value, ROWID columns, and identity columns.

For information on default values of data types, see the description of the DEFAULT clause for “CREATE TABLE” on page 1079.

NULL

Specifies the null value as the value of the column. Specify NULL only for nullable columns.

If the implicit or explicit list of columns includes a ROWID, an identity column, or a row change timestamp column that was defined as GENERATED ALWAYS, you must specify DEFAULT unless you specify the OVERRIDING USER VALUE clause to indicate that any user-specified value will be ignored and a unique system-generated value will be inserted.

For a ROWID or identity column that is defined as GENERATED BY DEFAULT, you can specify a value. However, a value can be inserted into ROWID column defined BY DEFAULT only if a single-column unique index is defined on the ROWID column and the specified value is a valid row ID value that was previously generated by DB2. When a value is inserted into an identity column defined BY DEFAULT, DB2 does not verify that the specified value is a unique value for the column unless the identity column has a single-column unique index.

Although an implicitly hidden DOCID column for XML values is defined as GENERATED ALWAYS, you can include the DOCID column in the explicit list of columns and specify a value for it. However, DB2 will ignore the value.

WITH *common-table-expression*

Specifies a common table expression. For an explanation of common table expression, see “common-table-expression” on page 670.

fullselect

Specifies a set of new rows in the form of the result table of a fullselect. If the result table is empty, SQLCODE is set to +100, and SQLSTATE is set to '02000'.

When the base object of the INSERT and the base object of the fullselect or any subquery of the fullselect are the same table, the fullselect is evaluated completely before any rows are inserted.

For an explanation of fullselect, see “fullselect” on page 662.

The number of columns in the result table must be equal to the number of names in the column list and the columns that are identified in the INCLUDE clause. The value of the first column of the result is inserted in the first column

in the list, the second value in the second column, and so on. Any values that are produced for a generated column must conform to the rules that are described for those columns under the VALUES clause. Assignments to included columns are only processed when the INSERT statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

If the object table is self-referencing, the fullselect must not return more than one row.

isolation-clause

Specifies the isolation level that is used when the fullselect is executed.

WITH

Introduces the isolation level, which can be one of the following:

RR	Repeatable read
RS	Read stability
CS	Cursor stability

The default isolation level of the statement is the isolation level of the package or plan in which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

QUERYNO *integer*

Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:

- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

multiple-row-insert

VALUES

Specifies the items for the rows to be inserted. The number of items in the VALUES clause must equal the number of names in the implicit or explicit column list. The first item in the list provides the value (or values) for the first column in the list. The second item in the list provides the value (or values) for the second column, and so on.

expression

Any expression of the type described in “Expressions” on page 180. The expression must not include a column name. For each row that is inserted, the corresponding column is assigned the value of the expression.

host-variable-array

Each host-variable array must be defined in the application program in

accordance with the rules for declaring an array. A host-variable array contains the data for a column of table that is a target of the INSERT. The number of rows to be inserted must be less than or equal to the dimension of each of the host-variable arrays.

An optional indicator array can be specified for each host-variable array. It should be specified if the SQLTYPE of any SQLVAR occurrence indicates that the SQLVAR is nullable. The indicators must be small integers. The indicator array must be large enough to contain an indicator for each row of input data.

DEFAULT

Specifies that the default value is assigned to the column. For each row inserted, the corresponding column is assigned its default value. DEFAULT can be specified only for columns that have a default value. For information on default values of data types, see the description of the DEFAULT clause for "CREATE TABLE" on page 1079.

NULL

Specifies the null value as the value of the column in each row inserted. For each row inserted, the corresponding column is assigned the NULL value. Specify NULL only for nullable columns.

FOR *host-variable* or *integer-constant* ROWS

Specifies the number of rows to be inserted. For a dynamic INSERT statement, this clause can be specified on the EXECUTE statement. For more information, see the EXECUTE statement. However, this clause is required when a dynamic SELECT statement contains more than one multiple-row INSERT statement.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, and must not include an indicator variable. Furthermore, *k* must be in the range, $0 < k \leq 32767$. *k* rows are inserted into the target table from the specified source data.

If a parameter marker is specified in this clause, a value must be provided with the USING clause of the associated EXECUTE or OPEN statement.

ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION

Specifies whether all of the rows should be inserted as an atomic operation or not.

ATOMIC

Specifies that if the insert for any row fails, all changes made to the database by any of the inserts, including changes made by successful inserts, are undone. This is the default.

NOT ATOMIC CONTINUE ON SQLEXCEPTION

Specifies that, regardless of the failure of any particular insert of a row, the INSERT statement will not undo any changes made to the database by the successful inserts of other rows, and inserting will be attempted for subsequent rows. However, the minimum level of atomicity is at least that of a single insert (that is, it is not possible for a partial insert to complete), including any triggers that might have been executed as a result of the INSERT statement.

This clause is only valid for a static INSERT statement. This clause must also not be specified if the INSERT statement is contained within a SELECT statement. For a dynamic INSERT statement, specify the clause on the PREPARE statement. For more information, see "PREPARE" on page 1405.

Notes

Insert rules: Insert values must satisfy the following rules. If they do not, or if any other errors occur during the execution of the INSERT statement, no rows are inserted and the position of the cursors are not changed.

- *Default values.* The value inserted in any column that is not in the column list is the default value of the column. Columns without a default value must be included in the column list. Similarly, if you insert into a view, the default value is inserted into any column of the base table that is not included in the view. Hence, all columns of the base table that are not in the view must have a default value.
- *Length.* If the insert value of a column is a number, the column must be a numeric column with the capacity to represent the integral part of the number. If the insert value of a column is a string, the column must be either a string column with a length attribute at least as great as the length of the string, or a datetime column if the string represents a date, time, or timestamp.
- *Assignment.* Insert values are assigned to columns in accordance with the assignment rules described in Chapter 2, “Language elements,” on page 47.
- *Uniqueness constraints.* If the identified table or the base table of the identified view has one or more unique indexes, each row inserted into the table must conform to the constraints imposed by those indexes.
- *Referential constraints.* Each nonnull insert value of a foreign key must be equal to some value of the parent key of the parent table in the relationship.
- *Check constraints.* The identified table or the base table of the identified view might have one or more check constraints. Each row inserted must conform to the conditions imposed by those constraints. Thus, each check condition must be true or unknown.
- *Field and validation procedures.* If the identified table or the base table of the identified view has a field or validation procedure, each row inserted must conform to the constraints imposed by that procedure.
- *Indexes with VARBINARY columns.* If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.
- *Views and the WITH CHECK OPTION.* For views defined with WITH CHECK OPTION, each row you insert into the view must conform to the definition of the view. If the view you name is dependent on other views whose definitions include WITH CHECK OPTION, the inserted rows must also conform to the definitions of those views. For an explanation of the rules governing this situation, see “CREATE VIEW” on page 1184.
For views that are not defined with WITH CHECK OPTION, you can insert rows that do not conform to the definition of the view. Those rows cannot appear in the view but are inserted into the base table of the view.
- *Omitting the column list.* When you omit the column list, you must specify a value for every column that was present in the table when the INSERT statement was bound or (for dynamic execution) prepared.
- *Triggers.* An INSERT statement might cause triggers to be activated. A trigger might cause other statements to be executed or raise error conditions based on the insert values. If an INSERT statement for a view activates an INSTEAD OF trigger, the validity, referential integrity, and check constraints are checked

against the data changes that are performed in the trigger, and not against the definition of the view that activates the trigger or the definition of the underlying tables or views.

When triggers are processed for an INSERT statement that inserts multiple rows depends on the atomicity option that is in effect for the INSERT statement:

- **ATOMIC.** The inserts are processed as a single statement. Any statement level triggers are activated one time for the statement, and the transition tables will include all of the rows that were inserted.
- **NOT ATOMIC CONTINUE ON SQLEXCEPTION.** The inserts are processed separately. Any statement level triggers are processed for each row that is inserted, and the transition table includes the individual row that is inserted. When errors are encountered with this option in effect, processing continues, and some of the specified rows will not be inserted. In this case, if an insert trigger is defined on the underlying base table, the statement level triggers will only be activated for rows that were successfully inserted.

Regardless of the failure of any particular source row, the INSERT statement will not undo any changes that are made to the database by the statement. Insert will be attempted for rows that follow the failed row. However, the minimum level of atomicity is at least that of a single source row (that is, it is not possible for a partial insert operation to complete), including any triggers that might have been activated as a result of the INSERT statement.

Inserting XML documents: When XML documents are inserted into a table that contains an XML index, the XML values that are inserted into the index are cast to the data type that is specified on the CREATE INDEX statement. If the XML value cannot be cast to the specified data type, the XML value is ignored for the XML index but the document is still inserted into the table. If the data type that is specified for casting is DECFLOAT, values can be rounded when they are inserted into the index. If the index is unique, the rounding that happens during the cast can result in duplicate values.

Number of rows inserted: Normally, after an INSERT statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows inserted. (For a complete description of the SQLCA, including exceptions to the above statement, see “SQL communication area (SQLCA)” on page 1646.) The value in SQLERRD(3) does not include the number of rows that were inserted as the result of a trigger.

Nesting user-defined functions or stored procedures: An INSERT statement can implicitly or explicitly refer to user-defined functions or stored procedures. This is known as *nesting* of SQL statements. A user-defined function or stored procedure that is nested within the INSERT must not access the table into which you are inserting values.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired at the execution of a successful insert operation. Until a commit or rollback operation releases the locks, only the application process that performed the insert can access the inserted row. If LOBs are not inserted into the row, application processes that are running with uncommitted read can also access the inserted row. The locks can also prevent other application processes from performing operations on the table. However, application processes that are running with uncommitted read can access locked pages and rows.

Locks are not acquired on declared temporary tables.

Inserting rows into a table with multilevel security : When you insert rows into a table with multilevel security, DB2 determines the value for the security label column of the row according to the following rules:

- If the user (the primary authorization ID) has write-down privilege or write-down control is not enabled, the user can set the security label for the row to any valid security label. The value that is specified must be assignable to a column that is defined as CHAR(8) FOR SBCS DATA NOT NULL. If the user does not specify a value for the security label or specifies DEFAULT, the security label of the row becomes the same as the security label of the user.
- If the user does not have write-down privilege and write-down control is enabled, the security label of the row becomes the same as the security label of the user.

Inserting rows into a table with a row change timestamp column: A row change timestamp column that is defined as GENERATED ALWAYS should not be specified in an INSERT statement, unless the OVERRIDING USER VALUE clause is specified to indicate that any specified value is ignored and a unique system-generated value is inserted.

Considerations for an INSERT without a column list: An INSERT statement without a column list does not include implicitly hidden columns, so columns that are defined as implicitly hidden must have a defined default value.

Inserting a row into catalog table SYSIBM.SYSSTRINGS: If the object table is SYSIBM.SYSSTRINGS, only certain values can be specified, as described in *DB2 Administration Guide*.

Datetime representation when using datetime registers: As explained under Datetime special registers, when two or more datetime registers are implicitly or explicitly specified in a single SQL statement, they represent the same point in time. This is also true when multiple rows are inserted. When ATOMIC is in effect for the INSERT statement, the special registers are evaluated one time for the processing of the statement. If NOT ATOMIC is in effect, the special registers are evaluated as each row of source data is processed.

Considerations for non-atomic processing of an INSERT statement: When NOT ATOMIC is specified the rows of source data are processed separately. Any references to special registers, sequence expressions, and functions in the INSERT statement are evaluated as each row of source data is processed, Statement level triggers are activated as each row of source data is processed.

If one or more errors occur during the execution of an insert of a row, processing continues. The row that was being inserted at the time of the error is not inserted. Execution continues with the next row to be inserted, and any other changes made during the execution of the multiple-row INSERT statement are not backed out. However, the insert of an individual row is an atomic action.

Diagnostics information for a multiple-row INSERT statement: A single multiple-row INSERT statement might encounter multiple conditions. These conditions can be errors or warnings. Use the GET DIAGNOSTICS statement to obtain information about all of the conditions that are encountered for one of these INSERT statements. See “GET DIAGNOSTICS” on page 1317 for more information.

If a warning occurs during the execution of an insert of a row, processing continues.

When multiple errors or warnings occur with a non-atomic INSERT statement, diagnostic information for each row is available using the GET DIAGNOSTICS statement. The SQLSTATE and SQLCODE reflect a summary of what happened during the INSERT statement:

- **SQLSTATE 01659, SQLCODE +252.** All rows were inserted, but one or more warnings occurred.
- **SQLSTATE 22529, SQLCODE -253.** At least one row was successfully inserted, but one or more errors occurred. Some warnings might also have occurred.
- **SQLSTATE 22530, SQLCODE -254.** No row was inserted. One or more errors occurred while trying to insert multiple rows of data.
- **SQLSTATE 429BI, SQLCODE -20252.** More errors occurred that DB2 is capable of recording. Statement processing is terminated.

When ATOMIC is in effect, if an insert value violates any constraints or if any other error occurs during the execution of an insert of a row, all changes made during the execution of the multiple-row INSERT statement are backed out. The SQLCA reflects the last warning encountered.

After an INSERT statement that inserts multiple rows of data, both atomic and non-atomic, information is returned to the program through the SQLCA. The SQLCA is set as follows:

- SQLCODE contains the SQLCODE.
- SQLSTATE contains the SQLSTATE.
- SQLERRD3 contains the number of rows actually inserted. SQLERRD3 is the number of rows inserted, if this is less than the number of rows requested, then an error occurred.
- SQLWARN flags are set if they were set during any single insert operation.

The SQLCA is used to return information on errors and warnings found during a multiple-row insert. If indicator arrays are provided, the indicator variable values are used to determine if the value from the host-variable array, or NULL, will be used. The SQLSTATE contains the warning from the last data mapping error.

Considerations for specifying the number of rows for a dynamic multiple-row

INSERT statement: Be aware of these considerations when specifying the number of rows to be inserted with a dynamic multiple-row INSERT statement that uses host-variable arrays:

- The FOR n ROWS clause can be specified as part of an INSERT statement or as part of an EXECUTE statement, but not both
- In the INSERT statement, you can specify a numeric constant in the FOR n ROWS clause to indicate the number of rows to be inserted or specify a parameter marker to indicate that the number of rows will be specified with the associated EXECUTE or OPEN statement. A multiple-row INSERT statement that is contained within a SELECT statement must include a FOR n ROWS clause.
- In an EXECUTE statement, when a dynamic INSERT statement is not contained within a SELECT statement, the number of rows can be specified with either the FOR n ROWS clause or the USING clause of the EXECUTE statement:
 - If the INSERT statement did not contain a FOR n ROWS clause, a value for the number of rows to be inserted can be specified in the FOR n ROWS clause of the EXECUTE statement with a numeric constant or host variable.
 - If a parameter marker was specified as part of a FOR n ROWS clause in the INSERT statement, a value for the number of rows must be specified with the USING clause of the EXECUTE statement.

- In an OPEN statement, when a dynamic SELECT statement contains one or more INSERT statements that have FOR *n* ROWS clauses with parameter markers, the values for the number of rows to be inserted (that is, the values for the parameter markers) must be specified with the USING clause of the OPEN statement.

DRDA considerations for a multiple-row INSERT statement: DB2 for z/OS limits the size of user data and control information to 10M (except for LOBs, which are processed in a different data stream) for a single multiple-row INSERT statement using host-variable arrays.

Multiple-row insert and fetch statements are supported by any requester or server that supports the DRDA Version 3 protocols. If an attempt is made to issue a multiple-row INSERT or FETCH statement on a server that does not support DRDA Version 3 protocols, an error occurs.

When a multiple-row INSERT statement is executed at a DB2 for z/OS requester, the number of rows being inserted at the requester might not be known in some cases. These cases include:

- The FOR *n* ROWS clause contains a constant value for *n* for either a static or dynamic INSERT statement.
- Host variables are specified on the USING clause of an EXECUTE statement for a dynamic INSERT statement.

In either case, if the number of rows that is being inserted is not known, the requester might flow more data than is required to the server. The number of rows that is actually inserted will be correct because the server knows the correct number of rows to insert. However, performance can be adversely affected. Consider the following scenario:

```
...
long serial_num [10];
struct {
short len;
char data [18];
}name [20]
...
EXEC SQL INSERT INTO T1 VALUES (:serial_num, :name) FOR 5 ROWS
```

At the requester, when this statement is executed, the number of rows being inserted, 5, is not known. As a result, the requester will flow 10 values for serial_num and 10 values for name to the server (because the maximum number of rows that can be inserted without error is 10, which is the size of the smallest host-variable array).

Use the following programming techniques to avoid or minimize problems:

- Avoid using constant values for *n* in the FOR *n* ROWS clause of INSERT statements. For static INSERT statements, this technique ensures that the value for *n* will be known at the requester.
- For dynamic INSERT statements, use the USING DESCRIPTOR clause instead of the USING *host-variables* clause on the EXECUTE statement. If a USING DESCRIPTOR clause is used on the EXECUTE statement, the value for 'n' must be indicated in the DESCRIPTOR.
- If neither of the above methods can be used:

- Declare your host-variable arrays as small as possible, or indicate that the size of your host-variable arrays are the size of 'n' in your descriptor. This avoids sending large numbers of host-variable-array entries that will not be used to the server.
- Ensure that varying length string arrays are initialized to a length of 0 (zero). This minimizes the amount of data that is sent to the server.
- Ensure that decimal host-variable arrays are initialized to valid values. This avoids a negative SQLCODE from being returned if the requester encounters invalid decimal data.

Other SQL statements in the same unit of work: The following statements cannot follow an INSERT statement in the same unit of work:

- An ALTER TABLE statement that changes the data type of a column (ALTER COLUMN SET DATA TYPE)
- An ALTER INDEX statement that changes the padding attribute of an index with varying-length columns (PADDED to NOT PADDED or vice versa)

Examples

Example 1: Insert values into sample table DSN8910.EMP.

```
INSERT INTO DSN8910.EMP
VALUES ('000205','MARY','T','SMITH','D11','2866',
       '1981-08-10','ANALYST',16,'F','1956-05-22',
       16345,500,2300);
```

Example 2: Assume that SMITH.TEMPEMPL is a created temporary table. Populate the table with data from sample table DSN8910.EMP.

```
INSERT INTO SMITH.TEMPEMPL
SELECT *
FROM DSN8910.EMP;
```

Example 3: Assume that SESSION.TEMPEMPL is a declared temporary table. Populate the table with data from department D11 in sample table DSN8910.EMP.

```
INSERT INTO SESSION.TEMPEMPL
SELECT *
FROM DSN8910.EMP
WHERE WORKDEPT='D11';
```

Example 4: Insert a row into sample table DSN8910.EMP_PHOTO_RESUME. Set the value for column EMPNO to the value in host variable HV_ENUM. Let the value for column EMP_ROWID be generated because it was defined with a row ID data type and with clause GENERATED ALWAYS.

```
INSERT INTO DSN8910.EMP_PHOTO_RESUME(EMPNO, EMP_ROWID)
VALUES (:HV_ENUM, DEFAULT);
```

You can only insert user-specified values into ROWID columns that are defined as GENERATED BY DEFAULT and not as GENERATED ALWAYS. Therefore, in the above example, if you were to try to insert a value into EMP_ROWID instead of specifying DEFAULT, the statement would fail unless you also specify OVERRIDING USER VALUE. For columns that are defined as GENERATED ALWAYS, the OVERRIDING USER VALUE clause causes DB2 to ignore any user-specified value and generate a value instead.

For example, assume that you want to copy the rows in DSN8910.EMP_PHOTO_RESUME to another table that has a similar definition (both tables have a ROWID columns defined as GENERATED ALWAYS). For the

following INSERT statement, the OVERRIDING USER VALUE clause causes DB2 to ignore the EMP_ROWID column values from DSN8910.EMP_PHOTO_RESUME and generate values for the corresponding ROWID column in B.EMP_PHOTO_RESUME.

```
INSERT INTO B.EMP_PHOTO_RESUME
  OVERRIDING USER VALUE
  SELECT * FROM DSN8910.EMP_PHOTO_RESUME;
```

Example 5: Assume that the T1 table has one column. Insert a variable (:hv) number of rows of data into the T1 table. The values to be inserted are provided in a host-variable array (:hva).

```
EXEC SQL INSERT INTO T1 FOR :hv ROWS VALUES (:hva:hvind) ATOMIC;
```

In this example, :hva represents the host-variable array and :hvind represents the array of indicator variables.

Example 6: Assume that the T2 table has 2 columns, C1 is a SMALL INTEGER column, and C2 is an INTEGER column. Insert 10 rows of data into the T2 table. The values to be inserted are provided in host-variable arrays :hva1 (an array of INTEGERS) and :hva2 (an array of DECIMAL(15,0) values). The data values for :hva1 and :hva2 are represented in Table 125:

Table 125. Data values for :hva1 and :hva2

Array entry	:hva1	:hva2
1	1	32768
2	-12	90000
3	79	2
4	32768	19
5	8	36
6	5	24
7	400	36
8	73	4000000000
9	-200	2000000000
10	35	88

```
EXEC SQL INSERT INTO T2 (C1, C2)
  FOR 10 ROWS VALUES (:hva1:hvind1, :hva2:hvind2)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

After execution of the INSERT statement, the following information will be in the SQLCA:

```
SQLCODE = -253
SQLSTATE = 22529
SQLERRD3 = 8
```

Although an attempt was made to insert 10 rows, only 8 rows of data were inserted. Processing continued after the first failed insert because NOT ATOMIC CONTINUE ON SQLEXCEPTION was specified. You can use the GET DIAGNOSTICS statement to find further information, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

The result of this statement is num_rows = 8 and num_cond = 2 (2 conditions).

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
                             :sqlcode = DB2_RETURNED_SQLCODE,
                             :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 4.

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
                             :sqlcode = DB2_RETURNED_SQLCODE,
                             :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 8.

Example 7: Assume the above table T2 with two columns. C1 is a SMALL INTEGER column, and C2 is an INTEGER column. Insert 8 rows of data into the T2 table. The values to be inserted are provided in host-variable arrays :hva1 (an array of INTEGERS) and :hva2 (an array of DECIMAL(15,0) values.) The data values for :hva1 and :hva2 are represented in Table 125 on page 1381.

```
EXEC SQL INSERT INTO T2 (C1, C2)
  FOR 8 ROWS VALUES (:hva1:hvind1, :hva2:hvind2)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

After execution of the INSERT statement, the following information will be in the SQLCA:

```
SQLCODE = -253
SQLSTATE = 22529
SQLERRD3 = 6
```

Although an attempt was made to insert 8 rows, only 6 rows of data were inserted. Processing continued after the first failed insert because NOT ATOMIC CONTINUE ON SQLEXCEPTION was specified. You can use the GET DIAGNOSTICS statement to find further information, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

The result of this statement is num_rows = 68 and num_cond = 2 (2 conditions).

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE,
                             :sqlcode = DB2_RETURNED_SQLCODE,
                             :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 4.

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE,
                             :sqlcode = DB2_RETURNED_SQLCODE,
                             :row_num = DB2_ROW_NUMBER;
```

The result of this statement is sqlstate = 22003, sqlcode = -302, and row_num = 8.

Example 8: Assume that table T1 has two columns. Insert a variable number (:hvn) or rows into T1. The values to be inserted are in host-variable arrays :hva and :hvb. In this example, the INSERT statement is contained within the SELECT statement of cursor CS1. The SELECT statement makes use of two other input host variables (:hv1 and :hv2) in the WHERE clause. Either a static or dynamic INSERT statement can be used.

```
-- Static INSERT statement:
DECLARE CS1 CURSOR WITH ROWSET POSITIONING FOR
  SELECT *
    FROM FINAL TABLE
      (INSERT INTO T1 VALUES (:hva, :hvb) FOR :hvn ROWS)
   WHERE C1 > :hv1 AND C2 < :hv2;
OPEN CS1;
-- Dynamic INSERT statement:
```



```

PREPARE INSSMT FROM
    'SELECT *
      FROM FINAL TABLE
        (INSERT INTO T1 VALUES ( ? , ? ) FOR ? ROWS)
      WHERE C1 > ? AND C2 < ?';
DECLARE CS1 CURSOR WITH ROWSET POSITIONING FOR :INSSMT;
OPEN CS1 USING :hva, :hvb, :hvn, :hv1, :hv2; (or OPEN CS1 USING DESCRIPTOR ...)

```

If the host-variable arrays for the multiple-row INSERT statement were to be specified using a descriptor, that descriptor (SQLDA) would have to describe all input host variables in the statement, and the order of the entries in the SQLDA should be the same as the order of the order of the host variables, host-variable arrays, and values for the FOR n ROWS clauses in the statement. For example, given the statement above, the SQLVAR entries in the descriptor must be assigned in the following order: :hvn, :hva, :hvb, :hv1, hv2. In addition, the SQLVAR entries for host-variable arrays must be tagged in the SQLDA as column arrays (by specifying a special value in part of the SQLNAME field for a host variable), and the SQLVAR entry for the number of rows value must be tagged in the SQLDA (by specifying another special value in part of the SQLNAME field for the host variable).

LABEL

The LABEL statement adds or replaces labels in the descriptions of tables, views, aliases, or columns in the catalog at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

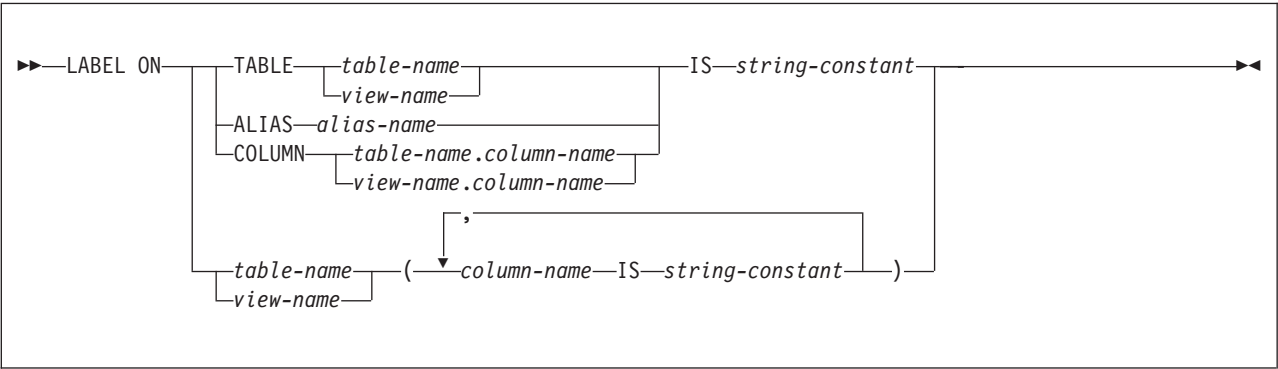
The privilege set that is defined below must include at least one of the following:

- Ownership of the table, view, or alias
- DBADM authority for its database (tables only)
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more details on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)

Syntax



Description

TABLE *table-name* or *view-name*

Identifies the table or view to which the label applies. The name must identify a table or view that exists at the current server. *table-name* must not identify a declared temporary table. The label is placed into the LABEL column of the SYSIBM.SYSTABLES catalog table for the row that describes the table or view.

ALIAS *alias-name*

Identifies the alias to which the label applies. The name must identify an alias that exists at the current server. The label is placed in the LABEL column of the SYSIBM.SYSTABLES catalog table for the row that describes the alias.

COLUMN *table-name.column-name* **or** *view-name.column-name*

Identifies the column to which the label applies. The name must identify a column of a table or view that exists at the current server. The name must not identify a column of a declared temporary table. The label is placed in the LABEL column of the SYSIBM.SYSCOLUMNS catalog table in the row that describes the column.

Do not use TABLE or COLUMN to define a label for more than one column in a table or view. Give the table or view name and then, in parentheses, a list in the form:

```
column-name IS string-constant,  
column-name IS string-constant,...
```

See Example 2 below.

The column names must not be qualified, each name must identify a column of the specified table or view, and that table or view must exist at the current server.

IS Introduces the label you want to provide.

string-constant

Can be any SQL character string constant of up to 30 bytes in length.

Examples

Example 1: Enter a label on the DEPTNO column of table DSN8910.DEPT.

```
LABEL ON COLUMN DSN8910.DEPT.DEPTNO  
IS 'DEPARTMENT NUMBER';
```

Example 2: Enter labels on two columns in table DSN8910.DEPT.

```
LABEL ON DSN8910.DEPT  
(MGRNO IS 'EMPLOYEE NUMBER FOR THE MANAGER',  
ADMNDEPT IS 'ADMINISTERING DEPARTMENT');
```

LOCK TABLE

The LOCK TABLE statement requests a lock on a table or table space at the current server. The lock is not acquired if the process already holds an appropriate lock.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

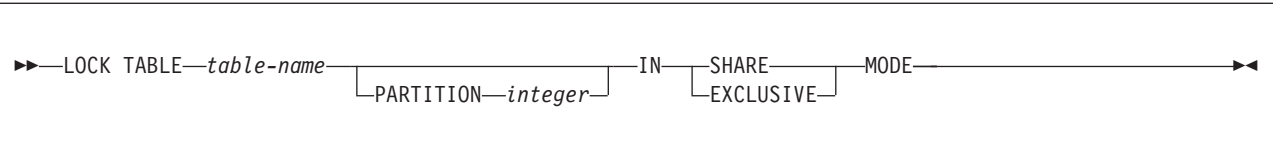
The privilege set that is defined below must include at least one of the following:

- The SELECT privilege on the identified table (the SELECT privilege does not apply to the auxiliary table)
- Ownership of the table
- DBADM authority for the database
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more details on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)

Syntax



Description

table-name

Identifies the table to be locked. The name must identify a table that exists at the current server. It must not identify a view, a temporary table (created or declared), or a catalog table. The lock might or might not apply exclusively to the table. The effect of locking an auxiliary table is to lock the LOB table space that contains the auxiliary table.

PARTITION *integer*

Identifies the partition of a partitioned table space to lock. The table identified by *table-name* must belong to a partitioned table space. The value specified for *integer* must be an integer that is no greater than the number of partitions in the table space.

IN SHARE MODE

For a lock on a table that is not an auxiliary table, requests the acquisition of a lock that prevents other processes from executing anything but read-only

operations on the table. For a lock on a LOB table space, IN SHARE mode requests a lock that prevents storage from being reallocated. When a LOB table space is locked, other processes can delete LOBs or update them to a null value, but they cannot insert LOBs with a nonnull value. The type of lock that the process holds after execution of the statement depends on what lock, if any, the process already holds.

IN EXCLUSIVE MODE

Requests the acquisition of an exclusive lock for the application process. Until the lock is released, it prevents concurrent processes from executing any operations on the table. However, unless the lock is on a LOB table space, concurrent processes that are running at an isolation level of uncommitted read (UR) can execute read-only operations on the table.

Notes

Releasing locks: If LOCK TABLE is a static SQL statement, the RELEASE option of bind determines when DB2 releases a lock. For RELEASE(COMMIT), DB2 releases the lock at the next commit point. For RELEASE(DEALLOCATE), DB2 releases the lock when the plan is deallocated (the application ends).

If LOCK TABLE is a dynamic SQL statement, DB2 uses RELEASE(COMMIT) and releases the lock at the next commit point, *unless* the table or table space is referenced by cached dynamic statements. Caching allows DB2 to keep prepared statements in memory past commit points. In this case, DB2 holds the lock until deallocation or until the commit after the prepared statements are freed from memory. Under some conditions, if a lock is held past a commit point, DB2 demotes the lock state of a segmented table or a nonsegmented table space to an intent lock at the commit point.

For more information on using LOCK TABLE (such as the size and duration of locks), and on locking in general, see *DB2 Application Programming and SQL Guide* or *DB2 Performance Monitoring and Tuning Guide*.

Syntax alternatives and synonyms: For compatibility with previous releases of DB2, PART can be specified as a synonym for PARTITION.

Example

Obtain a lock on the sample table named DSN8910.EMP, which resides in a partitioned table space. The lock obtained applies to every partition and prevents other application programs from either reading or updating the table.

```
LOCK TABLE DSN8910.EMP IN EXCLUSIVE MODE;
```

MERGE

The MERGE statement updates a target (a table or view, or the underlying tables or views of a fullselect) using the specified input data. Rows in the target that match the input data are updated as specified, and rows that do not exist in the target are inserted. Updating or inserting a row into a view updates or inserts the row into the tables on which the view is based, if no INSTEAD OF trigger is defined on this view.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges that are held by the privilege set that is defined below must include at least one of the following privileges:

- SYSADM authority
- Ownership of the table
- If the search condition contains a reference to a column of the table or view, the SELECT privilege for the referenced table or view
- If the insert operation is specified, the INSERT privilege for the table or view
- If the update operation is specified, at least one of the following privileges is required:
 - the UPDATE privilege for the table or view
 - the UPDATE privilege on each column that is updated
 - If the right side of the assignment clause contains a reference to a column of the table or view, the SELECT privilege for the referenced table or view

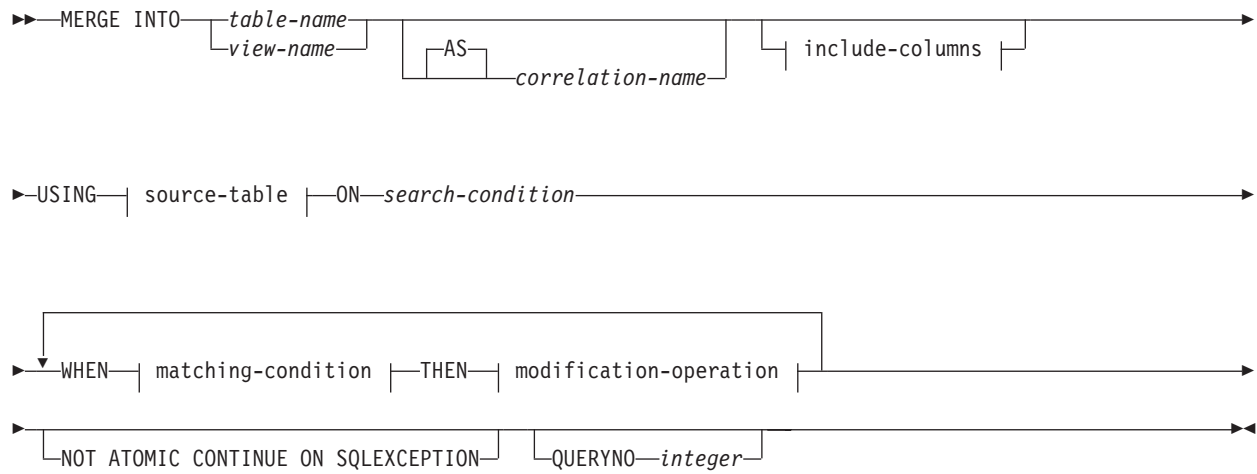
If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

If the insert operation or assignment clause includes a subquery, the privileges that are held by the privilege set must also include at least one of the following privileges:

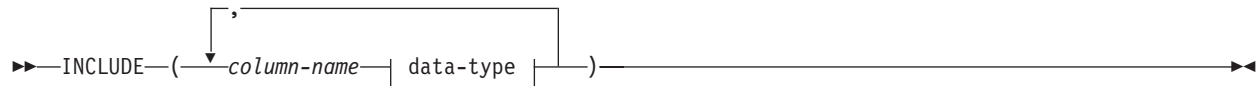
- SYSADM authority
- The SELECT privilege on every table or view that is identified in the subquery
- Ownership of the tables or views that are identified in the subquery

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)

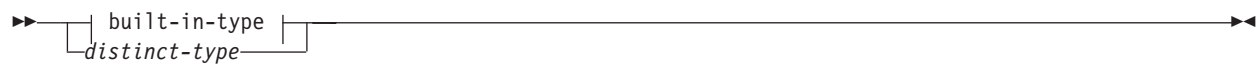
Syntax



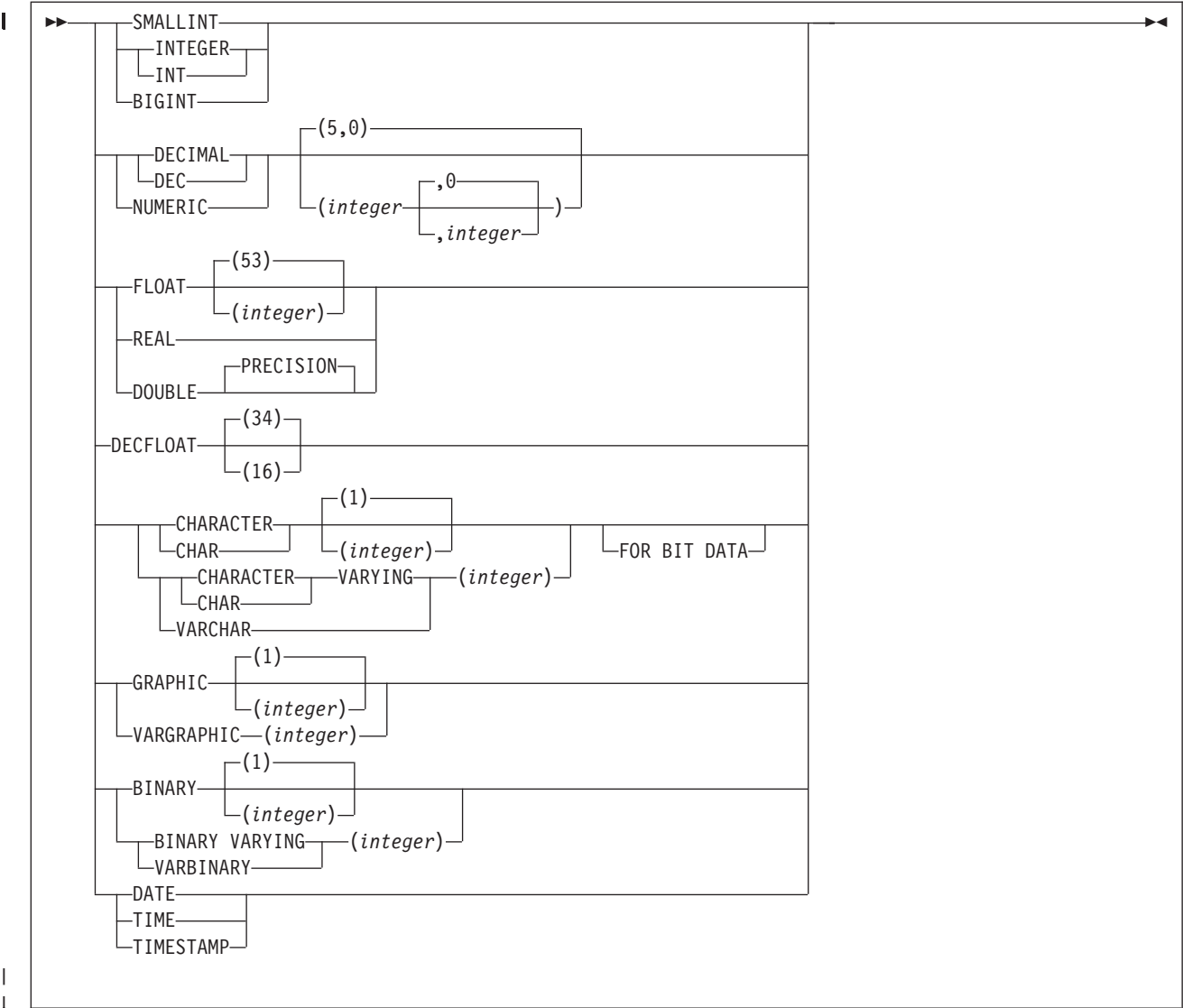
include-columns:



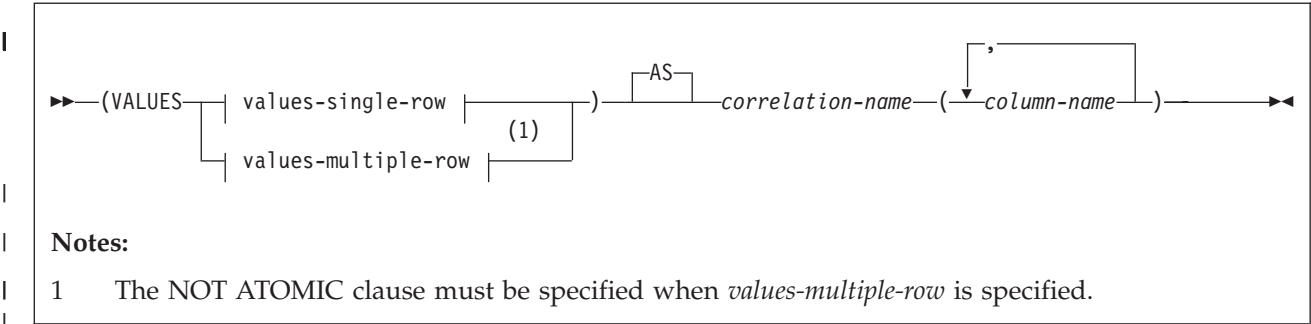
data-type:



built-in-type:



source-table:



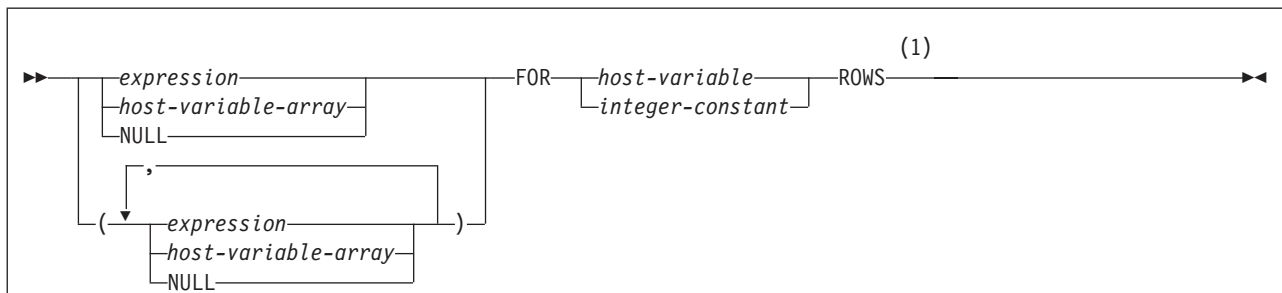
Notes:

- 1 The NOT ATOMIC clause must be specified when *values-multiple-row* is specified.

values-single-row:



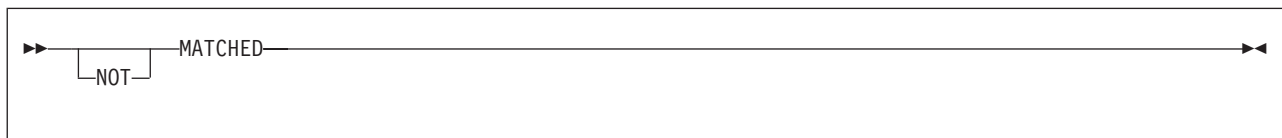
values-multiple-row:



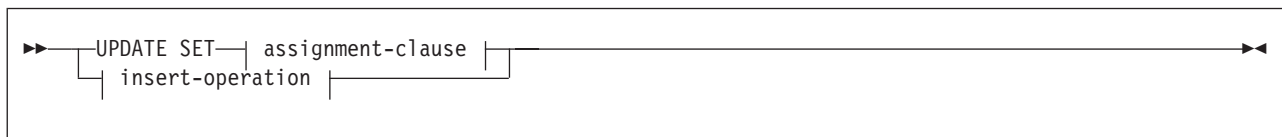
Notes:

- 1 For a static MERGE statement, if FOR n ROWS is not specified, *values-multiple-row* is treated as *values-single-row*. For a dynamic MERGE statement, FOR n ROWS does not need to be specified in the MERGE statement. It can be specified in the EXECUTE statement, but cannot be specified in both the MERGE and EXECUTE statements.

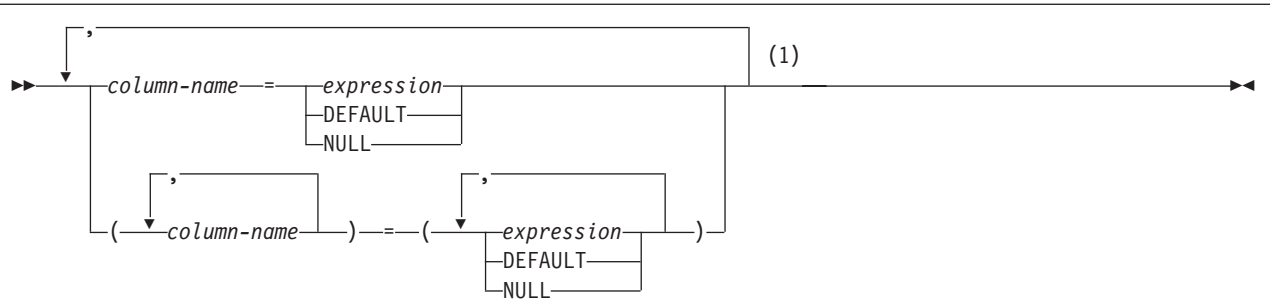
matching-condition:



modification-operation:



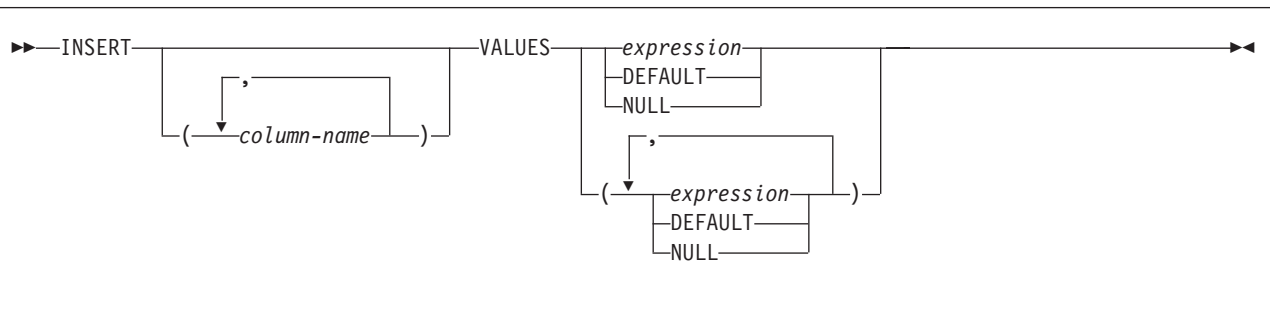
assignment-clause:



Notes:

- 1 The number of expressions, DEFAULT, and NULL keywords must match the number of column-names.

insert-operation:



Description

INTO *table-name* or *view-name*

Identifies the target of the insert or update operations of the MERGE statement. The name must identify a table or view that exists at the current server. The name must not identify:

- A catalog table
- A created global temporary table
- A read-only view
- A system-maintained materialized query table
- A table that is implicitly created for an XML column

If a view is specified as the target of the MERGE statement, the view must not be defined with any INSTEAD OF triggers.

AS *correlation-name*

correlation-name provides an alternate name that can be used when referencing columns of the intermediate result table. If no *correlation-name* is specified, the name of the intermediate result table is the name of the target table or view of the MERGE statement. Otherwise, the name is the *correlation-name*.

include-column

Specifies a set of columns that are included, along with the columns of the specified table or view, in the result table of the MERGE statement when it is nested in the FROM clause of the outer fullselect that is used in a SELECT statement, or in a SELECT INTO statement. The included columns are

appended to the end of the list of columns that are identified by *table-name* or *view-name*. If a value is not specified for an included column, a null value is returned for that column.

INCLUDE

Introduces a list of columns that is to be included in the result table of the MERGE statement. The included columns are only available if the MERGE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. INCLUDE can only be specified when the MERGE statement is nested in the FROM clause of a SELECT statement.

column-name

Specifies the name for a column of the result table of the MERGE statement that is not the same name as another included column or a column in the table or view that is specified in *table-name* or *view-name*.

data-type

Specifies the data type of the included column. The included columns are nullable.

built-in-type

Specifies a built-in data type. See “CREATE TABLE” on page 1079 for a description of each built-in type.

distinct-type

Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

USING VALUES *values-single-row* **or** *values-multiple-row*

Specifies the values for the row data to merge into the target table or view. *values-single-row* specifies a single row of source data. *values-multiple-row* specifies multiple rows of source data.

expression

Specifies an expression of the type that is described in “Expressions” on page 180. The expression must not include a column name. The expression must not reference a NEXT VALUE or PREVIOUS VALUE expression. If the expression is a single host variable, the host variable can identify a structure. Any host variable or structure that is specified must be described in the application program according to the rules for declaring host structures and variables.

To provide a null value, specify the NULL keyword on a CAST specification.

host-variable-array

Specifies a host variable array. Each host variable array must be defined in the application program in accordance with the rules for declaring an array. A host variable array contains the data to merge into a target column. The number of rows must be less than or equal to the dimension of each of the host variable arrays. An optional indicator array can be specified for each host variable array. An indicator array should be specified if the SQLTYPE of any SQLVAR occurrence indicates that a column is nullable. The dimension of the indicator array must be large enough to contain an indicator for each row of input data.

A host structure is not supported in *host-variable-array*.

host-variable-array is supported in C/C++, COBOL, and PL/I.

FOR *host-variable* or *integer-constant* ROWS

Specifies the number of rows to merge. For a dynamic MERGE statement, this clause can be specified on the EXECUTE statement. *host-variable* or *integer-constant* is assigned to a value *k*. If *host-variable* is specified, it must be an exact numeric type with a scale of zero and must not include an indicator variable. *k* must be in the range of 1 to 32767. *k* rows are merged into the target from the specified source data.

If a parameter marker is specified in FOR *n* ROWS, a value must be provided with the USING clause of the associated EXECUTE statement.

AS *correlation-name*

Specifies a correlation name for the *source-table*.

column-name

Specifies a column name to associate the input data to the SET *assignment-clause* for an update operation or the VALUES clause for an insert operation.

ON *search-condition*

Specifies join conditions between the *source-table* and the target table or view.

Each *column-name* in the search condition must name a column of the target table, view, or *source-table*. A subquery is not allowed in the *search-condition*. If a *column-name* exists in both the target and the *source-table*, the column name must be qualified.

For each row of the *source-table*, the *search-condition* is applied to each row of the target. If the *search-condition* is evaluated as true and the target is not empty, the specified WHEN MATCHED clause is used. Otherwise, the specified WHEN NOT MATCHED clause is used.

WHEN MATCHED or WHEN NOT MATCHED

Specifies the condition under which the *modification-operation* is run.

WHEN MATCHED

Specifies the operation to perform on the rows where the ON *search-condition* is true and the target is not empty. Only UPDATE can be specified after the THEN clause. WHEN MATCHED must not be specified more than one time.

WHEN NOT MATCHED

Specifies the operation to perform on the rows where the ON *search-condition* is false or unknown, or the target is empty. Only INSERT can be specified after the THEN clause. WHEN NOT MATCHED must not be specified more than one time.

THEN *update-operation* or THEN *insert-operation*

Specifies the operation to run when the *matching-condition* evaluates to true.

UPDATE SET

Specifies the update operation to run when the *matching-condition* evaluates to true.

The rows that are updated from a *source-row* are subject to more updates by subsequent *source-rows* in the same statement. The update is cumulative.

assignment-clause

Specifies a list of column updates.

column-name

Identifies a column to update. *column-name* must identify a column of the specified table or view, and that column must be updatable. The

column must not be a generated column, or a column of a view that is derived from a scalar function, a constant, or an expression. *column-name* can also identify an included column. The same *column-name* must not be specified more than one time.

Assignments to included columns are only processed when the MERGE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. There must be at least one assignment clause that specifies a *column-name* that is not an included column. A view column that is derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same MERGE statement.

expression

Specifies the new value of the column. The expression is any expression of the type that is described in “Expressions” on page 180. The expression must not include an aggregate function.

An expression can contain references to columns of *source-table* or target. A column name is first checked as a column of the target, and then checked as a column of the source table. For each row that is updated, the value of a target column in an expression is the value of the column in the row before the row is updated. *expression* cannot contain references to an included column.

DEFAULT

Specifies the default value for the column. The value that is assigned depends on how the column is defined.

A ROWID column must not be set to the DEFAULT keyword.

An identity column or a row change timestamp column that is defined as GENERATED ALWAYS can be set only to the DEFAULT keyword.

If the column is defined using the NOT NULL clause and the GENERATED clause is not used, or the WITH DEFAULT clause is not used, the DEFAULT keyword cannot be specified for that column.

NULL

Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

insert-operation

Specifies the insert operation to run for the rows where the *matching-condition* evaluates to true.

The rows that are inserted from a source-row are immediately subject for update by subsequent source-rows in the same statement.

INSERT

Specifies a list of column names and row value expressions to use of the insert operation.

The number of values for the row in the row-value expression must be equal to the number of names in the insert column list. The first value is inserted into the first column in the list, the second value into the second column, and so on.

column-name

Specifies the columns for which the insert values are provided. Each name must identify a column of the table or view

If an included column is not specified in the list of column names, the value of the included column is set to null. The column list cannot contain only included columns.

The same column must not be specified more than one time. A view column that cannot accept insert values must not be specified. A value cannot be inserted into a view column that is derived from one of the following:

- a constant, an expression, or a scalar function
- the same column of the base table as another column of the view

If the object of the operation is a view that contains columns that cannot accept insert values, a list of column names must be specified and the list must not specify these columns.

Omission of the column list is an implicit specification of a list in which every column of the table (that is not defined as implicitly hidden) or view is identified in left-to-right order. This list is established when the statement is prepared and therefore does not include columns that were added to the table after the statement was prepared.

VALUES

Introduces one or more rows of values to insert.

expression

Specifies an expression of the type that does not include a column name of the target. If *expression* is a host variable, the host variable can identify a host structure.

DEFAULT

Specifies to assign the default value to the column. DEFAULT must only be specified for columns that have a default value. If the column is specified in the INCLUDE column list, the column value is set to null.

DEFAULT must be specified for a column that is defined as GENERATED ALWAYS. A valid value can be specified for a column that is defined as GENERATED BY DEFAULT.

NULL

Specifies the null value as the value of the column. Specify NULL only for nullable columns.

NOT ATOMIC CONTINUE ON SQLEXCEPTION

The rows of input data are processed separately. Any statement level triggers are processed for each row of source data that is processed, and the transition table includes the individual row that was processed. When errors are encountered and this option is in effect, processing continues, and some of the specified rows will not be processed. In this case, if an appropriate trigger is defined on the underlying base table, the statement level trigger will only be activated for rows that were successfully processed.

Regardless of the failure of any particular source row, the MERGE statement will not undo any changes that are made to the database by the statement. Merge will be attempted for rows that follow the failed row. However, the minimum level of atomicity is at least that of a single source row (that is, it is not possible for a partial merge to complete), including any triggers that might have been activated as a result of the MERGE statement.

QUERYNO *integer*

Specifies the number for this SQL statement that is used in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If QUERYNO is not specified, the number that is associated with the SQL statement is the statement number that is assigned during precompilation. Thus, if the application program is changed and then precompiled, the statement number might change.

Notes

SQLCA and GET DIAGNOSTICS considerations: The GET DIAGNOSTICS statement can be used immediately after the MERGE statement to check which input rows fail during the merge operation. The GET DIAGNOSTICS statement information item, NUMBER, indicates the number of conditions that are raised. The GET DIAGNOSTICS condition information item, DB2_ROW_NUMBER, indicates the input source rows that cause an error.

Trigger considerations: A MERGE statement might cause triggers to be activated. A trigger might cause other statements to be executed or raise error conditions based on the source data values. A before-update or before-insert trigger processes immediately before the update or insert operation.

If a source row results in an insert, any after-insert triggers are activated after the insert operation completes.

If a source row results in updates, any after-update triggers are activated after all of the update operations complete.

Indexes with VARBINARY columns: If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.

Considerations for a MERGE without a column list in insert-operation: A MERGE statement without a column list specified as part of *insert-operation* does not include implicitly hidden columns, so such columns must have a defined default value.

Considerations for non-atomic processing of a MERGE statement: When NOT ATOMIC is specified the rows of source data are processed separately. Any references to special registers, sequence expressions, and functions in the MERGE statement are evaluated as each row of source data is processed. Statement level triggers are activated as each row of source data is processed.

If one or more errors occur during the operation for a row of source data, processing continues. The row that was being processed at the time of the error is not inserted or updated. Execution continues with the next row to be processed, and any other changes made during the execution of the multiple-row MERGE statement are not backed out. However, the processing of an individual row is an atomic action.

DRDA considerations: DB2 Connect™ Version 9.1 and subsequent releases support the MERGE statement. The support is for CLI only, with no embedded static SQL support.

When running a MERGE statement at a DB2 for z/OS requester, cases might exist where the requester does not know the number of rows in the source table. This includes the following cases:

- For static or dynamic MERGE statements, of the FOR *n* ROWS clause contains a constant value for *n*.
- For dynamic MERGE statements, of host variables are specified on the USING clause of an EXECUTE statement.

For both of these cases, if the number of rows in the source table is not known, the requester might send more data than is required to the server. The number of rows that are actually processed will be correct because the server knows the correct numbers of rows to process. However, performance might be adversely affected. Consider the following example:

```
...long serial num [10];
struct { short len;
char data [18];
}
name[20]...
EXEC SQL
MERGE INTO T1
  USING (VALUES (:serial_num, :name))
  FOR 5 ROWS...
```

When this statement is run at the requester, the number of rows to merge (five) is not known. As a result, the requester will send 10 values for serial-name and name to the server because 10 is the size of the smallest host variable array and is, therefore, the maximum number of rows that can merge without causing an error.

Do the following to help minimize performance problems:

- Avoid using numeric constants in the FOR *n* ROWS clause of the MERGE statement. For static MERGE statements, avoiding numeric constants ensures that the values for *n* will be known at the requester.
- For dynamic MERGE statements, use the USING DESCRIPTOR clause instead of the USING *host-variable* clause on the EXECUTE statement. If a USING DESCRIPTOR clause is used on the EXECUTE statement, the value for *n* must be indicated in the descriptor.
- If either of the previous methods cannot be used, do the following:
 - Make your host variable arrays as small as possible, or declare that the size of your host variable arrays are the size of *n* in the descriptor. This avoids sending a large number of unused host variable array entries to the server.
 - Ensure that varying length string arrays are initialized to a length of 0 (zero). Doing so minimizes the amount of data that is sent to the server.
 - Ensure that decimal host variable arrays are initialized to valid values. Doing so causes the requester to avoid sending a negative SQLCODE if the requester encounters invalid decimal data.

Examples

Example 1: For activities that exist in the archive table, update the description in the archive table. Otherwise, insert the activity and its description into the archive table:


```

MERGE INTO ARCHIVE AR
  USING (VALUES (:hv_activity, :hv_description)
        FOR :hv_nrows ROWS)
    AS AC (ACTIVITY, DESCRIPTION)
  ON (AR.ACTIVITY = AC.ACTIVITY)
  WHEN MATCHED THEN UPDATE SET DESCRIPTION = AC.DESRIPTION
  WHEN NOT MATCHED THEN INSERT (ACTIVITY, DESCRIPTION)
    VALUES (AC.ACTIVITY, AC.DESRIPTION)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;

```

Example 2: Use the transaction data to merge rows into the account table. Update the balance from the transaction data against an account ID and insert new accounts from the transaction data where the accounts do not already exist.

```

MERGE INTO ACCOUNT AS A
  USING (VALUES (:hv_id, :hv_amount)
        FOR 3 ROWS)
    AS T (ID, AMOUNT)
  ON (A.ID = T.ID)
  WHEN MATCHED THEN UPDATE SET BALANCE = A.BALANCE + T.AMOUNT
  WHEN NOT MATCHED THEN INSERT (ID, BALANCE)
    VALUES (T.ID, T.AMOUNT)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;

```

Example 3: Update the list of activities that are organized by group A in the archive table. Update the activities information (description and date when last modified) in the archive table if the activities exist in the archive table and are also organized by group A. Insert new activities into the archive table.

```

-- hv_nrows = 3
-- hv_activity(1) = 'D'; hv_description(1) = 'Dance'; hv_date(1) = '03/01/07'
-- hv_activity(2) = 'S'; hv_description(2) = 'Singing'; hv_date(2) = '03/17/07'
-- hv_activity(3) = 'T'; hv_description(3) = 'Tai-chi'; hv_date(3) = '05/01/07'
-- hv_group = 'A';
-- note that hv_group is not an array. All 3 values contain the same values
MERGE INTO ARCHIVE AR
  USING (VALUES (:hv_activity, :hv_description, :hv_date, :hv_group)
        FOR :hv_nrows ROWS)
    AS AC (ACTIVITY, DESCRIPTION, DATE, GROUP)
  ON AR.ACTIVITY = AC.ACTIVITY AND AR.GROUP = AC.GROUP
  WHEN MATCHED
  THEN UPDATE SET (DESCRIPTION, DATE, LAST_MODIFIED)
    = (AC.DESRIPTION, AC.DATE, CURRENT_TIMESTAMP)
  WHEN NOT MATCHED
  THEN INSERT (GROUP, ACTIVITY, DESCRIPTION, DATE, LAST_MODIFIED)
    VALUES (AC.GROUP, AC.ACTIVITY, AC.DESRIPTION, AC.DATE, CURRENT_TIMESTAMP)
  NOT ATOMIC CONTINUE ON SQLEXCEPTION;

```

OPEN

The OPEN statement opens a cursor so that it can be used to process rows from its result table.

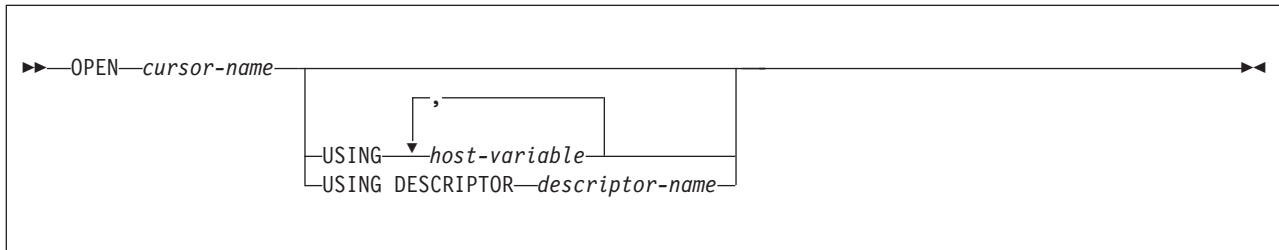
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See “DECLARE CURSOR” on page 1191 for the authorization required to use a cursor.

Syntax



Description

cursor-name

Identifies the cursor to be opened. The *cursor-name* must identify a declared cursor as explained in “DECLARE CURSOR” on page 1191. When the OPEN statement is executed, the cursor must be in the closed state.

The SELECT statement of the cursor is either one of the following types of SELECT statements:

- The *select-statement* that is specified in the DECLARE CURSOR statement
- The prepared *select-statement* that is identified by the *statement-name* that is specified in the DECLARE CURSOR statement.

If the statement has not been successfully prepared, or is not a *select-statement*, the cursor cannot be successfully opened.

The result table of the cursor is derived by evaluating the SELECT statement. The evaluation uses the current values of any special registers or PREVIOUS VALUE expressions that are specified in the SELECT statement and the current values of any host variables that are specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table can be derived during the execution of the OPEN statement and a temporary copy of a result table can be created to hold those rows. They can be derived during the execution of later FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table. If the table is empty, the position of the cursor is effectively “after the last row.” The DB2 system does not indicate an empty table when the OPEN statement is executed. But it does indicate that condition, on the first execution of FETCH, by returning values of +100 for SQLCODE and '02000' for SQLSTATE.

USING

Introduces a list of host variables whose values are substituted for the parameter markers (question marks) or host variables in the statement of the cursor, depending on the declaration of the cursor:

- If the DECLARE CURSOR statement included *statement-name*, the statement was prepared with a PREPARE statement. The host variables specified in the USING clause of the OPEN statement replace any parameter markers in the prepared statement. This reflects the typical use of the USING clause of the OPEN statement. For an explanation of parameter marker replacement, see “PREPARE” on page 1405.

If the prepared statement includes parameter markers, you must use USING. If the prepared statement does not include parameter markers, USING is ignored.

- If the DECLARE CURSOR statement included *select-statement* and the SELECT statement included host variables, the USING clause of the OPEN statement can be used to specify host variables that are to override the values that were specified when the cursor was defined. In this case, the OPEN statement is executed as if each host variable in the SELECT statement were a parameter marker except that the attributes of the target variable are the same as the host variables in the SELECT statement. The effect is to override the values of the host variables in the SELECT statement of the cursor with the values of the host variables specified in the USING clause. The overriding value is always the value of the main variable because indicator variables are ignored in this context without warning.

host-variable,...

Identifies host structures or variables that must be described in the application program in accordance with the rules for declaring host structures and variables. When the statement is executed, a reference to a structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Where appropriate, locator variables can be provided as the source of values for parameter markers.

DESCRIPTOR *descriptor-name*

Identifies an SQLDA that contains a valid description of the input host variables.

Before the OPEN statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
A REXX SQLDA does not contain this field.
- SQLABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables

The SQLDA must have enough storage to contain all SQLVAR occurrences. If LOBs or distinct types are present in the result table, there must be additional SQLVAR entries for each input host variable. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see “SQL descriptor area (SQLDA)” on page 1656.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement.

See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C.

Notes

Errors occurring on OPEN: In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/REOPT(ONCE) bind options can cause some SQL statements to receive “delayed” errors. For example, an OPEN statement might receive an SQLCODE that normally occurs during PREPARE processing. Or a FETCH statement might receive an SQLCODE that normally occurs at OPEN time.

Closed state of cursors: All cursors in an application process are in the closed state when:

- The application process is started.
- A new unit of work is started for the application process unless the WITH HOLD option has been used in the DECLARE CURSOR statement.
- The application was precompiled with the CONNECT(1) option (which implicitly closes any open cursors).

A cursor can also be in the closed state because:

- A CLOSE statement was executed.
- An error was detected that made the position of the cursor unpredictable.

To retrieve rows from the result table of a cursor, you must execute a FETCH statement when the cursor is open. The only way to change the state of a cursor from closed to open is to execute an OPEN statement.

Effect of a temporary copy of a result table: DB2 can process a cursor in two different ways:

- It can create a temporary copy of the result table during the execution of the OPEN statement. You can specify INSENSITIVE SCROLL on the cursor to force the use of a temporary copy of the result table.
- It can derive the result table rows as they are needed during the execution of later FETCH statements.

If the result table is not read-only, DB2 uses the latter method. If the result table is read-only, either method could be used. The results produced by these two methods could differ in the following respects:

When a temporary copy of the result table is used: An error can occur that would otherwise not occur until some later FETCH statement. Insert operations that are executed while the cursor is open cannot affect the result table once all the rows have been materialized in the temporary copy of the result table. For a scrollable insensitive cursor, update and delete operations that are executed while the cursor is open cannot affect the result table. For a scrollable sensitive static cursor, update and delete operations can affect the result table if the rows are subsequently fetched with sensitive FETCH statements.

When a temporary copy of the result table is not used: Insert, update, and delete operations that are executed while the cursor is open can affect the result table. The effect of such operations is not always predictable.

For example, if cursor C is positioned on a row of its result table defined as `SELECT * FROM T`, and you insert a row into T, the effect of that insert on the result table is not predictable because its rows are not ordered. A later `FETCH C` might or might not retrieve the new row of T. To avoid these changes, you can specify `INSENSITIVE SCROLL` for the cursor to force the use of a temporary copy of the result table.

Parameter marker replacement: Before the `OPEN` statement is executed, each parameter marker in the query is effectively replaced by its corresponding host variable. The replacement is an assignment operation in which the source is the value of the host variable and the target is a variable within DB2. The assignment rules are those described for assignment to a column in “Assignment and comparison” on page 102. For a typed parameter marker, the attributes of the target variable are those specified by the `CAST` specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Parameter markers.

Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column:

- V must be compatible with the target.
- If V is a string, its length (excluding trailing blanks) must not be greater than the length attribute of the target.
- If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, V must not be null.

When the `SELECT` statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by the value of its corresponding host variable. For example, if V is `CHAR(6)` and the target is `CHAR(8)`, the value used in place of P is the value of V padded on the right with two blanks. For more on the process of replacement, see Parameter marker replacement.

Considerations for scrollable cursors: Following an `OPEN cursor` statement, a `GET DIAGNOSTICS` statement can be used to get the attributes of the cursor such as the following information (for more information, see “GET DIAGNOSTICS” on page 1317):

- `DB2_SQL_ATTR_CURSOR_HOLD`. Whether the cursor was defined with the `WITH HOLD` attribute.
- `DB2_SQL_ATTR_CURSOR_SCROLLABLE`. Scrollability of the cursor.
- `DB2_SQL_ATTR_CURSOR_SENSITIVITY`. Effective sensitivity of the cursor.
The sensitivity information can be used by applications (such as an ODBC driver) to determine what type of `FETCH` (`INSENSITIVE` or `SENSITIVE`) to issue for a cursor defined as `ASENSITIVE`.
- `DB2_SQL_ATTR_CURSOR_ROWSET`. Whether the cursor can be used to access rowsets.
- `DB2_SQL_ATTR_CURSOR_TYPE`. Whether a cursor type is forward-only, static, or dynamic.

In addition, if subsystem parameter DISABSCS is set to NO, a subset of the above information is returned in the SQLCA:

- The scrollability of the cursor is in SQLWARN1.
- The sensitivity of the cursor is in SQLWARN4.
- The effective capability of the cursor is in SQLWARN5.

Number of rows inserted: SQL data change statements and routines that modify SQL data embedded in the cursor definition are completely executed, and the result table is stored in a temporary table when the cursor opens. If statement execution is successful, the SQLERRD(3) field contains the sum of the number of rows that qualified for insert, update, and delete operations. If an error occurs during execution of an OPEN statement that involves a cursor that contains a data change statement within a fullselect, the results of that data change statement are rolled back.

Materialization of the rows of the result table and NEXT VALUE expressions: If the rows of the result table of a cursor are materialized when the cursor is opened and the SELECT statement of the cursor contains NEXT VALUE expressions, the expressions are processed when the cursor is opened. Otherwise, the NEXT VALUE expressions are evaluated as the rows of the result table are retrieved.

Example

The OPEN statement in the following example places the cursor at the beginning of the rows to be fetched.

```
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT DEPTNO, DEPTNAME, MGRNO FROM DSN8910.DEPT
    WHERE ADMRDEPT = 'A00';
EXEC SQL OPEN C1;
DO WHILE (SQLCODE = 0);
    EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM;
END;
EXEC SQL CLOSE C1;
```

PREPARE

The PREPARE statement creates an executable SQL statement from a string form of the statement. The character-string form is called a *statement string*. The executable form is called a *prepared statement*.

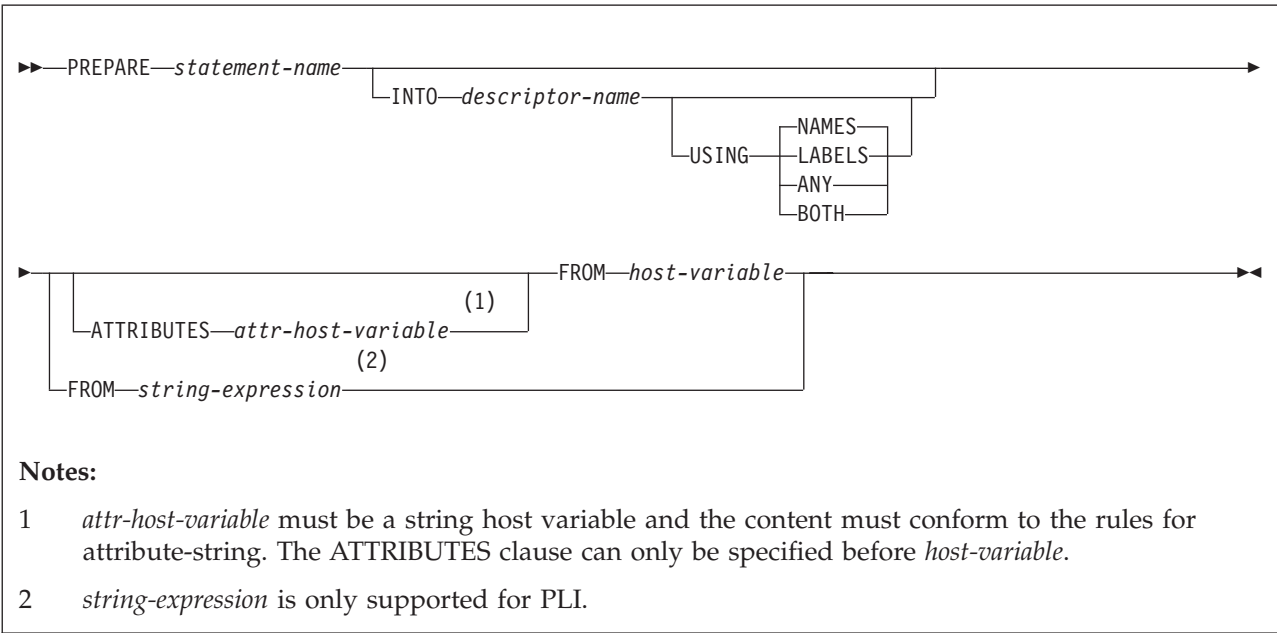
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

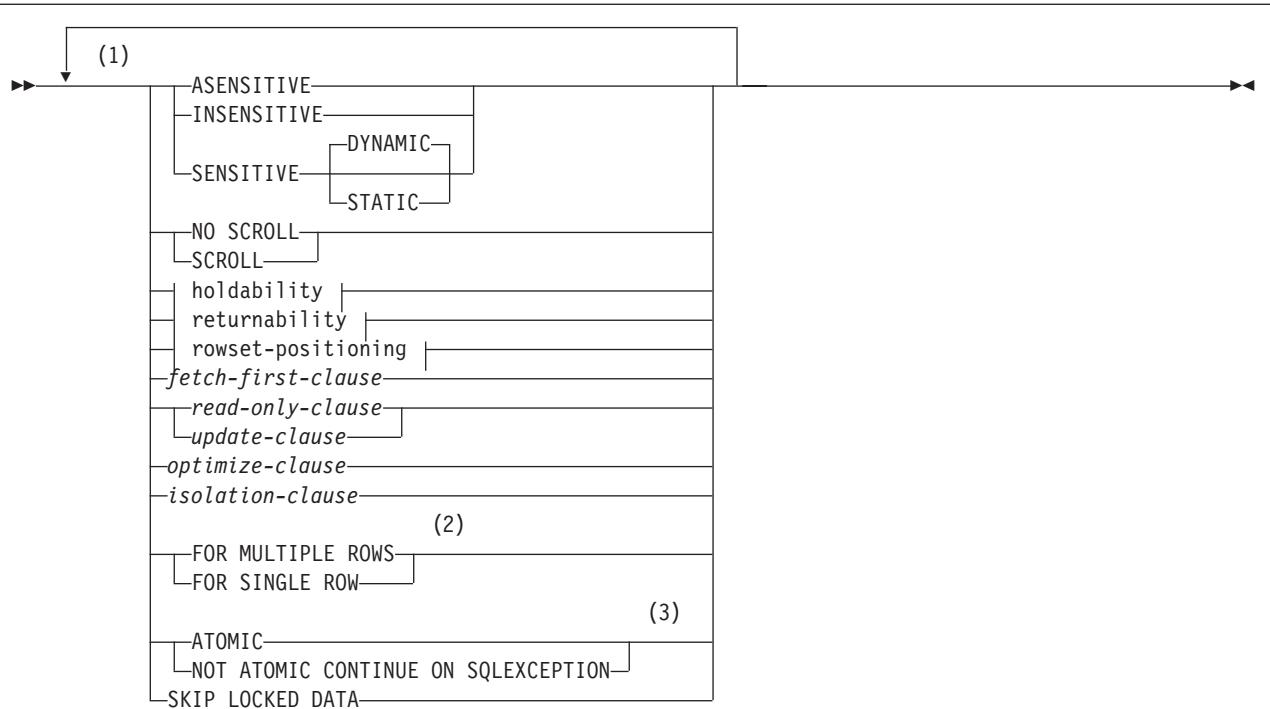
Authorization

The authorization rules are those defined for the dynamic preparation of the SQL statement specified by the PREPARE statement. For example, see Chapter 4, “Queries,” on page 629 for the authorization rules that apply when a SELECT statement is prepared.

Syntax



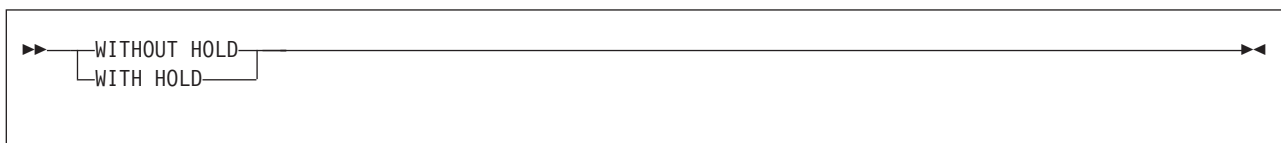
attribute-string



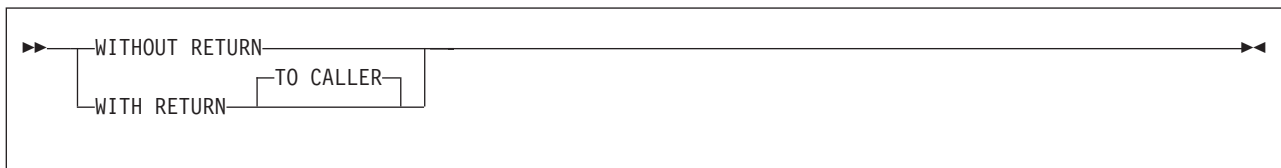
Notes:

- 1 The same clause must not be specified more than once. If the options are not specified, their defaults are whatever was specified for the corresponding option in an associated DECLARE CURSOR or INSERT statement.
- 2 The FOR SINGLE ROW or FOR MULTIPLE ROWS clause must only be specified for an INSERT or a MERGE statement.
- 3 The ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION clause must only be specified for an INSERT statement.

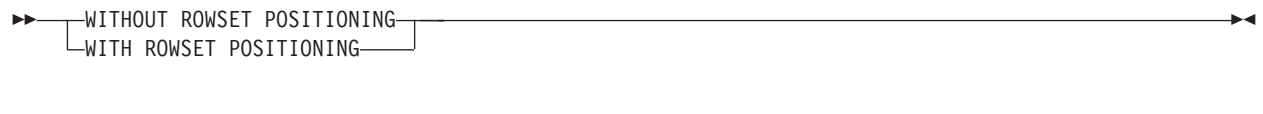
holdability:



returnability:



rowset-positioning:



Description

statement-name

Names the prepared statement. If the name identifies an existing prepared statement, that prepared statement is destroyed. The name must not identify a prepared statement that is the SELECT statement of an open cursor.

INTO

If you use INTO, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by the descriptor name. Thus, the PREPARE statement:

```
EXEC SQL PREPARE S1 INTO :SQLDA FROM :V1;
```

is equivalent to:

```
EXEC SQL PREPARE S1 FROM :V1;  
EXEC SQL DESCRIBE S1 INTO :SQLDA;
```

descriptor-name

Identifies the SQLDA. For languages other than REXX, SQLN must be set to indicate the number of SQLVAR occurrences. See “DESCRIBE” on page 1237 and “SQL descriptor area (SQLDA)” on page 1656 for information about how to determine the number of SQLVAR occurrences to use and for an explanation of the information that is placed in the SQLDA.

See “Identifying an SQLDA in C or C++” on page 1674 for how to represent *descriptor-name* in C.

USING

Indicates what value to assign to each SQLNAME variable in the SQLDA when INTO is used. If the requested value does not exist, SQLNAME is set to length 0.

NAMES

Assigns the name of the column. This is the default.

LABELS

Assigns the label of the column. (Column labels are defined by the LABEL statement.)

ANY

Assigns the column label, and, if the column has no label, the column name.

BOTH

Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result table contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $2 \times n$ or $3 \times n$, where n is the number of columns in the object being described. For each of the columns, the first n occurrences of SQLVAR, which are the base SQLVAR entries, contain the column names. Either the second or third n occurrences of SQLVAR, which are the extended SQLVAR entries, contain the column labels. If there are

no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

A REXX SQLDA does not include the SQLN field, so you do not need to set SQLN for REXX programs.

ATTRIBUTES *attr-host-variable*

Specifies the attributes for this cursor that are in effect if a corresponding attribute has not been specified as part of the outermost fullselect of the associated SELECT statement. If attributes are specified for the outermost fullselect, they are used instead of the corresponding attributes specified on the PREPARE statement. In turn, if attributes are specified in the PREPARE statement, they are used instead of the corresponding attributes specified on a DECLARE CURSOR statement.

attr-host-variable must identify a host variable that is described in the program in accordance with the rules for declaring string variables. *attr-host-variable* must be a string variable (either fixed-length or varying-length) that has a length attribute that does not exceed the maximum length of a CHAR or VARCHAR. Leading and trailing blanks are removed from the value of the host variable. The host variable must contain a valid *attribute-string*.

An indicator variable can be used to indicate whether or not attributes are actually provided on the PREPARE statement. Thus, applications can use the same PREPARE statement regardless of whether attributes need to be specified or not.

The options that can be specified as part of the *attribute-string* are as follows:

ASENSITIVE, INSENSITIVE, SENSITIVE STATIC, or SENSITIVE DYNAMIC

Specifies the desired sensitivity of the cursor to inserts, updates, or deletes that made to the rows underlying the result table. The sensitivity of the cursor determines whether DB2 can materialize the rows of the result into a temporary table. The default is ASENSITIVE.

ASENSITIVE

Specifies that the cursor should be as sensitive as possible. A cursor that defined as ASENSITIVE will be either insensitive or sensitive dynamic; it will not be sensitive static. For information about how the effective sensitivity of the cursor is returned to the application with the GET DIAGNOSTICS statement or in the SQLCA, see "OPEN" on page 1400.

INSENSITIVE

Specifies that the cursor does not have sensitivity to inserts, updates, or deletes that are made to the rows underlying the result table. As a result, the size of the result table, the order of the rows, and the values for each row do not change after the cursor is opened. In addition, the cursor is read-only. The SELECT statement or *attribute-string* of the PREPARE statement cannot contain a FOR UPDATE clause, and the cursor cannot be used for positioned updates or deletes.

SENSITIVE

Specifies that the cursor has sensitivity to changes made to the database after the result table is materialized. The cursor is always

35. The scrollability and sensitivity of the cursor are independent and do not have to be specified together. Thus, the cursor might be defined as SCROLL INSENSITIVE, but the PREPARE statement might specify SENSITIVE STATIC as an override for the sensitivity.

| sensitive to positioned updates and deletes that are made using the
| same cursor. However, the *select-statement* of the cursor must not
| contain an SQL data change statement if the cursor is defined as either
| SENSITIVE DYNAMIC or SENSITIVE STATIC. When the current value
of a row no longer satisfies the *select-statement* or *statement-name*, that
row is no longer visible through the cursor. When a row of the result
table is deleted from the underlying base table, the row is no longer
visible through the cursor.

In addition, the cursor has sensitivity to changes made to values
outside the cursor (that is, by other cursors or committed changes by
other application processes). If DB2 can not make changes made
outside the cursor visible to the cursor, an error is issued at OPEN
CURSOR. Whether the cursor is sensitive to changes made outside this
cursor depends on whether DYNAMIC or STATIC is in effect for the
cursor and whether SENSITIVE or INSENSITIVE FETCH statements
are used.

Whether the cursor is sensitive to newly inserted rows depends on
whether DYNAMIC or STATIC is in effect for the cursor. The default is
DYNAMIC.

DYNAMIC

Specifies that the result table of the cursor is dynamic in that the
size of the result table can change after the cursor is opened as
rows are inserted into or deleted from the underlying table, and
the order of the rows can change. Inserts, deletes, and updates that
are made by the same application process are immediately visible.
Inserts, deletes, and updates that are made by other application
processes are visible after they are committed.

All FETCH statements for sensitive dynamic cursors are sensitive
to changes made by this cursor, changes made by other cursors in
the same application process, and committed changes made by
other application processes.

If a SENSITIVE DYNAMIC cursor is not possible, an error is
returned, an error is returned.

STATIC

Specifies that the order of the rows and size of the result table is
static. The size of the result table does not grow after the cursor is
opened and the rows are materialized. The order of the rows is
established as the result table is materialized. Rows that are
inserted into the underlying table are not added to the result table
of the cursor regardless of how the rows were inserted. Rows in
the result table do not move if columns in the ORDER BY clause
are updated in rows that have already been materialized.

Whether the changes that are made outside the cursor are visible
to the cursor depends on the type of FETCH that is used with a
SENSITIVE STATIC cursor. For more information, see
Considerations for FETCH statements used with a sensitive static
cursor.

Using a function that is not deterministic (built-in or user-defined)
in the WHERE clause of *select-statement* or *statement-name* of a
SENSITIVE STATIC cursor can cause misleading results. This
occurs because DB2 constructs a temporary result table and
retrieves rows from this table for INSENSITIVE FETCH statements.

When DB2 processes a SENSITIVE FETCH statement, rows are fetched from the underlying table and predicates are re-evaluated if they contain non-correlated subqueries. Using a function that is not deterministic can yield a different result for the re-evaluated query causing the row to no longer be considered a match.

If SENSITIVE STATIC is specified and a sensitive static cursor is not possible, then an error is returned.

If ASENSITIVE, INSENSITIVE, SENSITIVE DYNAMIC, or SENSITIVE STATIC is specified as part of the ATTRIBUTES clause, SCROLL must be specified.

SCROLL or NO SCROLL

Specifies whether the cursor is scrollable.

SCROLL

Specifies that the cursor is scrollable.

NO SCROLL

Specifies that the cursor is not scrollable.

WITHOUT RETURN or WITH RETURN TO CALLER

Specifies the intended use of the result table of the cursor.

WITHOUT RETURN

Specifies that the result table of the cursor is not intended to be used as a result set that will be returned from the program or procedure.

WITH RETURN TO CALLER

Specifies that the result table of the cursor is intended to be used as a result set that will be returned from the program or procedure to the caller. Specifying TO CALLER is optional.

When a cursor that is declared using the WITH RETURN TO CALLER clause remains open at the end of a program or procedure, that cursor defines a result set from the program or procedure. Use the CLOSE statement to close cursors that are not intended to be a result set from the program or procedure. Although DB2 will automatically close any cursors that are not declared using WITH RETURN TO CALLER, the use of the CLOSE statement is recommended to increase the portability of applications. (For Java external procedures, all cursors are implicitly declared WITH RETURN TO CALLER.)

For non-scrollable cursors, the result set consists of all rows from the current cursor position to the end of the result table. For scrollable cursors, the result set consists of all rows of the result table.

The caller is the program or procedure that executed the SQL CALL statement that either invokes a procedure that contains the DECLARE CURSOR statement, or directly or indirectly invokes a program that contains the DECLARE CURSOR statement. For example, if the caller is a procedure, the result set is returned to the procedure. If the caller is a client application, the result set is returned to the client application.

rowset-positioning

Specifies whether rows of data can be accessed as a rowset on a single FETCH statement for this cursor.

WITHOUT ROWSET POSITIONING

Specifies that the cursor can only be used with row positioned FETCH statements.

WITH ROWSET POSITIONING

Specifies that this cursor can be used with rowset positioned or row positioned FETCH statements

fetch-first-clause

Limits the number of rows that can be fetched. It improves the performance of queries with potentially large result sets when only a limited number of rows are needed. If the clause is specified, the number of rows retrieved will not exceed n , where n is the value of the integer. An attempt to fetch $n+1$ rows is handled the same way as normal end of date. The value of integer must be positive and non-zero. The default is 1.

If the OPTIMIZE FOR clause is not specified, a default of "OPTIMIZE FOR integer ROWS" is assumed. If both the FETCH FIRST and OPTIMIZE FOR clauses are specified, the lower of the integer values from these clauses is used to influence optimization and the communications buffer size.

read-only-clause

Declares that the result table is read-only and therefore the cursor cannot be referred to in positioned UPDATE and DELETE statements.

update-clause

Identifies the columns that can updated in a later positioned UPDATE statement. Each column must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect. The clause must not be specified if the result table of the fullselect is read-only. The clause must also not be specified if a created temporary table is referenced in the first FROM clause of the select-statement.

If the clause is specified without a list of columns, the columns that can be updated include all the updatable columns of the table or view that is identified in the first FROM clause of the fullselect.

optimize-clause

Requests special optimization of the *select-statement*. If the clause is omitted, optimization is based on the assumption that all rows of the result table will be retrieved. If the clause is specified, optimization is based on the assumption that the number of rows retrieved will not exceed n , where n is the value of the integer. The clause does not limit the number of rows that can be fetched or affect the result in any way other than performance.

isolation-clause

Specifies the isolation level at which the statement is executed and, for *select-statement*, the type of locks that are acquired. See "isolation-clause" on page 676.

FOR MULTIPLE ROWS or FOR SINGLE ROW

Specifies if a variable number of rows will be provided for a dynamic INSERT or MERGE statement.

FOR MULTIPLE ROWS

Specifies that multiple rows can be provided with host variable arrays on an EXECUTE statement for the statement that is being prepared. FOR MULTIPLE ROWS must only be specified for an INSERT or a MERGE statement.

FOR SINGLE ROW

Specifies that multiple rows must not be provided with host variable arrays on an EXECUTE statement for the statement that is being prepared. FOR SINGLE ROW must only be specified for an INSERT or a MERGE statement.

ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION

Specifies if all rows are inserted as an atomic operation. This clause can only be specified for dynamic INSERT statements.

ATOMIC

Specifies that if the insert for any row fails, all changes that are made to the database by any of the inserts, including changes that are made by successful inserts, are undone. This is the default.

NOT ATOMIC CONTINUE ON SQLEXCEPTION

Specifies that, regardless of the failure of any particular insert of a row, the INSERT statement will not undo any changes that are made to the database by the successful inserts of other rows, and inserting will be attempted for subsequent rows. However, the minimum level of atomicity is at least that of a single insert (that is, it is not possible for a partial insert operation to complete), including any triggers that might have been activated as a result of the INSERT statement.

This clause must not be specified if the INSERT statement is contained within a SELECT statement.

For preparing the MERGE statement, atomicity is specified only on the MERGE statement itself.

SKIP LOCKED DATA

Specifies that the select-statement, searched UPDATE statement (including a searched update operation in a MERGE statement), or searched DELETE statement that is prepared in the PREPARE statement will skip rows when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only when isolation CS or RS is in effect and applies only to row level or page level locks.

SKIP LOCKED DATA can be specified only in the searched UPDATE statement. SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is repeatable read (WITH RR) or uncommitted read (WITH UR).

FROM

Specifies the statement string. The statement string is the value of the specified *string-expression* or the identified *host-variable*.

host-variable

Must identify a host variable that is described in the application program in accordance with the rules for declaring string variables. If the source string is over 32KB in length, the *host-variable* must be a CLOB or DBCLOB variable. The maximum source string length is 2MB although the host variable can be declared larger than 2MB. An indicator variable must not be specified. In PL/I, COBOL and Assembler language, the host variable must be a varying-length string variable. In C, the host variable must not be a NUL-terminated string. In SQL PL, an SQL variable is used in place of a host variable and the value must not be null.

string-expression

string-expression is any PL/I expression that yields a string. *string-expression*

cannot be preceded by a colon. Variables that are within *string-expression* that include operators or functions should not be preceded by a colon. When *string-expression* is specified, the precompiler-generated structures for *string-expression* use an EBCDIC CCSID and an informational message is returned.

Notes

Rules for statement strings: The value of the specified *statement-name* is called the *statement string*. The statement string must be one of the following SQL statements:

ALLOCATE CURSOR	RELEASE SAVEPOINT
ALTER	RENAME
ASSOCIATE LOCATORS	REVOKE
COMMENT	ROLLBACK
COMMIT	SAVEPOINT
CREATE	<i>select-statement</i>
DECLARE GLOBAL	SET CURRENT DEGREE
TEMPORARY TABLE	SET CURRENT DEBUG MODE
DELETE	SET CURRENT DECFLOAT ROUNDING MODE
DROP	SET CURRENT LOCALE LC_CTYPE
EXPLAIN	SET CURRENT MAINTAINED TABLE
FREE LOCATOR	TYPES FOR OPTIMIZATION
GRANT	SET CURRENT OPTIMIZATION HINT
HOLD LOCATOR	SET CURRENT PRECISION
INSERT	SET CURRENT REFRESH AGE
LABEL	SET CURRENT ROUTINE VERSION
LOCK TABLE	SET CURRENT RULES
MERGE	SET CURRENT SQLID
REFRESH TABLE	SET ENCRYPTION PASSWORD
	SET PATH
	SET SCHEMA
	SIGNAL
	TRUNCATE
	UPDATE

The statement string must not:

- Begin with EXEC SQL
- End with END-EXEC or a semicolon
- Include references to variables

Parameter markers: Although a statement string cannot include references to host variables, it can include *parameter markers*. The parameter markers are replaced by the values of host variables when the prepared statement is executed. A parameter marker is a question mark (?) that appears where a host variable could appear if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see the EXECUTE statement, “OPEN” on page 1400, and *DB2 Application Programming and SQL Guide*.

The two types of parameter markers are typed and untyped:

Typed parameter marker

A parameter marker that is specified with its target data type. A typed parameter marker has the general form:

CAST(? AS data-type)

This invocation of a CAST specification is a “promise” that the data type of the parameter at run time will be of the data type that is specified or some data type that is assignable to the specified data type. For example, in the following UPDATE statement, the value of the argument of the TRANSLATE function will be provided at run time:

```
UPDATE EMPLOYEE
  SET LASTNAME = TRANSLATE(CAST(? AS VARCHAR(12)))
  WHERE EMPNO = ?
```

The data type of the value that is provided for the TRANSLATE function will either be VARCHAR(12), or some data type that can be converted to VARCHAR(12). For more information, refer to “Assignment and comparison” on page 102.

Untyped parameter marker

A parameter marker that is specified without its target data type. An untyped parameter marker has the form of a single question mark. The context in which the parameter marker appears determines its data type. For example, in the above UPDATE statement, the data type of the untyped parameter marker in the predicate is the same as the data type of the EMPNO column.

Typed parameter markers can be used in dynamic SQL statements wherever a host variable is supported and the data type is based on the promise made in the CAST specification.

Untyped parameters markers can be used in dynamic SQL statements in selected locations where host variables are supported. Table 126, Table 127 on page 1416, Table 128 on page 1417, and Table 129 on page 1418 show these locations and the resulting data type of the parameter. The tables group the locations into expressions, predicates, functions, and other statements to help show where untyped parameter markers are allowed.

Table 126. Untyped parameter marker usage in expressions (including select list, CASE, and VALUES)

Location of untyped parameter marker	Data type (or error if not supported)
Alone in a select list. For example: SELECT ?	Error
Both operands of a single arithmetic operator, after considering operator precedence and the order of operation rules. Includes cases such as: ? + ? + 10	Error
One operand of a single operator in an arithmetic expression (except datetime arithmetic expressions). Includes cases such as: ? + ? * 10	The data type of the other operand
Any operand of a datetime expression. For example: 'timecol + ?' or '? - datecol'	Error
Labeled duration in a datetime expression	Error
Both operands of a CONCAT operator	Error

Table 126. Untyped parameter marker usage in expressions (including select list, CASE, and VALUES) (continued)

Location of untyped parameter marker	Data type (or error if not supported)
One operand of a CONCAT operator when the other operand is any character data type except CLOB	If the other operand is CHAR(<i>n</i>) or VARCHAR(<i>n</i>), where <i>n</i> is less than 128, the data type is VARCHAR(254 - <i>n</i>). In all other cases, the data type is VARCHAR(254).
One operand of a CONCAT operator when the other operand is any graphic data type except DBCLOB	If the other operand is GRAPHIC(<i>n</i>) or VARGRAPHIC(<i>n</i>), where <i>n</i> is less than 64, the data type is VARGRAPHIC(127 - <i>n</i>). In all other cases, the data type is VARGRAPHIC(127).
One operand of a CONCAT operator when the other operand is any binary type except BLOB	If the other operand is BINARY(<i>n</i>) or VARBINARY(<i>n</i>) where <i>n</i> is less than 128, the data type is VARBINARY(255- <i>n</i>). In all other cases, the data type is VARBINARY(255)
One operand of a CONCAT operator when the other operand is a LOB string	The data type of the other operand (the LOB string)
The <i>expression</i> following the CASE keyword in a simple CASE expression	Error
Any or all <i>expressions</i> following the WHEN keyword in a simple CASE expression	The result of applying the “Rules for result data types” on page 119 to the expression following CASE and the expressions following WHEN that are not untyped parameter markers
A <i>result-expression</i> in any CASE expression when all the other <i>result-expressions</i> are either NULL or untyped parameter markers.	Error
A <i>result-expression</i> in any CASE expression when at least one other <i>result-expression</i> is neither NULL nor an untyped parameter marker.	The result of applying the “Rules for result data types” on page 119 to all the <i>result-expressions</i> that are not NULL or untyped parameter markers
Alone as a <i>column-expression</i> in a single-row VALUES clause that is not within an INSERT statement or the VALUES clause of in insert operation of a MERGE statement	Error
Alone as a <i>column-expression</i> in a single-row VALUES clause within an INSERT statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type
Alone as a <i>column-expression</i> in a <i>values-single-row</i> or <i>values-multiple-row</i> clause of <i>source-table</i> for a MERGE statement	The data type of the column of the source-table, or if the data type is a distinct type, the source data type of the distinct type. The column of the source-table must be referenced elsewhere in the MERGE statement such that its data type can be determined from the context in which it is used, and all such references must resolve to the same data type.
Alone as a <i>column-expression</i> in the VALUES clause of an insert operation of a MERGE statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type
Alone as a <i>column-expression</i> on the right side of <i>assignment-clause</i> for an update operation of a MERGE statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type

Table 126. Untyped parameter marker usage in expressions (including select list, CASE, and VALUES) (continued)

Location of untyped parameter marker	Data type (or error if not supported)
Alone as a <i>column-expression</i> on the right side of a SET clause in an UPDATE statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type

Table 127. Untyped parameter marker usage in predicates

Location of untyped parameter marker	Data type (or error if not supported)
Both operands of a comparison operator	Error
One operand of a comparison operator when the other operand is not an untyped parameter marker	The data type of the other operand. If the operand has a datetime data type, the result of DESCRIBE INPUT will show the data type as CHAR(255) although DB2 uses the datetime data type in any comparisons.
All the operands of a BETWEEN predicate	Error
Two operands of a BETWEEN predicate (either the first and second, or the first and third)	The data type of the operand that is not a parameter marker
Only one operand of a BETWEEN predicate	The result of applying the “Rules for result data types” on page 119 on the other operands that are not parameter markers
All the operands of an IN predicate, for example, ? IN (?, ?, ?)	Error
The first and second operands of an IN predicate, for example, ? IN (?, A, B)	The result of applying the “Rules for result data types” on page 119 on the operands in the IN list that are not parameter markers
The first operand of an IN predicate and zero or more operands of the IN list except for the first operand of the IN list, for example, ? IN (A, ?, B, ?)	The result of applying the “Rules for result data types” on page 119 on the operands in the IN list that are not parameter markers
The first operand of an IN predicate when the right side is a fullselect of fullselect, for example, ? IN (fullselect)	The data type of the selected column
Any or all operands of the IN list of the IN predicate and the first operand of the IN predicate is not an untyped parameter marker, for example, A IN (?, A, ?)	The data type of the first operand (the operand on the left side of the IN list)
All the operands of a LIKE predicate	The first and second operands (<i>match-expression</i> and <i>pattern-expression</i>) are VARCHAR(4000). The third operand (<i>escape-expression</i>) is VARCHAR(1).
The first operand (the <i>match-expression</i>) when at least one other operand (the <i>pattern-expression</i> or <i>escape-expression</i>) is not an untyped parameter marker.	VARCHAR(4000), VARGRAPHIC(2000), or VARBINARY(4000), depending on the data type of the first operand that is not an untyped parameter marker

Table 127. Untyped parameter marker usage in predicates (continued)

Location of untyped parameter marker	Data type (or error if not supported)
The second operand (the <i>pattern-expression</i>) when at least one other operand (the <i>match-expression</i> or <i>escape-expression</i>) is not an untyped parameter marker. When the pattern specified in a LIKE predicate is a parameter marker and a fixed-length character host variable is used to replace the parameter marker, specify a value for the host variable that is the correct length. If you do not specify the correct length, the select does not return the intended results.	VARCHAR(4000), VARGRAPHIC(2000), or VARBINARY(4000), depending on the data type of the first operand that is not an untyped parameter marker.
The third operand (the <i>escape-expression</i>) when at least one other operand (the <i>match-expression</i> or <i>pattern-expression</i>) is not an untyped parameter marker	CHAR(1), GRAPHIC(1), or BINARY(1), depending on the data type of the first operand that is not an untyped parameter marker
Operand of a NULL predicate	Error

Table 128. Untyped parameter marker usage in functions

Location of untyped parameter marker	Data type (or error if not supported)
All arguments of COALESCE or NULLIF	Error
Any argument of COALESCE or NULLIF when at least one other argument is not an untyped parameter marker	The result of applying the “Rules for result data types” on page 119 on the arguments that are not untyped parameter markers, the data type of the other argument
First argument of COLLATION_KEY	VARGRAPHIC(2000)
Second argument of COLLATION_KEY	VARCHAR(255)
First argument of LOWER	VARCHAR(4000)
Second argument of LOWER	VARCHAR(255)
Any argument other than the first argument of MAX	The data type of the corresponding parameter in the function instance
Any argument other than the first argument of MIN	The data type of the corresponding parameter in the function instance
Both arguments of POSSTR or POSITION	VARCHAR(4000) for both arguments
One argument of POSSTR or POSITION when the other argument is a character data type	VARCHAR(4000)
One argument of POSSTR or POSITION when the other argument is a graphic data type	VARGRAPHIC(2000)
One argument of POSSTR or POSITION when the other argument is a BINARY or VARBINARY data type	VARBINARY(4000)
One argument of POSSTR or POSITION when the other argument is a BLOB	BLOB(4000)
First argument of SUBSTR or SUBSTRING	VARCHAR(4000)
Second or third argument of SUBSTR or SUBSTRING	INTEGER
One argument of TIMESTAMP	TIME

Table 128. Untyped parameter marker usage in functions (continued)

Location of untyped parameter marker	Data type (or error if not supported)
First argument of <code>TIMESTAMP_FORMAT</code>	<code>VARCHAR(255)</code>
First argument of <code>TRANSLATE</code>	Error
Second or third argument of <code>TRANSLATE</code>	<code>VARCHAR(4000)</code> , <code>VARGRAPHIC(2000)</code> , depending on whether the data type of the first argument is character or graphic
Fourth argument of <code>TRANSLATE</code>	<code>VARCHAR(1)</code> or <code>VARGRAPHIC(1)</code> , depending on whether the data type of the first argument is character or graphic
First argument of <code>UPPER</code>	<code>VARCHAR(4000)</code>
Second argument of <code>UPPER</code>	<code>VARCHAR(255)</code>
First argument of <code>VARCHAR_FORMAT</code>	<code>TIMESTAMP</code>
Unary minus	<code>DOUBLE PRECISION</code>
Unary plus	<code>DOUBLE PRECISION</code>
The argument of any built-in scalar function (except those that are described in this table)	Error
The argument of a built-in aggregate function	Error
The argument of a user-defined scalar function, user-defined aggregate function, or user-defined table function	The data type of the corresponding parameter in the function instance

Table 129. Untyped parameter marker usage in statements

Location of untyped parameter marker	Data type (or error if not supported)
<code>FOR n ROWS</code> clause of an <code>INSERT</code> or <code>MERGE</code> statement	Integer
The value on the right side of a <code>SET</code> clause in an <code>UPDATE</code> statement or the <code>UPDATE</code> clause of the <code>MERGE</code> statement	The data type of the column of the source-table, or if the column is defined as a distinct type, the source data type of the distinct type. The column of the source-table must be referenced elsewhere in the <code>MERGE</code> statement such that its data type can be determined from the context in which it is used, and all such references must resolve to the same data type.

Considerations for `FETCH` statements used with a sensitive static cursor: Whether changes made outside the cursor are visible to the cursor depends on the type of `FETCH` that is used with a `SENSITIVE STATIC` cursor:

- A `SENSITIVE FETCH` is sensitive to all updates and deletes that are made by this cursor (including changes made by triggers) and committed updates and deletes by all other application processes because every fetched row is retrieved from the underlying base table and not a temporary table. This is the default type of `FETCH` statement for a `SENSITIVE` cursor.

Changes that are made to the underlying data using this cursor result in an automatic refresh of the row. The changes that are made using this type of cursor can result in holes in the result table of the cursor. In addition, re-fetching rows (fetching rows that have already been retrieved) can result in holes in the result table. If a sensitive `FETCH` is issued to re-fetch a row and the row no

longer qualifies for the search condition of the query, it results in a "delete hole" or an "update hole". In this case, no data is returned, and the cursor is left positioned on the hole.

- An INSENSITIVE FETCH is not sensitive to updates and deletes that are made outside this cursor; however, it is sensitive to all updates and deletes that are made by this cursor. Changes that made with triggers are not visible with an INSENSITIVE FETCH until the content of the rows are updated in the result table with a SENSITIVE FETCH statement. If an application does not want to be sensitive to changes that are made outside this cursor (that is, the application does not want to see changes made either with another cursor or by another application process), INSENSITIVE can be explicitly specified as part of the FETCH statement for a SENSITIVE STATIC cursor. This type of FETCH is useful for refreshing data in user data buffers. For more information, see INSENSITIVE.

Error checking: When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, a prepared statement is not created and the error condition that prevents its creation is reported in the SQLCA.

In local and remote processing, the DEFER(PREPARE) and REOPT(ALWAYS)/REOPT(ONCE) bind options can cause some SQL statements to receive "delayed" errors. For example, DESCRIBE, EXECUTE, and OPEN might receive an SQLCODE that normally occurs during PREPARE processing.

Reference and execution rules: Prepared statements can be referred to in the following kinds of statements, with the following restrictions shown:

In... The prepared statement...

DESCRIBE

has no restrictions

DECLARE CURSOR

must be SELECT when the cursor is opened

EXECUTE

must *not* be SELECT

A prepared statement can be executed many times. Indeed, if a prepared statement is not executed more than once and does not contain parameter markers, it is more efficient to use the EXECUTE IMMEDIATE statement rather than the PREPARE and EXECUTE statements.

Prepared statement persistence: All prepared statements created by a unit of work are destroyed when the unit of work is terminated, with the following exceptions:

- A SELECT statement whose cursor is declared with the option WITH HOLD persists over the execution of a commit operation if the cursor is open when the commit operation is executed.
- SELECT, INSERT, UPDATE, MERGE, and DELETE statements that are bound with KEEP DYNAMIC(YES) are kept past the commit operation if your system is enabled for dynamic statement caching, and none of the following are true:
 - SQL RELEASE has been issued for the site
 - Bind option DISCONNECT(AUTOMATIC) was used
 - Bind option DISCONNECT(CONDITIONAL) was used and there are no hold cursors for the site

Scope of a statement name: The scope of a *statement-name* is the same as the scope of a *cursor-name*. See “DECLARE CURSOR” on page 1191 for more information about the scope of a *cursor-name*.

Preparation with PREPARE INTO and REOPT bind option: If bind option REOPT(ALWAYS) or REOPT(ONCE) is in effect, PREPARE INTO is equivalent to a PREPARE and a DESCRIBE being performed. If a statement has input variables, the DESCRIBE causes the statement to be prepared with default values, and the statement must be prepared again when it is opened or executed. When REOPT(ONCE) is in effect, the statement is always prepared twice even if there are no input variables. Therefore, to avoid having a statement prepared twice, avoid using PREPARE INTO when REOPT(ALWAYS) or REOPT(ONCE) is in effect.

Relationship of cursor attributes on PREPARE statements and SELECT or DECLARE CURSOR statements: Cursor attributes that are specified as part of the *select-statement* are used instead of any corresponding options that specified with the ATTRIBUTES clause on PREPARE. Attributes that are specified as part of the ATTRIBUTES clause of PREPARE take precedence over any corresponding option that is specified with the DECLARE CURSOR statement. The order for using cursor attributes is as follows:

- SELECT (highest priority)
- PREPARE statement ATTRIBUTES clause
- DECLARE CURSOR (lowest priority)

For example, assume that host variable MYQ has been set to the following SELECT statement:

```
SELECT WORKDEPT, EMPNO, SALARY, BONUS, COMM
FROM EMP
WHERE WORKDEPT IN ('D11', 'D21')
FOR UPDATE OF SALARY, BONUS, COMM
```

If the following PREPARE statement were issued, then the FOR UPDATE clause specified as part of the SELECT statement would be used instead of the FOR READ ONLY clause specified with the ATTRIBUTES clause as part of the PREPARE statement. Thus, the cursor would be updatable.

```
attrstring = 'FOR READ ONLY';
EXEC SQL PREPARE stmt1 ATTRIBUTES :attrstring FROM :MYQ;
```

Examples

Example 1: In this PL/I example, an INSERT statement with parameter markers is prepared and executed. Before execution, values for the parameter markers are read into the host variables S1, S2, S3, S4, and S5.

```
EXEC SQL PREPARE DEPT_INSERT FROM
'INSERT INTO DSN8910.DEPT VALUES(?,?,?,?)';
-- Check for successful execution and read values into host variables
EXEC SQL EXECUTE DEPT_INSERT USING :S1, :S2, :S3, :S4, :S5;
```

Example 2: Prepare a dynamic SELECT statement specifying the attributes of the cursor with a host variable on the PREPARE statement. Assume that the text of the SELECT statement is in a variable named stmttxt, and that the desired attributes of the cursor are in a variable named attrvar.

```
EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
EXEC SQL PREPARE mystmt ATTRIBUTES :attrvar
FROM :stmttxt;
```

```
EXEC SQL DESCRIBE mystmt INTO :mysqlda;  
EXEC SQL OPEN mycursor;  
EXEC SQL FETCH FROM mycursor USING DESCRIPTOR :mysqlda;
```

REFRESH TABLE

The REFRESH TABLE statement refreshes the data in a materialized query table. The statement deletes all rows in the materialized query table, executes the fullselect in the table definition to recalculate the data from the tables specified in the fullselect, inserts the calculated result into the materialized query table, and updates the catalog for the refresh timestamp and cardinality of the table. The table can exist at the current server or at any DB2 subsystem with which the current server can establish a connection.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privilege set for REFRESH TABLE must include at least one of the following authorities:

- Ownership of the materialized query table
- DBADM or DBCTRL authority on the database that contains the materialized query table
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statements dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke). For more information on these behaviors, including a list of the DYNAMICRULES bind option values, see “Authorization IDs and dynamic SQL” on page 64.

Syntax

►►—REFRESH TABLE—*table-name*—

—QUERYNO—*integer*—

—◄◄

Description

table-name

Identifies the table to be refreshed. The name must identify a materialized query table. REFRESH TABLE evaluates the fullselect in the *materialized-query-definition* clause to refresh the table. The isolation level for the fullselect is the isolation level of the materialized query table recorded when CREATE TABLE or ALTER TABLE was issued.

QUERYNO *integer*

Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan

table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Notes

Automatic query rewrite using materialized query tables is not attempted for the fullselect in the materialized query table definition during the processing of REFRESH TABLE statement.

After successful execution of a REFRESH TABLE statement, the SQLCA field SQLERRD(3) will contain the number of rows inserted into the materialized query table.

The EXPLAIN output for REFRESH TABLE *table-name* is the same as the EXPLAIN output for INSERT INTO *table-name fullselect* where *fullselect* is from the materialized query table definition.

The REFRESH TABLE statement is supported over DRDA protocol, but not supported over DB2 private protocol.

If the materialized query table has a security label column, the REFRESH TABLE statement does not do any checking for multilevel security with row-level granularity when it deletes and repopulates the data in the table by executing the fullselect. Instead, DB2 performs the checking for multilevel security with row-level granularity when the materialized query table is exploited in automatic query rewrite or is used directly.

The REFRESH TABLE statement can be used to remove a table space from the logical page list and reset recover-pending status. This can only be done by using REFRESH TABLE to repopulate a materialized query table where the materialized query table is the only table in the table space.

Example

Issue a statement to refresh the content of a materialized query table that is named SALESCOUNT. The statement recalculates the data from the fullselect that was used to define SALESCOUNT and refreshes the content of SALESCOUNT with the recalculated results.

```
REFRESH TABLE SALESCOUNT;
```

RELEASE (connection)

The RELEASE (connection) statement places one or more connections in the release pending state.

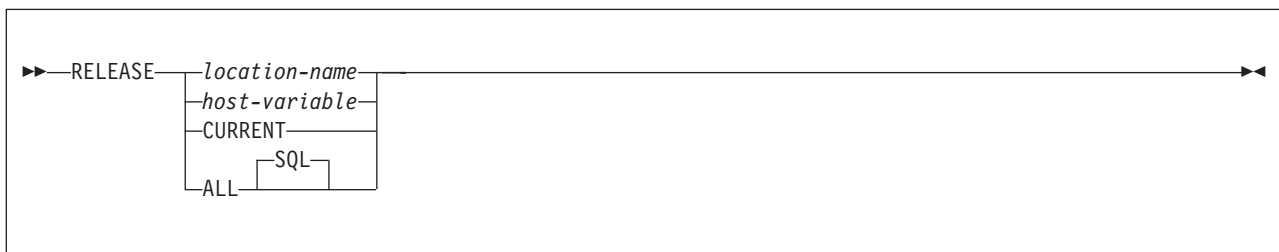
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required.

Syntax



Description

location-name **or** *host-variable*

Identifies an SQL connection or a DB2 private connection by the specified location name or the location name contained in the host variable. If a host variable is specified:

- It must be a character string variable with a length attribute that is not greater than 16. (A C NUL-terminated character string can be up to 17 bytes.)
- It must not be followed by an indicator variable.
- The location name must be left-justified within the host variable and must conform to the rules for forming an ordinary location identifier.
- If the length of the location name is less than the length of the host variable, it must be padded on the right with blanks.

The specified location name or the location name contained in the host variable must identify an existing SQL connection or DB2 private connection of the application process.

CURRENT

Identifies the current SQL connection of the application process. The application process must be in the connected state.

ALL or ALL SQL

Identifies all existing connections (including local, and SQL, and DB2 private connections) of the application process. An error or warning does not occur if no connections exist when the statement is executed.

If the RELEASE (connection) statement is successful, each identified connection is placed in the release-pending state and, therefore, will be ended during the next

commit operation. If the RELEASE (connection) statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.

Notes

RELEASE and CONNECT (Type 1): Using CONNECT (Type 1) semantics does not prevent using RELEASE (connection).

Scope of RELEASE: RELEASE (connection) does not close cursors, does not release any resources, and does not prevent further use of the connection.

Resource considerations for remote connections: Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be in the release pending status and one that is going to be reused should not be in the release pending status. Remote connections can also be ended during a commit operation as a result of the DISCONNECT(AUTOMATIC) or DISCONNECT(CONDITIONAL) bind option.

If the current SQL connection is in the release pending status when a commit operation is performed, the connection is ended and the application process is in the unconnected state. In this case, the next executed SQL statement should be CONNECT or SET CONNECTION.

Connection states: ROLLBACK does not reset the state of a connection from release pending to held.

If the current SQL connection is in the release pending state when a commit operation is performed, the connection is ended and the application process is in the unconnected state. In this case, the next executed SQL statement must be CONNECT or SET CONNECTION.

For further information, see “Application process connection states” on page 34.

Location names CURRENT and ALL: A database server named CURRENT or ALL can only be identified by a host variable or a delimited identifier. A connection in the release pending state is ended during a commit operation even though it has an open cursor defined with WITH HOLD.

Encoding scheme of a host variable: If the RELEASE statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

ALL PRIVATE: ALL PRIVATE can continue to be specified as in prior releases. The clause was introduced and was intended for use only with DB2 private protocol access. ALL PRIVATE specifies that all existing DB2 private connections of the application process are to be released. An error or warning does not occur if no DB2 private connections exist when the statement is executed.

Examples

Example 1: The SQL connection to TOROLAB1 is not needed in the next unit of work. The following statement causes it to be ended during the next commit operation:

```
EXEC SQL RELEASE TOROLAB1;
```

Example 2: The current SQL connection is not needed in the next unit of work. The following statement causes it to be ended during the next commit operation:

```
EXEC SQL RELEASE CURRENT;
```

Example 3: The first phase of an application involves explicit CONNECTs to remote servers and the second phase involves the use of DB2 private protocol access with the local DB2 subsystem as the server. None of the existing connections are needed in the second phase and their existence could prevent the allocation of DB2 private connections. Accordingly, the following statement is executed before the commit operation that separates the two phases:

```
EXEC SQL RELEASE ALL SQL;
```

RELEASE SAVEPOINT

The RELEASE SAVEPOINT statement releases the identified savepoint and any subsequently established savepoints within a unit of recovery.

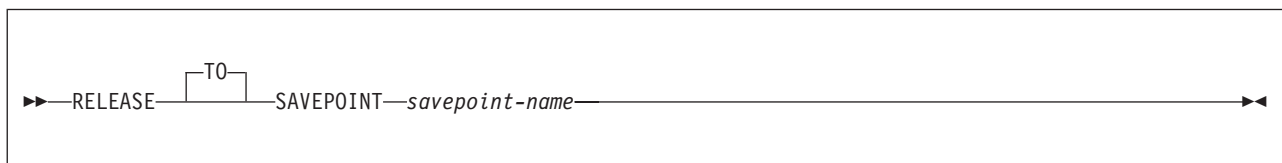
Invocation

This statement can be imbedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

savepoint-name

| Identifies the savepoint to release. If the named savepoint does not exist, an
| error occurs. The name must identify a savepoint that exists at the current
| server. After a savepoint is released, it is no longer maintained and rollback to
| the savepoint is no longer possible.

Notes

Savepoint names: The name of the savepoint that was released can be reused in another SAVEPOINT statement, regardless of whether the UNIQUE keyword was specified on an earlier SAVEPOINT statement that specified this same savepoint name.

Example

Assume that a main routine sets savepoint A and then invokes a subroutine that sets savepoints B and C. When control returns to the main routine, release savepoint A and any subsequently set savepoints. Savepoints B and C, which were set by the subroutine, are released in addition to A.

```

:
  RELEASE SAVEPOINT A;
```

RENAME

The RENAME statement renames an existing table or index.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

To rename a table, the privilege set that is defined below must include at least one of the following privileges:

- Ownership of the table
- DBADM, DBCTRL, or DBMAINT authority for the database that contains the table
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

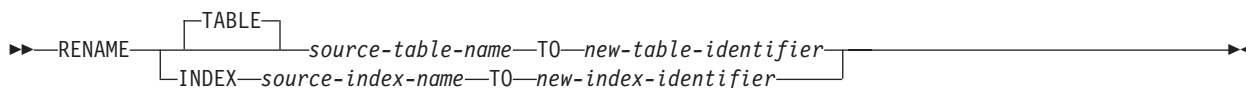
To rename an index, the privilege set that is defined below must include at least one of the following privileges:

- Ownership of the table for which the index is defined
- Ownership of the index that is being renamed
- DBADM, DBCTRL, or DBMAINT authority for the database that contains the index
- SYSADM or SYSCTRL authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

Syntax



Description

source-table-name

Identifies the existing table that is to be renamed. The name, including the implicit or explicit qualifier, must identify a table that exists at the current server. The name must not identify a declared temporary table, a catalog table, an active resource limit specification table, a materialized query table, a clone

table, a table with a trigger defined on it, a view, or a synonym. If you specify a three-part name or alias for the source table, the source table must exist at the current server. If any view definitions or materialized query table definitions currently refer to the source table, an error occurs.

new-table-identifier

Specifies the new name for the table without a qualifier. The qualifier of the *source-table-name* is used to qualify the new name for the table. The qualified name must not identify a table, view, alias, or synonym that exists at the current server.

source-index-name

Identifies the existing index that is to be renamed. The name, including an implicit or explicit qualifier, must identify an index that exists at the current server. The name must not identify a system defined catalog index, an index for a declared temporary table, or an index for an active resource limit specification table.

new-index-identifier

Specifies that new name for the index without a qualifier. The qualifier of the *source-index-name* is used to qualify the new name for the index. The qualified name must not identify an index that exists at the current server.

Notes

Effects of the statement: The specified table or index is renamed to the new name. For a renamed table, all privileges and indexes on the table are preserved. For a renamed index, all privileges are preserved.

Invalidation of plans and packages: When any table except an auxiliary table is renamed, plans and packages that refer to that table are invalidated. When an auxiliary table is renamed, plans and packages that refer to the auxiliary table are not invalidated.

Considerations for aliases: If an alias name is specified for *table-name*, the table must exist at the current server, and the table that is identified by the alias is renamed. The name of the alias is not changed and continues to refer to the old table name after the rename.

Changing the name of an alias with the RENAME statement is not supported. To change the name to which an alias refers, you must drop the alias and then recreate it.

Considerations for plan tables: The RENAME INDEX statement does not update the contents of a plan table. Rows that exist in a plan table that are generated from a EXPLAIN statement can contain the name of an index in the access path selections. When an index is renamed, any entries in existing plan tables that refer to the old index name are not updated.

Transfer of authorization, referential integrity constraints, and indexes: All authorizations associated with the source table name are *transferred* to the new (target) table name. The authorization catalog tables are updated appropriately.

Referential integrity constraints involving the source table are updated to refer to the new table. The catalog tables are updated appropriately.

Indexes that are defined for the source table are *transferred* to the new table. The index catalog tables are updated appropriately.

Object identifier: Renamed tables and indexes keep the same object identifier as the original table or index.

Renaming registration tables: If an application registration table (ART) or object registration table (ORT) or an index of an ART or ORT is specified as the source table for RENAME, when RENAME completes, it is as if that table had been dropped. There is no ART or ORT once the ART or ORT table has been renamed.

Catalog table updates: Entries in the following catalog tables are updated to reflect the new table:

- SYSAUXRELS
- SYSCHECKS
- SYSCHECKS2
- SYSCHECKDEP
- SYSCOLAUTH
- SYSCOLDIST
- SYSCOLDIST_HIST
- SYSCOLDISTSTATS
- SYSCOLSTATS
- SYSCOLUMNS
- SYSCOLUMNS_HIST
- SYSCONSTDEP
- SYSFIELDS
- SYSFOREIGNKEYS
- SYSINDEXES
- SYSINDEXES_HIST
- SYSKEYCOLUSE
- SYSPLANDEP
- SYSPACKDEP
- SYSRELS
- SYSSEQUENCESDEP
- SYSSYNONYMS
- SYSTABAUTH
- SYSTABCONST
- SYSTABLES
- SYSTABLES_HIST
- SYSTABSTATS
- SYSTABSTATS_HIST

Entries in SYSSTMT and SYSPACKSTMT are not updated.

Entries in the following catalog tables are updated to reflect the new index:

- SYSDEPENDENCIES
- SYSINDEXES
- SYSINDEXES_HIST

- SYSINDEXESPART
- SYSINDEXESPART_HIST
- SYSINDEXSPACESTATS
- SYSINDEXSTATS
- SYSINDEXSTATS_HIST
- SYSKEYS
- SYSKEYTARGETS
- SYSKEYTARGETS_HIST
- SYSKEYTARGETSTATS
- SYSKEYTGTDIST
- SYSKEYTGTDIST_HIST
- SYSKEYTGTDISTSTATS
- SYSOBJROLEDEP
- SYSPACKDEP
- SYSPLANDEP
- SYSRELS
- SYSTABCONST
- SYSTABLEPART
- SYSVTREE

Examples

Example 1: Change the name of the EMP table to EMPLOYEE:

```
RENAME TABLE EMP TO EMPLOYEE;
```

Example 2: Change the name of the EMP_USA_HIS2002:

```
RENAME TABLE EMP_USA_HIS2002 TO EMPLOYEE_UNITEDSTATES_HISTORY2002;
```

Example 3: Change the name of the EMPINDX1 to EMPLOYEE_INDEX:

```
RENAME INDEX COMPANY.EMPINDX1 TO EMPLOYEE_INDEX;
```

REVOKE

The REVOKE statement revokes privileges from authorization IDs. There is a separate form of the statement for each of these classes of privilege:

- Collection
- Database
- Distinct type
- Function or stored procedure
- Package
- Plan
- Schema
- Sequence
- System
- Table or view
- Use

The applicable objects are always at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

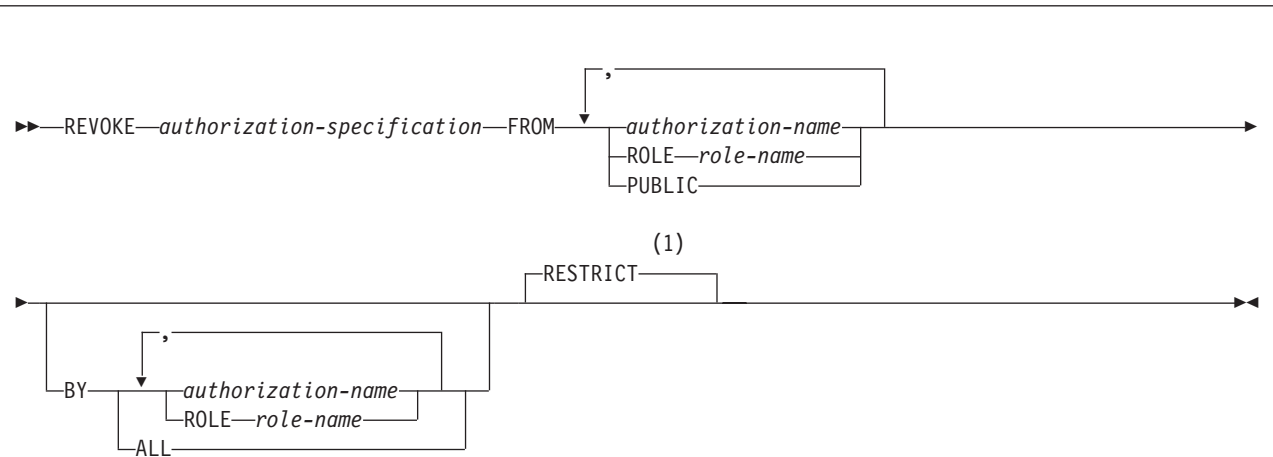
If the authorization mechanism was not activated when the DB2 subsystem was installed, an error condition occurs.

Authorization

If the BY clause is not specified, the authorization ID of the statement must have granted at least one of the specified privileges to every *authorization-name* specified in the FROM clause (including PUBLIC, if specified). If the BY clause is specified, the authorization ID of the statement must have SYSADM or SYSCTRL authority.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. The owner can be a role. If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. However, if the process is running in a trusted context that is defined with the ROLE AS OBJECT OWNER AND QUALIFIER CLAUSE, the privilege set is the privileges that are held by the role that is in effect.

Syntax



Notes:

- 1 The RESTRICT clause is the default only for the forms of the REVOKE statement that allow it.

Description

authorization-specification

Specifies one or more privileges for the class of privilege. The same privilege must not be specified more than once.

FROM

Specifies from what authorization IDs the privileges are revoked.

authorization-name,...

Lists one or more authorization IDs. Do not use the same authorization ID more than one time. If the *authorization-name* is specified in lowercase, it must be delimited using double quotes.

The value of CURRENT RULES determines if you can use the ID of the REVOKE statement itself (to revoke privileges from yourself). When CURRENT RULES is:

DB2 You cannot use the ID of the REVOKE statement.

STD

You can use the ID of the REVOKE statement.

ROLE *role-name*

Lists one or more roles. Do not specify the same role more than one time.

PUBLIC

Revokes a grant of privileges to PUBLIC.

BY Lists grantors who have granted privileges and revokes each named privilege that was explicitly granted to some named user by one of the named grantors. Only an authorization ID or role with SYSADM or SYSCTRL authority can use BY, even if the authorization ID or role names only itself in the BY clause.

authorization-name,...

Lists one or more authorization IDs of users who were the grantors of the privileges named. Do not use the same authorization ID more than once. Each grantor that is listed must have explicitly granted some named privilege to all of the named users or roles.

ROLE *role-name*

Lists one or more roles that were the grantors of the privileges named. Do not specify the same role more than one time. Each grantor that is listed must have explicitly granted some named privilege to all of the named users or roles.

ALL

Revokes each named privilege from all named users who were explicitly granted the privilege, regardless of who granted it.

RESTRICT

Prevents the named privilege from being revoked when certain conditions apply. RESTRICT is the default only for the forms of the REVOKE statement that allow it. These forms are revoking the USAGE privilege on distinct types, the EXECUTE privilege on user-defined functions and stored procedures, and the USAGE privilege on sequences.

Notes

Revoked privileges: The privileges revoked from an authorization ID or a role are those that are identified in the statement and which were granted to the user by the grantor. Other privileges can be revoked as the result of a cascade revoke. For more on DB2 privileges, see *DB2 Administration Guide*.

Cascade revoke: Revoking a privilege from a user can also cause that privilege to be revoked from other users. This is called a *cascade revoke*. The following rules must be true for privilege P' to be revoked from U3 when U1 revokes privilege P from U2:

- P and P' are the same privilege.
- U2 granted privilege P' to U3.
- No one granted privilege P to U2 prior to the grant by U1.
- U2 does not have installation SYSADM authority.

The rules also apply to the implicit grants that are made as a result of a CREATE VIEW statement.

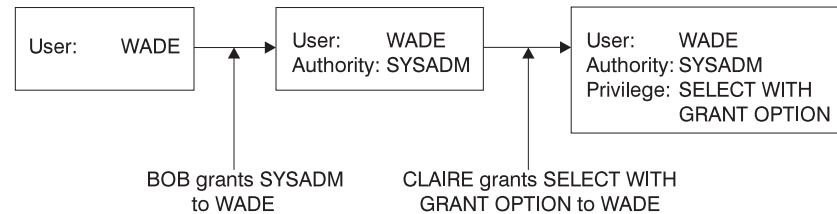
Cascade revoke does not occur under any of the following conditions:

- The privilege was granted by a current install SYSADM.
- The privilege is the USAGE privilege on a distinct type and the revokee owns any of these items:
 - A user-defined function or stored procedure that uses the distinct type
 - A table that has a column that uses the distinct type
 - A sequence whose data type is the distinct type
- The privilege is the USAGE privilege on a sequence and the revokee owns any of these items:
 - A trigger that has a NEXT VALUE or PREVIOUS VALUE expression that specifies the sequence
 - An inline SQL function that has a NEXT VALUE or PREVIOUS VALUE expression in the function body that specifies the sequence
- The privilege is the EXECUTE privilege on a user-defined function and the revokee owns any of these items:
 - A user-defined function that is sourced on the function
 - A view that uses the function

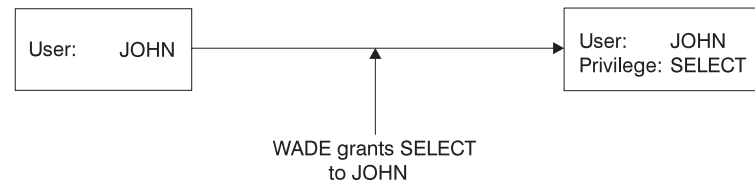
- A trigger package that uses the function
- A table that uses the function in a check constraint or a user-defined default type
- The privilege is the EXECUTE privilege on a stored procedure and the revokee owns any of these items:
 - A trigger package that refers to the stored procedure in a CALL statement.

Refer to the diagrams for the following example:

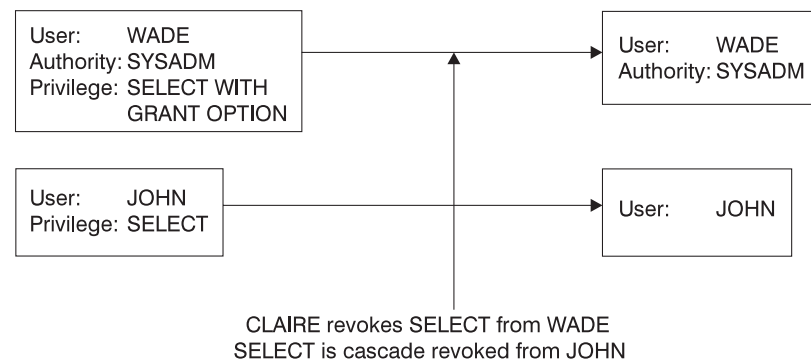
1. Suppose BOB grants SYSADM authority to WADE. Later, CLAIRE grants the SELECT privilege on a table with the WITH GRANT OPTION to WADE.



2. WADE grants the SELECT privilege to JOHN on the same table.



3. When CLAIRE revokes the SELECT privilege on the table from WADE, the SELECT privilege on that table is also revoked from JOHN.



The grant from WADE to JOHN is removed because WADE had not been granted the SELECT privilege from any other source before CLAIRE made the grant. The SYSADM authority granted to WADE from BOB does not affect the cascade revoke. For more on SYSADM and install SYSADM authority, see *DB2 Administration Guide*. For another example of cascading revokes, see *DB2 Administration Guide*.

Revoking a SELECT privilege that was exercised to create a view or materialized query table causes the view to be dropped, unless the owner of the view was directly granted the SELECT privilege from another source before the view was created. Revoking a SYSADM privilege that was required to create a view causes

the view to be dropped. For details on when SYSADM authority is required to create a view, see *Authorization* in “CREATE VIEW” on page 1184.

Invalidation of plans and packages: A revoke or cascaded revoke of any privilege or role that was exercised to create a plan or package makes the plan or package invalid when the revokee no longer holds the privilege from any other source. Corresponding authorization caches are cleared even if the revokee has the privilege from any other source.³⁶

Inoperative plans and packages: A revoke or cascaded revoke of the EXECUTE privilege on a user-defined function that was exercised to create a plan or package makes the plan or package inoperative and causes the corresponding authorization caches to be cleared when the revokee no longer holds the privilege from any other source.³⁶

Privileges belonging to an authority: You can revoke an administrative authority, but you cannot separately revoke the specific privileges inherent in that administrative authority.

Let P be a privilege inherent in authority X. A user with authority X can also have privilege P as a result of an explicit grant of P. In this case:

- If X is revoked, the user still has privilege P.
- If P is revoked, the user still has the privilege because it is inherent in X.

Revoking of privileges in a trusted context: Revokes that are made in a trusted context that is defined with the ROLE AS OBJECT OWNER clause result in the revoker being the role in effect. If the statement is prepared dynamically, the revoker is the role that is associated with the user that is running the statement. If the statement is embedded in a program, the revoker is the owner of the plan or package. If the ROLE AS OBJECT OWNER clause is not specified for the trusted context, the revoker is the authorization ID of the process.

Ownership privileges: The privileges inherent in the ownership of an object cannot be revoked.

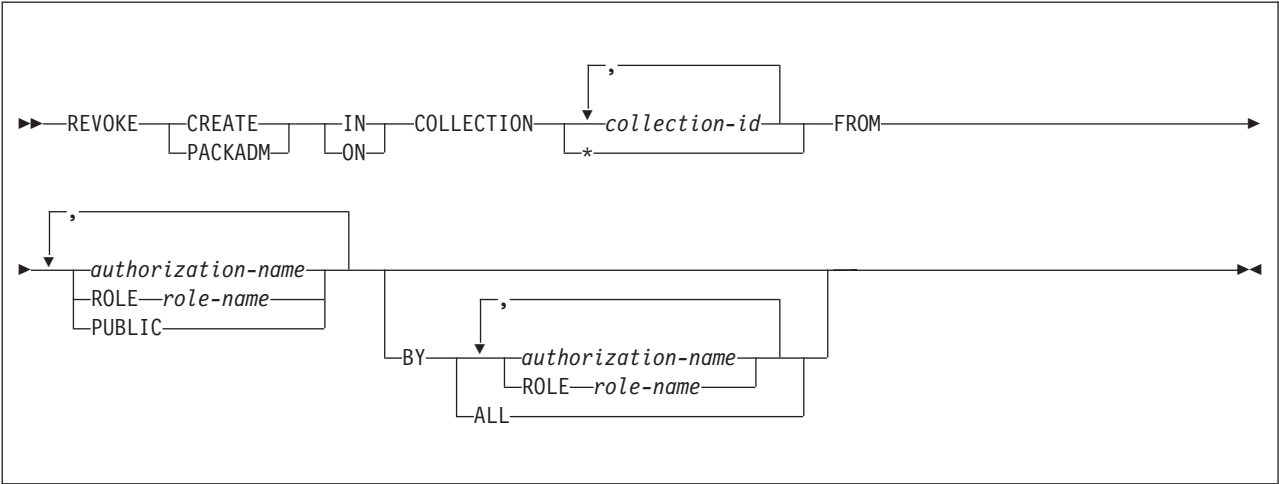
PUBLIC AT ALL LOCATIONS: PUBLIC AT ALL LOCATIONS can continue to be specified as an alternative to PUBLIC as in prior releases. PUBLIC AT ALL LOCATIONS was introduced and was intended for use only with DB2 private protocol access. PUBLIC AT ALL LOCATIONS revokes a grant of the specified privileges from PUBLIC at all locations.

³⁶ Dependencies on stored procedures can be checked only if the procedure name is specified as a constant and not via a host variable in the CALL statement.

REVOKE (collection privileges)

This form of the REVOKE statement revokes privileges on collections.

Syntax



Description

CREATE IN

Revokes the privilege to use the BIND subcommand to create packages in the designated collections.

The word ON can be used instead of IN.

PACKADM ON

Revokes the package administrator authority for the designated collections.

The word IN can be used instead of ON.

COLLECTION *collection-id*,...

Identifies the collections on which the specified privilege is revoked. For each identified collection, you (or the indicated grantors) must have granted the specified privilege on that collection to all identified users (including PUBLIC if specified). The same collection must not be identified more than once.

COLLECTION *

Indicates that the specified privilege on COLLECTION * is revoked. You (or the indicated grantors) must have granted the specified privilege on COLLECTION * to all identified users (including PUBLIC if specified). Privileges granted on specific collections are not affected.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

Examples

Example 1: Revoke the privilege to create new packages in collections QAACLONE and DSN8CC61 from CLARK.

```
REVOKE CREATE IN COLLECTION QAACLONE, DSN8CC61 FROM CLARK;
```

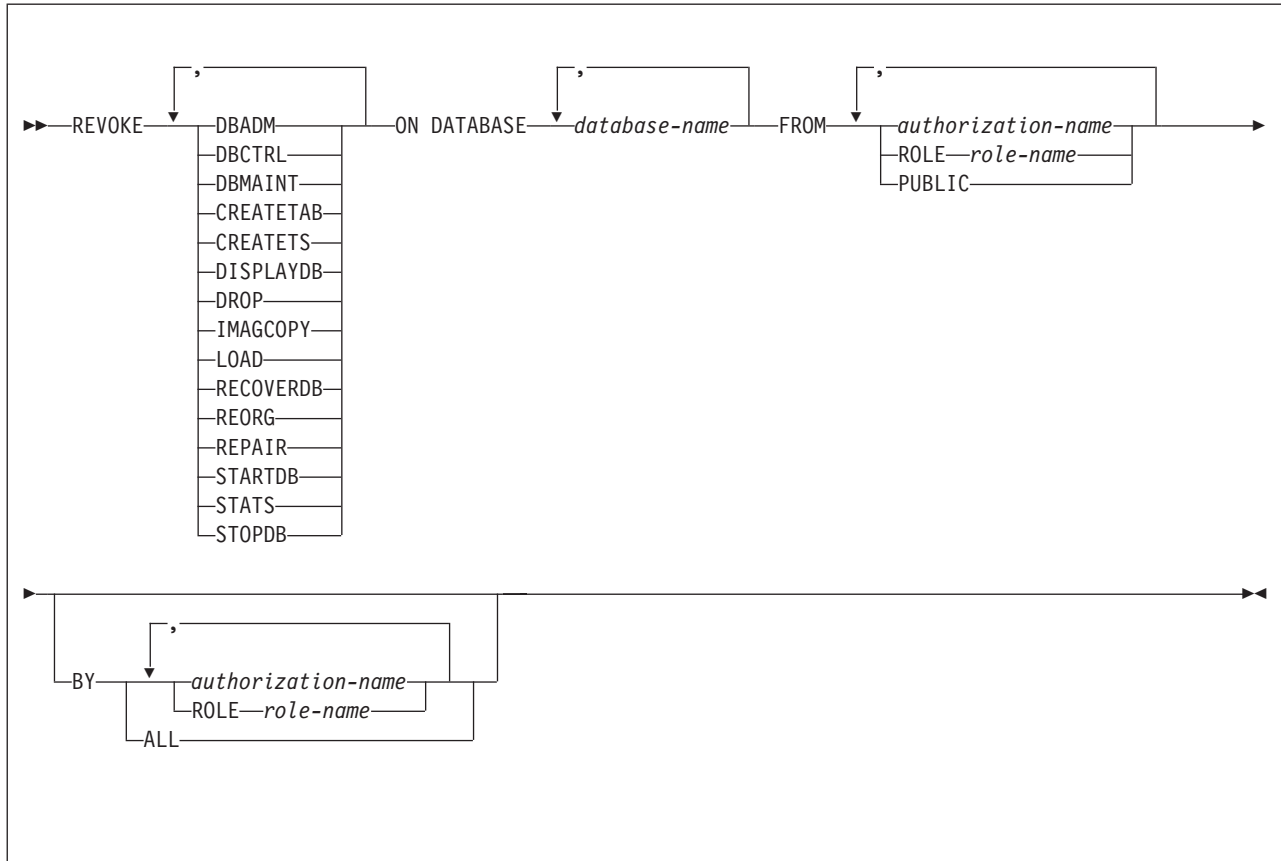
| *Example 2:* Revoke the privilege to create new packages in collections DSN8CC91
| from role ROLE1:

| REVOKE CREATE IN COLLECTION DSN8CC91 FROM ROLE ROLE1;
|

REVOKE (database privileges)

This form of the REVOKE statement revokes database privileges.

Syntax



Description

Each keyword listed revokes the privilege described, but only as it applies to or within the databases named in the statement.

DBADM

Revokes the database administrator authority.

DBCTRL

Revokes the database control authority.

DBMAINT

Revokes the database maintenance authority.

CREATETAB

Revokes the privilege to create new tables. If CREATETAB privilege is revoked from DSNDB04, tables cannot be created in implicitly created databases. For a work file database, you cannot revoke the privilege from PUBLIC. When a work file database is created, PUBLIC implicitly receives the CREATETAB privilege (without GRANT authority); this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

CREATETS

Revokes the privilege to create new table spaces.

DISPLAYDB

Revokes the privilege to issue the DISPLAY DATABASE command.

DROP

Revokes the privilege to issue the DROP or ALTER statements in the specified databases.

IMAGCOPY

Revokes the privilege to run the COPY, MERGECOPY, and QUIESCE utilities against table spaces of the specified databases, and to run the MODIFY RECOVERY utility.

LOAD

Revokes the privilege to use the LOAD utility to load tables.

RECOVERDB

Revokes the privilege to use the RECOVER and REPORT utilities to recover table spaces and indexes.

REORG

Revokes the privilege to use the REORG utility to reorganize table spaces and indexes.

REPAIR

Revokes the privilege to use the REPAIR and DIAGNOSE utilities.

STARTDB

Revokes the privilege to issue the START DATABASE command.

STATS

Revokes the privilege to use the RUNSTATS utility to update statistics, and the CHECK utility to test whether indexes are consistent with the data they index, and the MODIFY STATISTICS utility to delete unwanted statistics history records from the corresponding catalog tables.

STOPDB

Revokes the privilege to issue the STOP DATABASE command.

ON DATABASE *database-name,...*

Identifies databases on which you are revoking the privileges. For each database you identify, you (or the indicated grantors) must have granted at least one of the specified privileges on that database to all identified users (including PUBLIC, if specified). The same database must not be identified more than once.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

Examples

Example 1: Revoke drop privileges on database DSN8D91A from user PEREZ.

```
REVOKE DROP
  ON DATABASE DSN8D91A
  FROM PEREZ;
```

Example 2: Revoke repair privileges on database DSN8D91A from all local users. (Grants to specific users will not be affected.)

```
REVOKE REPAIR
  ON DATABASE DSN8D91A
  FROM PUBLIC;
```

Example 3: Revoke authority to create new tables and load tables in database DSN8D91A from users WALKER, PIANKA, and FUJIMOTO.

```
REVOKE CREATETAB,LOAD
      ON DATABASE DSN8D91A
      FROM WALKER,PIANKA,FUJIMOTO;
```

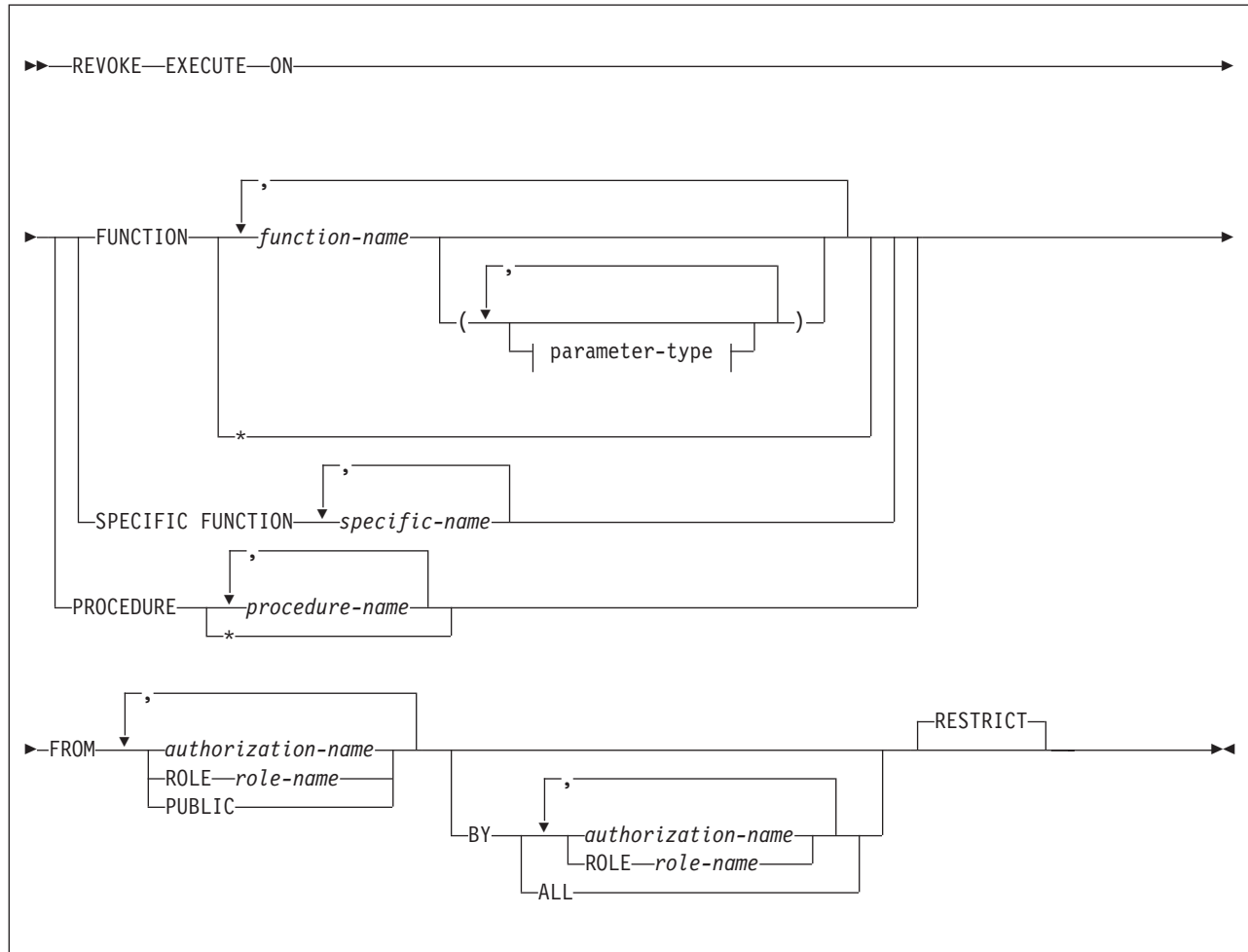
Example 4: Revoke load privileges on database DSN8D91A from role ROLE1:

```
REVOKE LOAD
      ON DATABASE DSN8D91A
      FROM ROLE ROLE1;
```

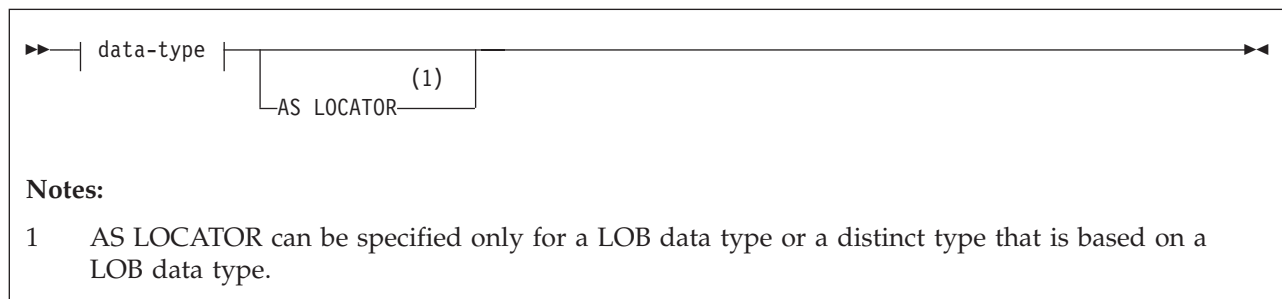
REVOKE (function or procedure privileges)

This form of the REVOKE statement revokes privileges on user-defined functions, cast functions that were generated for distinct types, and stored procedures.

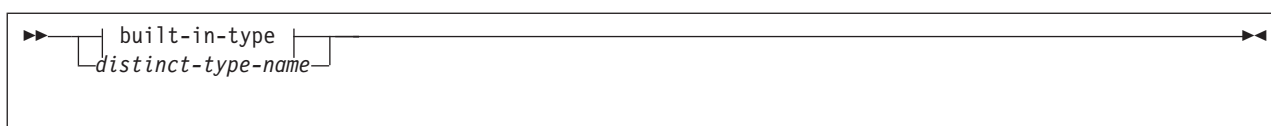
Syntax



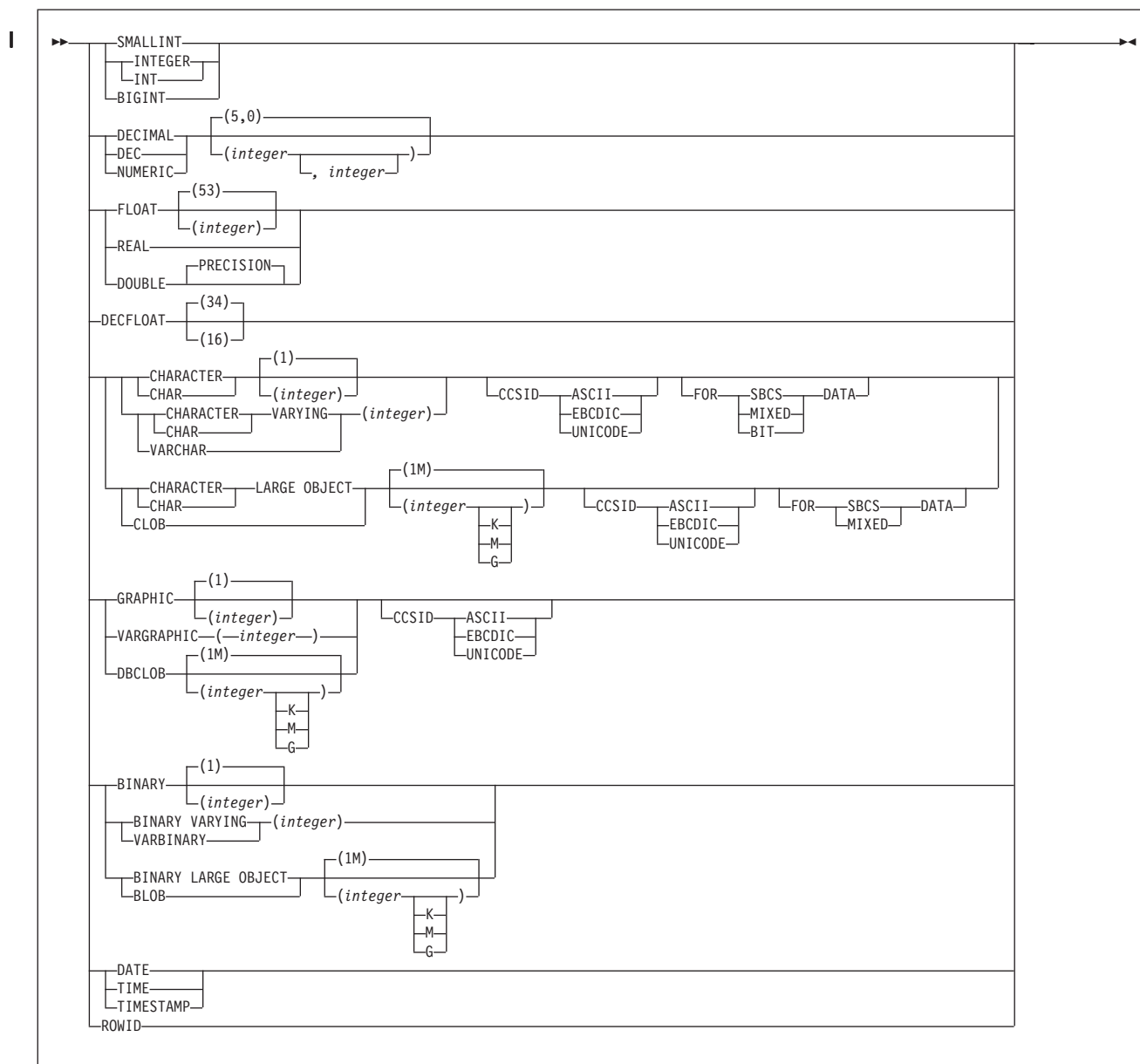
parameter-type:



data-type:



built-in-type:



Description

EXECUTE

Revokes the privilege to run the identified user-defined function, cast function that was generated for a distinct type, or stored procedure.

FUNCTION or SPECIFIC FUNCTION

Identifies the function from which the privilege is revoked. The function must exist at the current server, and it must be a function that was defined with the

CREATE FUNCTION statement or a cast function that was generated by a CREATE TYPE statement. The function can be identified by name, function signature, or specific name.

If the function was defined with a table parameter (the LIKE TABLE was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to identify the function. Instead, identify the function with its function name, if unique, or with its specific name.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function can have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

An * can be specified for a qualified or unqualified *function-name*. An * (or *schema-name.**) indicates that the privilege is revoked for all the functions in the schema. You (or the indicated grantors) must have granted the privilege on FUNCTION * to all identified users (including PUBLIC if specified). Privileges granted on specific functions are not affected.

FUNCTION *function-name (parameter-type,...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be granted. Synonyms for data types are considered a match.

If the function was defined with a table parameter (the LIKE TABLE *name* AS LOCATOR clause was specified in the CREATE FUNCTION statement to indicate that one of the input parameters is a transition table), the function signature cannot be used to uniquely identify the function. Instead, use one of the other syntax variations to identify the function with its function name, if unique, or its specific name.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function. If you do not explicitly qualify the function name with a schema name, the function name is implicitly qualified with a schema name as described in the preceding description for FUNCTION *function-name*.

(parameter-type,...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, DB2 searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). Similarly DECFLOAT() will be considered a match for DECFLOAT(16) or

DECFLOAT(34). However, FLOAT cannot be specified with empty parentheses because its parameter value indicates a specific data type (REAL or DOUBLE).

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or encoding scheme attribute, specifying the FOR *subtype* DATA clause or the CCSID clause is optional. Omission of either clause indicates that DB2 ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE *procedure-name*

Identifies a stored procedure that is defined at the current server.

An * can be specified for a qualified or unqualified *procedure-name*. An * (or *schema-name.**) indicates that the privilege is revoked for all the procedures in the schema. You (or the indicated grantors) must have granted the privilege on PROCEDURE * to all identified users (including PUBLIC if specified). Privileges granted on specific procedures are not affected.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

RESTRICT

Prevents the EXECUTE privilege from being revoked on a user-defined function or stored procedure if the revokee owns any of the following objects and does not have the EXECUTE privilege from another source:

- A function that is sourced on the function
- A view that uses the function
- A trigger package that uses the function or stored procedure
- A table that uses the function in a check constraint or user-defined default clause
- A materialized query table whose fullselect uses the function
- An extended index that uses the function

Examples

Example 1: Revoke the EXECUTE privilege on function CALC_SALARY for user JONES. Assume that there is only one function in the schema with function CALC_SALARY.

```
REVOKE EXECUTE ON FUNCTION CALC_SALARY FROM JONES;
```

Example 2: Revoke the EXECUTE privilege on procedure VACATION_ACCR from all users at the current server.

```
REVOKE EXECUTE ON PROCEDURE VACATION_ACCR FROM PUBLIC;
```

Example 3: Revoke the privilege of the administrative assistant to grant EXECUTE privileges on function DEPT_TOTAL to other users. The administrative assistant will still have the EXECUTE privilege on function DEPT_TOTALS.

```
REVOKE EXECUTE ON FUNCTION DEPT_TOTALS  
FROM ADMIN_A;
```

Example 4: Revoke the EXECUTE privilege on function NEW_DEPT_HIRES for HR (Human Resources). The function has two input parameters with data types of INTEGER and CHAR(10), respectively. Assume that the schema has more than one function that is named NEW_DEPT_HIRES.

```
REVOKE EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10))  
FROM HR;
```

You can also code the CHAR(10) data type as CHAR().

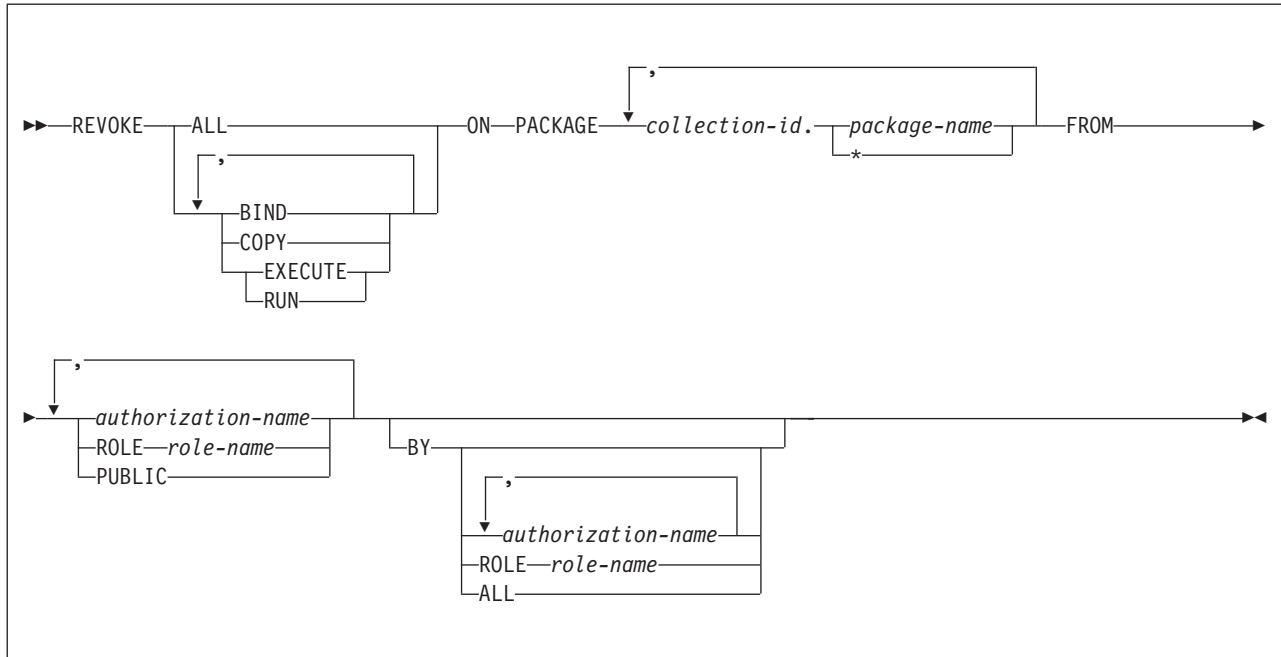
Example 5: Revoke the EXECUTE privilege on function FIND_EMPDEPT from role ROLE1:

```
REVOKE EXECUTE ON FUNCTION FIND_EMPDEPT  
FROM ROLE ROLE1;
```


REVOKE (package privileges)

This form of the REVOKE statement revokes privileges on packages.

Syntax



Description

BIND

Revokes the privilege to use the **BIND** and **REBIND** subcommands for the designated packages. In addition, if the value of field **BIND NEW PACKAGE** on installation panel **DSNTIPP** is **BIND**, the additional **BIND** privilege of adding new versions of packages is revoked. (For details, see “Notes” on page 1346 for “**GRANT (package privileges)**” on page 1345.)

COPY

Revokes the privilege to use the **COPY** option of the **BIND** subcommand for the designated packages.

EXECUTE

Revokes the privilege to run application programs that use the designated packages and to specify the packages following **PKLIST** for the **BIND PLAN** and **REBIND PLAN** commands. **RUN** is an alternate name for the same privilege.

ALL

Revokes all package privileges for which you have authority for the packages named in the **ON** clause.

ON PACKAGE *collection-id.package-name,...*

Identifies packages for which you are revoking privileges. The revoking of a package privilege applies to all versions of that package. For each package that you identify, you (or the indicated grantors) must have granted at least one of the specified privileges on that package to all identified users (including **PUBLIC**, if specified). An authorization ID with **PACKADM** authority over the

collection or all collections, SYSADM, or SYSCTRL authority can specify all packages in the collection by using * for *package-name*. The same package must not be specified more than once.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

Notes

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports specifying PROGRAM as a synonym for PACKAGE.

Examples

Example 1: Revoke the privilege to copy all packages in collection DSN8CC61 from LEWIS.

```
REVOKE COPY ON PACKAGE DSN8CC61.* FROM LEWIS;
```

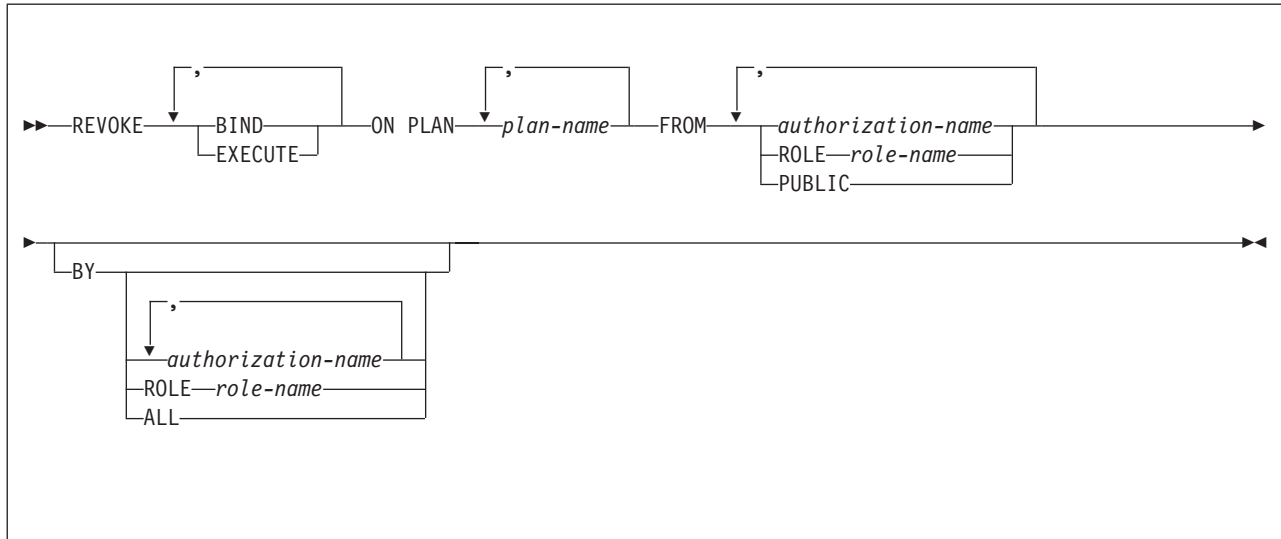
Example 2: Revoke the privilege to run all packages in collection DSN9CC13 from role ROLE1:

```
REVOKE EXECUTE ON PACKAGE DSN9CC13.* FROM ROLE ROLE1;
```

REVOKE (plan privileges)

This form of the REVOKE statement revokes privileges on application plans.

Syntax



Description

BIND

Revokes the privilege to use the **BIND**, **REBIND**, and **FREE** subcommands for the identified plans.

EXECUTE

Revokes the privilege to run application programs that use the identified plans.

ON PLAN *plan-name*,...

Identifies application plans for which you are revoking privileges. For each plan that you identify, you (or the indicated grantors) must have granted at least one of the specified privileges on that plan to all identified users (including **PUBLIC**, if specified). The same plan must not be specified more than once.

FROM

Refer to “**REVOKE**” on page 1432 for a description of the **FROM** clause.

BY Refer to “**REVOKE**” on page 1432 for a description of the **BY** clause.

Examples

Example 1: Revoke authority to bind plan **DSN8IP91** from user **JONES**.

```
REVOKE BIND ON PLAN DSN8IP91 FROM JONES;
```

Example 2: Revoke authority previously granted to all users at the current server to bind and execute plan **DSN8CP91**. (Grants to specific users will not be affected.)

```
REVOKE BIND,EXECUTE ON PLAN DSN8CP91 FROM PUBLIC;
```

Example 3: Revoke authority to execute plan **DSN8CP91** from users **ADAMSON** and **BROWN**.

```
REVOKE EXECUTE ON PLAN DSN8CP91 FROM ADAMSON,BROWN;
```

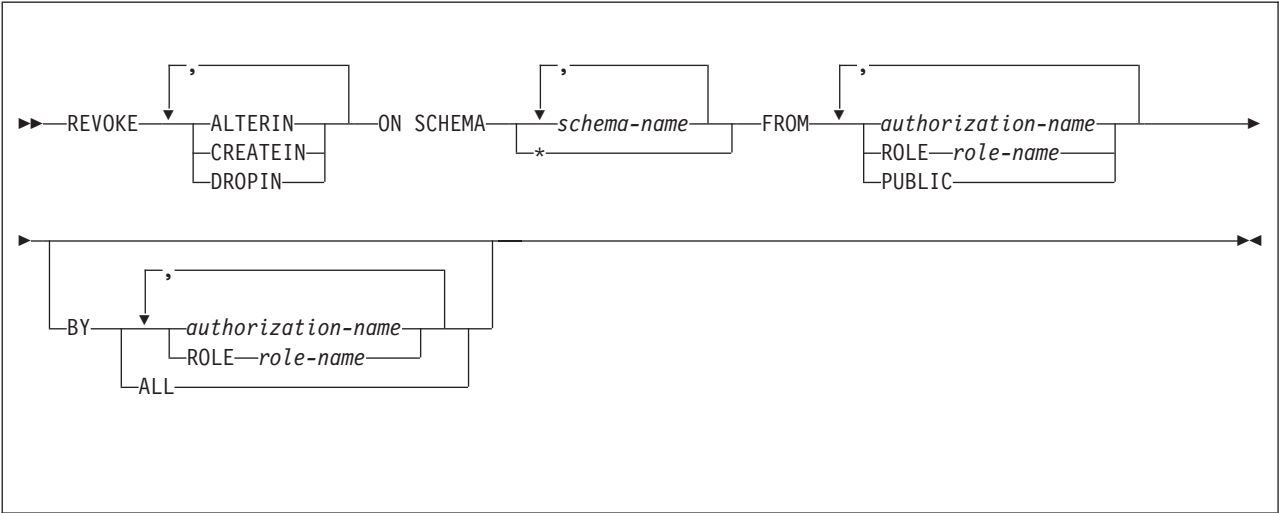
Example 4: Revoke authority to bind plan DSN91PLN from role ROLE1:

```
REVOKE BIND ON PLAN DSN91PLN FROM ROLE ROLE1;
```

REVOKE (schema privileges)

This form of the REVOKE statement revokes privileges on schemas.

Syntax



Description

ALTERIN

Revokes the privilege to alter sequences, stored procedures, and user-defined functions, or specify a comment for distinct types, cast functions that are generated for distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

CREATEIN

Revokes the privilege to create distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

DROPIN

Revokes the privilege to drop distinct types, sequences, stored procedures, triggers, and user-defined functions in the designated schemas.

SCHEMA *schema-name*

Identifies the schema on which the privilege is revoked.

SCHEMA *

Indicates that the specified privilege on all schemas is revoked. You (or the indicated grantors) must have previously granted the specified privilege on SCHEMA * to all identified users (including PUBLIC if specified). Privileges granted on specific schemas are not affected.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

Examples

Example 1: Revoke the CREATEIN privilege on schema T_SCORES from user JONES.

```
REVOKE CREATEIN ON SCHEMA T_SCORES FROM JONES;
```

Example 2: Revoke the CREATEIN privilege on schema VAC from all users at the current server.

```
REVOKE CREATEIN ON SCHEMA VAC FROM PUBLIC;
```

Example 3: Revoke the ALTERIN privilege on schema DEPT from the administrative assistant.

```
REVOKE ALTERIN ON SCHEMA DEPT FROM ADMIN_A;
```

Example 4: Revoke the ALTERIN and DROPIN privileges on schemas NEW_HIRE, PROMO, and RESIGN from HR (Human Resources).

```
REVOKE ALTERIN, DROPIN ON SCHEMA NEW_HIRE, PROMO, RESIGN FROM HR;
```

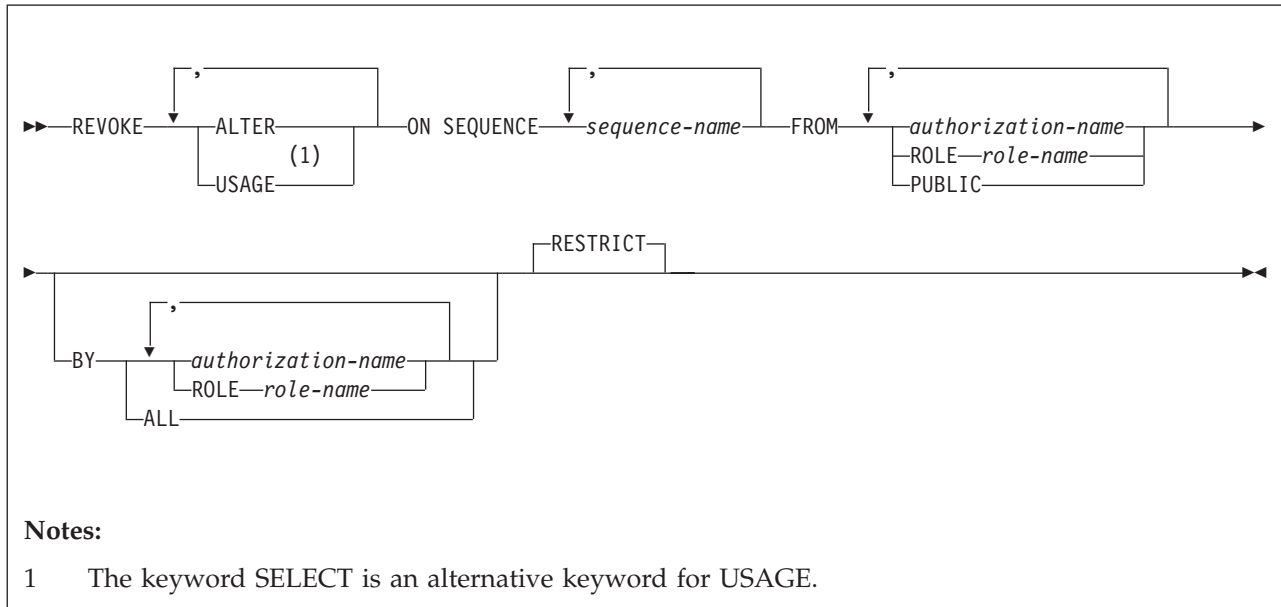
Example 5: Revoke the ALTERIN privilege on schemas EMPLOYEE from role ROLE1:

```
REVOKE ALTERIN ON SCHEMA EMPLOYEE FROM ROLE ROLE1;
```

REVOKE (sequence privileges)

This form of the REVOKE statement revokes the privileges on a user-defined sequence.

Syntax



Description

ALTER

Revokes the privilege to alter a sequence or record a comment on a sequence.

USAGE

Revokes the **USAGE** privilege to use a sequence. This privilege is needed when the **NEXT VALUE** or **PREVIOUS VALUE** expression is invoked for a sequence name.

SEQUENCE *sequence-name*

Identifies the sequence. The name, including the implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error occurs. *sequence-name* must not be the name of an internal sequence object that is generated by the system for an identity column.

FROM

Refer to “**REVOKE**” on page 1432 for a description of the **FROM** clause.

BY Refer to “**REVOKE**” on page 1432 for a description of the **BY** clause.

RESTRICT

Prevents the **USAGE** privilege from being revoked on a sequence if the revokee owns one of the following objects and does not have the **USAGE** privilege from another source:

- A trigger that specifies the sequence in a **NEXT VALUE** or **PREVIOUS VALUE** expression
- An inline SQL function that specifies the sequence in a **NEXT VALUE** or **PREVIOUS VALUE** expression

Examples

Example 1: Revoke USAGE privilege on sequence MYNUM to user JONES.

```
REVOKE USAGE
  ON SEQUENCE MYNUM
  FROM JONES;
```

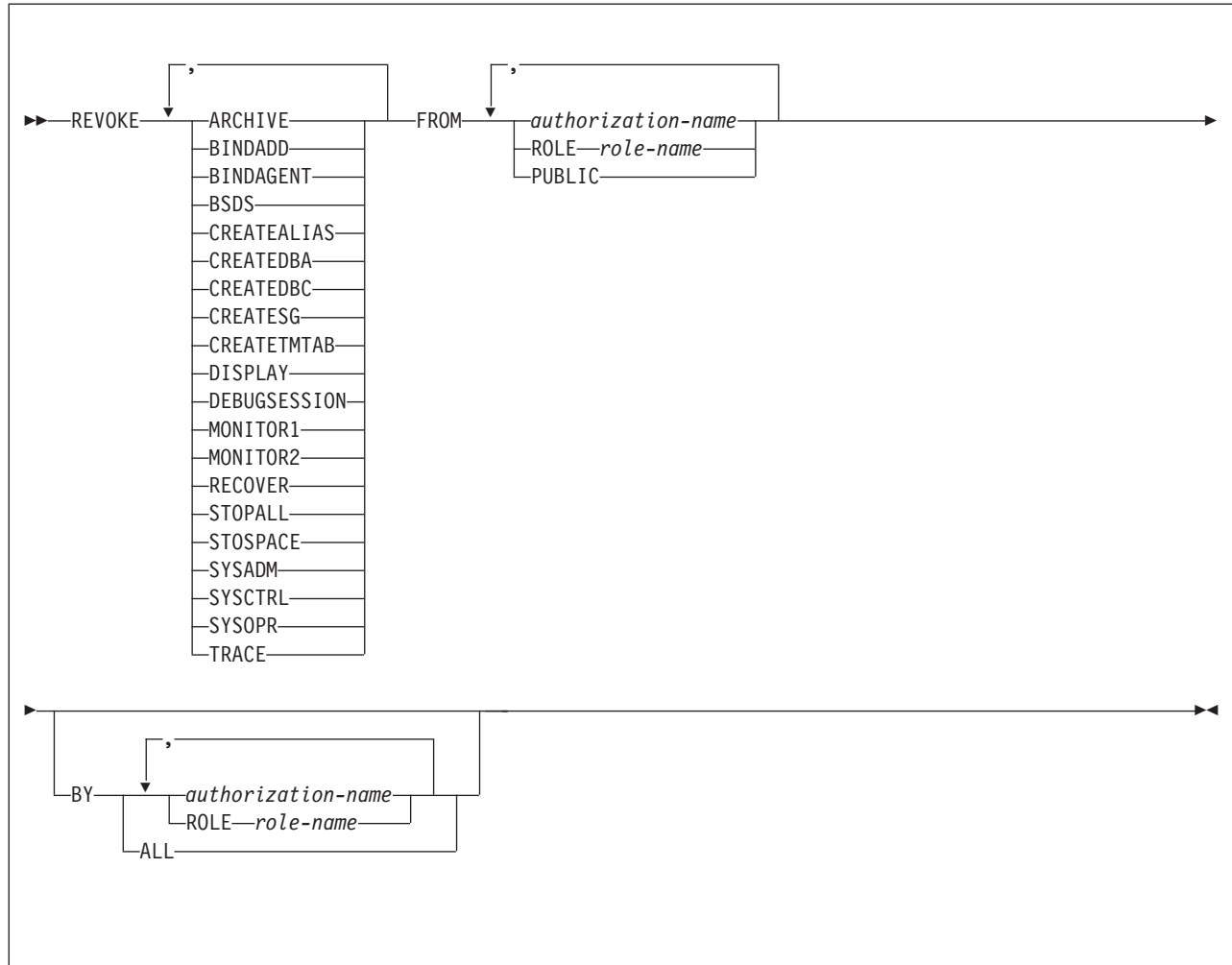
Example 2: Revoke the USAGE privilege on sequence ORDER_SEQ from role ROLE1:

```
REVOKE USAGE
  ON SEQUENCE ORDER_SEQ
  FROM ROLE ROLE1;
```


REVOKE (system privileges)

This form of the REVOKE statement revokes system privileges.

Syntax



Description

ARCHIVE

Revokes the privilege to use the ARCHIVE LOG command.

BINDADD

Revokes the privilege to create plans and packages using the BIND subcommand with the ADD option.

BINDAGENT

Revokes the privilege to issue the BIND, FREE PACKAGE, or REBIND subcommands for plans and packages and the DROP PACKAGE statement on behalf of the grantor. The privilege also allows the holder to copy and replace plans and packages on behalf of the grantor.

A revoke of this privilege does not cascade.

BSDS

Revokes the privilege to issue the RECOVER BSDS command.

CREATEALIAS

Revokes the privilege to use the CREATE ALIAS statement.

CREATEDBA

Revokes the privilege to issue the CREATE DATABASE statement and acquire DBADM authority over those databases.

CREATEDBC

Revokes the privilege to issue the CREATE DATABASE statement and acquire DBCTRL authority over those databases.

CREATESG

Revokes the privilege to create new storage groups.

CREATETMTAB

Revokes the privilege to use the CREATE GLOBAL TEMPORARY TABLE statement.

DISPLAY

Revokes the privilege to use the following commands:

- The DISPLAY ARCHIVE command for archive log information
- The DISPLAY BUFFERPOOL command for the status of buffer pools
- The DISPLAY DATABASE command for the status of all databases
- The DISPLAY FUNCTION SPECIFIC command for statistics about accessed external user-defined functions
- The DISPLAY LOCATION command for statistics about threads with a distributed relationship
- The DISPLAY PROCEDURE command for statistics about accessed stored procedures
- The DISPLAY THREAD command for information on active threads with in DB2
- The DISPLAY TRACE command for a list of active traces

DEBUGSESSION

Revokes the privilege to create a debug session, which prevents client application debugging of native SQL or Java procedures that are executed within the session.

MONITOR1

Revokes the privilege to obtain IFC data classified as serviceability data, statistics, accounting, and other performance data that does not contain potentially secure data.

MONITOR2

Revokes the privilege to obtain IFC data classified as containing potentially sensitive data such as SQL statement text and audit data. (Having the MONITOR2 privilege also implies having MONITOR1 privileges, however, revoking the MONITOR2 privilege does not cause the revoke of an explicitly granted MONITOR1 privilege.)

RECOVER

Revokes the privilege to issue the RECOVER INDOUBT command.

STOPALL

Revokes the privilege to use the STOP DB2 command.

STOSPACE

Revokes the privilege to use the STOSPACE utility.

SYSADM

Revokes the system administrator authority.

SYSCTRL

Revokes the system control authority.

SYSOPR

Revokes the system operator authority.

TRACE

Revokes the privilege to use the MODIFY TRACE, START TRACE, and STOP TRACE commands.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

Examples

Example 1: Revoke DISPLAY privileges from user LUTZ.

```
REVOKE DISPLAY
FROM LUTZ;
```

Example 2: Revoke BSDS and RECOVER privileges from users PARKER and SETRIGHT.

```
REVOKE BSDS,RECOVER
FROM PARKER,SETRIGHT;
```

Example 3: Revoke TRACE privileges previously granted to all local users. (Grants to specific users will not be affected.)

```
REVOKE TRACE
FROM PUBLIC;
```

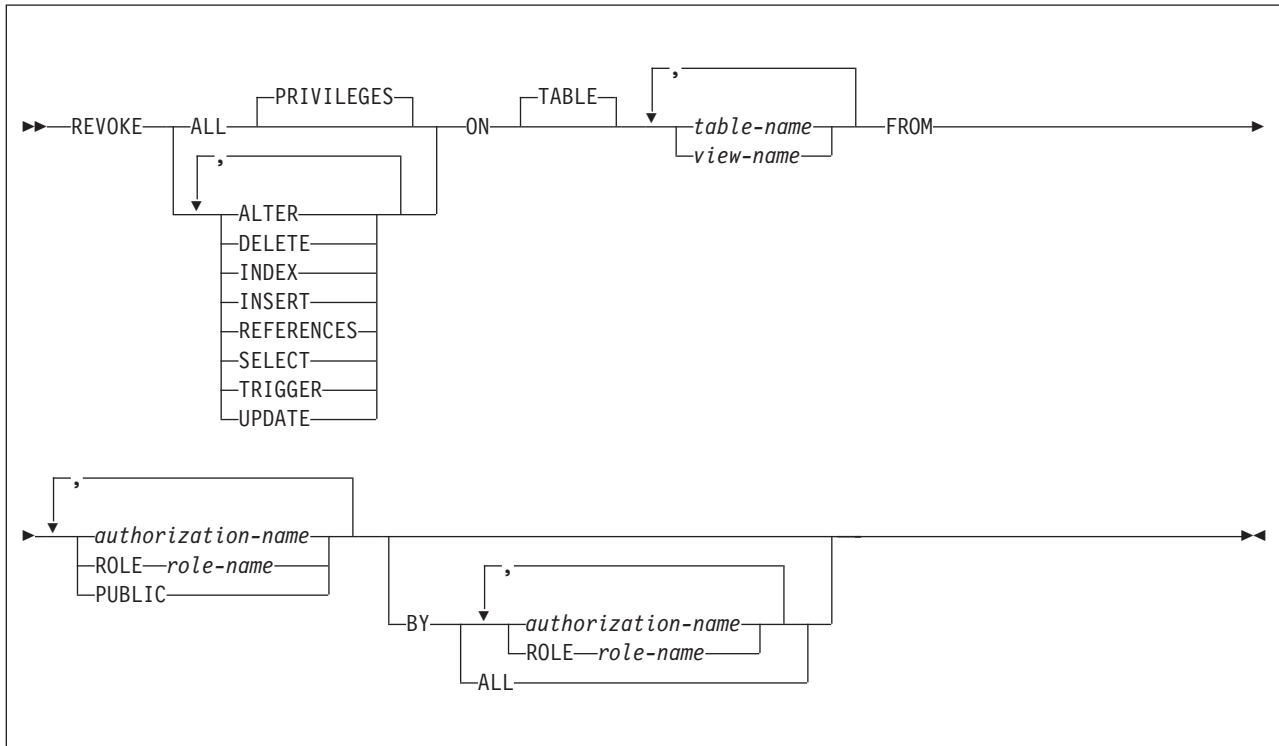
Example 4: Revoke ARCHIVE privileges from role ROLE1:

```
REVOKE ARCHIVE
FROM ROLE ROLE1;
```

REVOKE (table or view privileges)

This form of the REVOKE statement revokes privileges on one or more tables or views.

Syntax



Description

ALL or ALL PRIVILEGES

If you specify **ALL**, the authorization ID of the statement must have granted a least one privilege on each identified table or view to each *authorization-name*. The privilege revoked from an authorization ID are those privileges on the table or view that the authorization ID of the statement granted to the authorization ID.

If you do not use **ALL**, you must use one or more of the keywords listed below. Each keyword revokes the privilege described, but only as it applies to the tables or views named in the **ON** clause.

ALTER

Revokes the privilege to alter the specified table or create a trigger on the specified table.

DELETE

Revokes the privilege to delete rows in the specified table or view.

INDEX

Revokes the privilege to create an index on the specified table.

INSERT

Revokes the privilege to insert rows into the specified table or view.

REFERENCES

Revokes the privilege to define and drop referential constraints. Although you can use a list of column names with the GRANT statement, you cannot use a list of column names with REVOKE; the privilege is revoked for all columns.

SELECT

Revokes the privilege to create a view or read data from the specified table or view. A view or a materialized query table is dropped when the SELECT privilege that was used to create it is revoked, unless the owner of the view or materialized query table was directly granted the SELECT privilege from another source before the view or materialized query table was created.

TRIGGER

Revokes the privilege to create a trigger on the specified table.

UPDATE

Revokes the privilege to update rows in the specified table or view. A list of column names can be used only with GRANT, not with REVOKE.

ON *table-name* or *view-name*

Names one or more tables or views on which you are revoking the privileges. The list can consist of table names, view names, or a combination of the two. A table or view must not be identified more than one time, and a declared temporary table and a table that is implicitly created for an XML column must not be identified.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY If you omit BY, you must have granted each named privilege to each of the named users. More precisely, each privilege must have been granted to each user by a GRANT statement whose authorization ID is also the authorization ID of your REVOKE statement. Each of these grants is then revoked. (No single privilege need be granted on all tables and views.)

If BY is specified, each named grantor must satisfy the above requirement. In that case, the authorization ID of the statement need not satisfy the requirement unless it is one of the named grantors.

Refer to “REVOKE” on page 1432 for a description of the BY clause.

Notes

For a created temporary table or a view of a created temporary table, only ALL or ALL PRIVILEGES can be revoked. Specific table or view privileges cannot be revoked.

For a declared temporary table, no privileges can be revoked because none can be granted. When a declared temporary table is defined, PUBLIC implicitly receives all table privileges (without GRANT authority) for the table. These privileges are not recorded in the DB2 catalog.

PUBLIC AT ALL LOCATIONS: PUBLIC AT ALL LOCATIONS can continue to be specified as an alternative to PUBLIC as in prior releases. PUBLIC AT ALL LOCATIONS was introduced and was intended for use only with DB2 private protocol access.

Examples

Example 1: Revoke SELECT privileges on table DSN8910.EMP from user PULASKI.

```
REVOKE SELECT ON TABLE DSN8910.EMP FROM PULASKI;
```

Example 2: Revoke update privileges on table DSN8910.EMP previously granted to all local DB2 users. (Grants to specific users are not affected.)

```
REVOKE UPDATE ON TABLE DSN8910.EMP FROM PUBLIC;
```

Example 3: Revoke all privileges on table DSN8910.EMP from users KWAN and THOMPSON.

```
REVOKE ALL ON TABLE DSN8910.EMP FROM KWAN, THOMPSON;
```

Example 4: Revoke the grant of SELECT and UPDATE privileges on the table DSN8910.DEPT to every user in the network. Doing so does not affect users who obtained these privileges from some other grant.

```
REVOKE SELECT, UPDATE ON TABLE DSN8910.DEPT  
FROM PUBLIC;
```

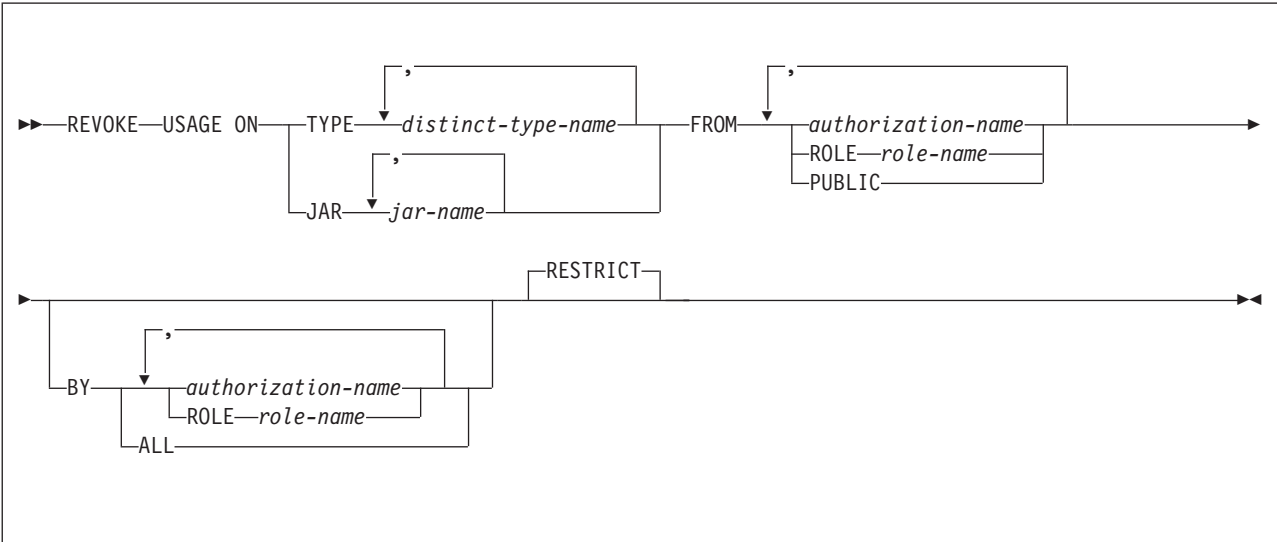
Example 5: Revoke the ALTER privileges on the table DSN8910.EMP that were previously granted to role ROLE1:

```
REVOKE ALTER ON TABLE DSN8910.EMP  
FROM ROLE ROLE1;
```

REVOKE (type or JAR file privileges)

This form of the REVOKE statement revokes the privilege to use distinct types (user-defined data types) or JAR files.

Syntax



Description

USAGE

Revokes the privilege to use the distinct type in tables, functions procedures, or the privilege to use the JAR file.

TYPE *distinct-type-name*

Identifies a distinct type. The name, including the implicit or explicit schema qualifier, must identify a unique distinct type that exists at the current server.

JAR *jar-name*

Identifies the JAR file. The name, including the implicit or explicit schema name, must identify a unique JAR file that exists at the current server.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

RESTRICT

Prevents the USAGE privilege from being revoked on a distinct type or JAR file if any of the following conditions exist and the revokee does not have the USAGE privilege from another source:

- The revokee owns a function or stored procedure that uses the distinct type or references the JAR file.
- The revokee owns a JAR file whose path references the JAR file for which USAGE is being revoked.
- The revokee owns a table that has a column that uses the distinct type.
- A sequence exists for which the data type of the sequence is the distinct type.

Notes

Alternative syntax and synonyms: To provide compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports DATA TYPE or DISTINCT TYPE as a synonym for TYPE.

Examples

Example 1: Revoke the USAGE privilege on distinct type SHOESIZE from user JONES.

```
REVOKE USAGE ON TYPE SHOESIZE FROM JONES;
```

Example 2: Revoke the USAGE privilege on distinct type US_DOLLAR from all users at the current server except for those who have been specifically granted USAGE and not through PUBLIC.

```
REVOKE USAGE ON TYPE US_DOLLAR FROM PUBLIC;
```

Example 3: Revoke the USAGE privilege on distinct type CANADIAN_DOLLARS from the administrative assistant (ADMIN_A).

```
REVOKE USAGE ON TYPE CANADIAN_DOLLARS  
FROM ADMIN_A;
```

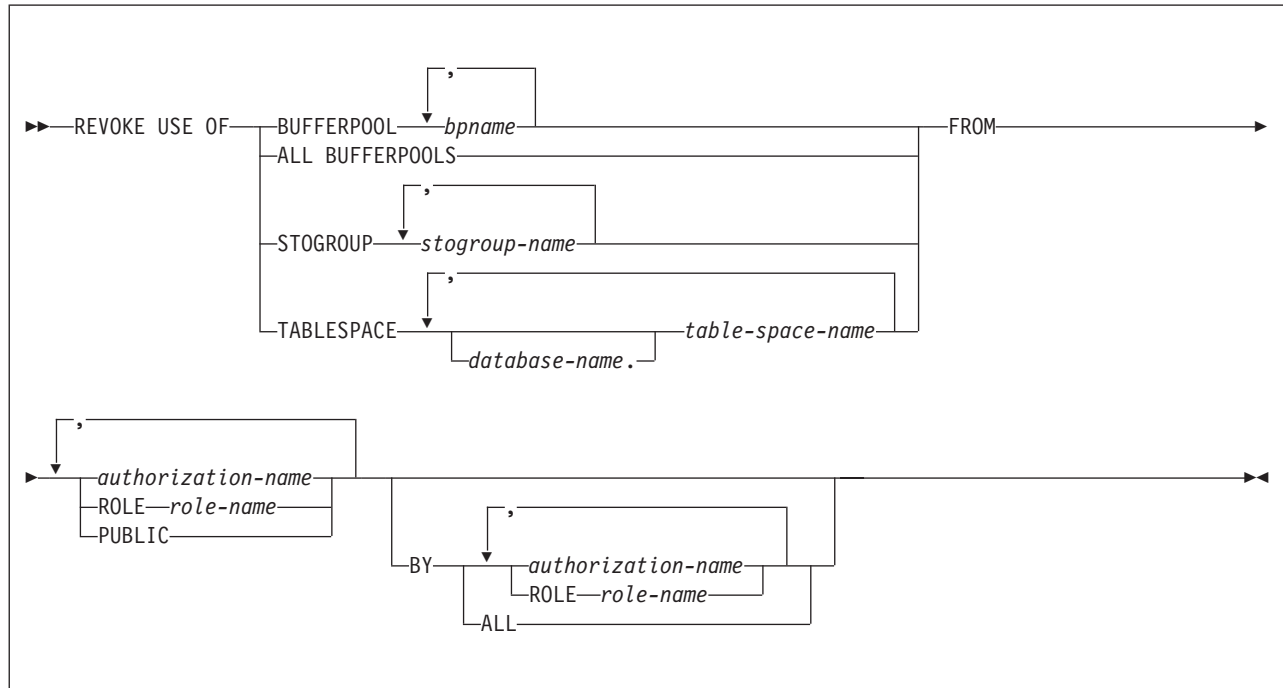
Example 4: Revoke the USAGE privilege on distinct type MILES from the role ROLE1:

```
REVOKE USAGE ON TYPE MILES  
FROM ROLE ROLE1;
```


REVOKE (use privileges)

This form of the REVOKE statement revokes authority to use particular buffer pools, storage groups, or table spaces.

Syntax



Description

BUFFERPOOL *bpname*,...

Revokes the privilege to refer to any of the identified buffer pools in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement. See “Naming conventions” on page 51 for more details about *bpname*.

ALL BUFFERPOOLS

Revokes the privilege to refer to any buffer pool in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

STOGROUP *stogroup-name*,...

Revokes the privilege to refer to any of the identified storage groups in a CREATE INDEX, CREATE TABLESPACE, ALTER INDEX, or ALTER TABLESPACE statement.

TABLESPACE *database-name.table-space-name*,...

Revokes the privilege to refer to any of the specified table spaces in a CREATE TABLE statement. The default *database-name* is DSNDB04.

For table spaces in a work file database you cannot revoke the privilege from PUBLIC. When a table space is created in a work file database, PUBLIC implicitly receives the TABLESPACE privilege (without GRANT authority); this privilege is not recorded in the DB2 catalog, and it cannot be revoked.

FROM

Refer to “REVOKE” on page 1432 for a description of the FROM clause.

BY Refer to “REVOKE” on page 1432 for a description of the BY clause.

Notes

You can revoke privileges for only one type of object with each statement. Thus you can revoke the use of several table spaces with one statement, but not the use of a table space and a storage group.

For each object you name, you (or the indicated grantors) must have granted the USE privilege on that object to all identified users (including PUBLIC, if specified). The same object must not be identified more than once.

Revoking the privilege USE OF ALL BUFFERPOOLS does not cascade to all other privileges that can be granted under that privilege. A user with the privilege USE OF ALL BUFFERPOOLS WITH GRANT OPTION can make two types of grants:

- GRANT USE OF ALL BUFFERPOOLS TO *userid*. This privilege is revoked when the original user's privilege is revoked.
- GRANT USE OF BUFFERPOOL BP_{*n*} TO *userid*. This privilege is *not revoked* when the original user's privilege is revoked.

Examples

Example 1: Revoke authority to use buffer pool BP2 from user MARINO.

```
REVOKE USE OF BUFFERPOOL BP2
FROM MARINO;
```

Example 2: Revoke a grant of the USE privilege on the table space DSN8S91D in the database DSN8D91A. The grant is to PUBLIC, that is, to everyone at the local DB2 subsystem. (Grants to specific users are not affected.)

```
REVOKE USE OF TABLESPACE DSN8D91A.DSN8S91D
FROM PUBLIC;
```

Example 3: Revoke the authority to use storage group SG1 from role ROLE1:

```
REVOKE USE OF STOGROUP SG1
FROM ROLE ROLE1;
```

ROLLBACK

The ROLLBACK statement can be used to end a unit of recovery and back out all the relational database changes that were made by that unit of recovery. If relational databases are the only recoverable resources used by the application process, ROLLBACK also ends the unit of work. ROLLBACK can also be used to back out only the changes made after a savepoint was set within the unit of recovery without ending the unit of recovery. Rolling back to a savepoint enables selected changes to be undone.

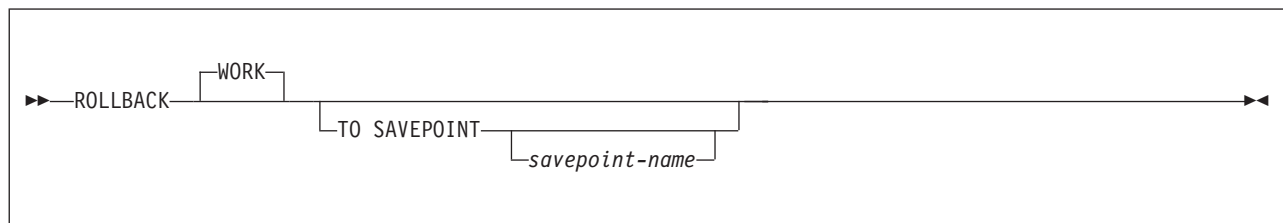
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared. It can be used in the IMS or CICS environment only if the TO SAVEPOINT clause is specified.

Authorization

None required.

Syntax



Description

When ROLLBACK is used without the SAVEPOINT clause, the unit of recovery in which the ROLLBACK statement is executed is ended and a new unit of recovery is started.

All changes that are made by the following statements during the unit of recovery are backed out:

- ALTER
- COMMENT
- CREATE
- DELETE
- DROP
- EXPLAIN
- GRANT
- INSERT
- LABEL
- MERGE
- REFRESH TABLE
- RENAME
- REVOKE
- SELECT INTO with an SQL data change statement

- select-statement with an SQL data change statement
- TRUNCATE when the IMMEDIATE clause is not specified
- UPDATE

ROLLBACK without the TO SAVEPOINT clause also causes the following actions to occur:

- All locks that are implicitly acquired during the unit of recovery are released. See “LOCK TABLE” on page 1386 for an explanation of the duration of explicitly acquired locks.
- All cursors are closed, all prepared statements are destroyed, and any cursors that are associated with the prepared statements are invalidated.
- All rows and all logical work files of every created temporary table of the application process are deleted. (All the rows of a declared temporary table are not implicitly deleted. As with base tables, any changes that are made to a declared temporary table during the unit of recovery are undone to restore the table to its state at the last commit point.)
- All LOB locators, including those that are held, are freed.

TO SAVEPOINT

Specifies that the unit of recovery is not to be ended and that only a partial rollback (to a savepoint) is to be performed. If a savepoint name is not specified, rollback is to the last active savepoint. For example, if in a unit of recovery, savepoints A, B, and C are set in that order and then C is released, ROLLBACK TO SAVEPOINT causes a rollback to savepoint B.

savepoint-name

Identifies the savepoint to which to roll back. The name must identify a savepoint that exists at the current server.

All database changes (including changes made to a declared temporary tables but excluding changes made to created temporary tables) that were made after the savepoint was set are backed out. Changes that are made to created temporary tables are not logged and are not backed out; a warning is issued instead. (A warning is also issued when a created temporary table is changed and there is an active savepoint.)

In addition, none of the following items are backed out:

- The opening or closing of cursors
- Changes in cursor positioning
- The acquisition and release of locks
- The caching of the rolled back statements

Any savepoints that are set after the one to which rollback is performed are released. The savepoint to which rollback is performed is not released.

ROLLBACK with or without the TO SAVEPOINT clause has no effect on connections.

Notes

The following information applies only to rolling back all changes in the unit of recovery (the ROLLBACK statement without the TO SAVEPOINT clause):

- *Stored procedures.* The ROLLBACK statement cannot be used if the procedure is in the calling chain of a user-defined function or a trigger or if DB2 is not the commit coordinator.

- *IMS or CICS.* Using a ROLLBACK to SAVEPOINT statement in an IMS or CICS environment only rolls back DB2 resources. Any other recoverable resources updated in the environment are not rolled back. To do a rollback operation in these environments, SQL programs must use the call prescribed by their transaction manager. The effect of these rollback operations on DB2 data is the same as that of the SQL ROLLBACK statement.

A rollback operation in an IMS or CICS environment might handle the closing of cursors that were declared with the WITH hold option differently than the SQL ROLLBACK statement does. If an application requests a rollback operation from CICS or IMS, but no work has been performed in DB2 since the last commit point, the rollback request will not be broadcast to DB2. If the application had opened cursors using the WITH HOLD option in a previous unit of work, the cursors will not be closed, and any prepared statements associated with those cursors will not be destroyed.

- *Implicit rollback operations:* In all DB2 environments, the abend of a process is an implicit rollback operation.

ROLLBACK and table spaces that are not logged: If ROLLBACK is executed for a unit of work that includes changes to a non-LOB table space that is not logged (specifies the NOT LOGGED attribute), that table space is marked RECOVER-pending and the table space is placed in the logical page list. The table space is therefore not available after the rollback operation completes. See *DB2 Utility Guide and Reference* for more information about the RECOVER utility.

Examples

Example 1: Roll back all DB2 database changes made since the unit of recovery was started.

```
ROLLBACK WORK;
```

Example 2: After a unit of recovery started, assume that three savepoints A, B, and C were set and that C was released:

```
...
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
...
SAVEPOINT B ON ROLLBACK RETAIN CURSORS;
...
SAVEPOINT C ON ROLLBACK RETAIN CURSORS;
...
RELEASE SAVEPOINT C;
...
```

Roll back all DB2 database changes only to savepoint A:

```
ROLLBACK WORK TO SAVEPOINT A;
```

If a savepoint name was not specified (that is, ROLLBACK WORK TO SAVEPOINT), the rollback would be to the last active savepoint that was set, which is B.

SAVEPOINT

The **SAVEPOINT** statement sets a savepoint within a unit of recovery to identify a point in time within the unit of recovery to which relational database changes can be rolled back.

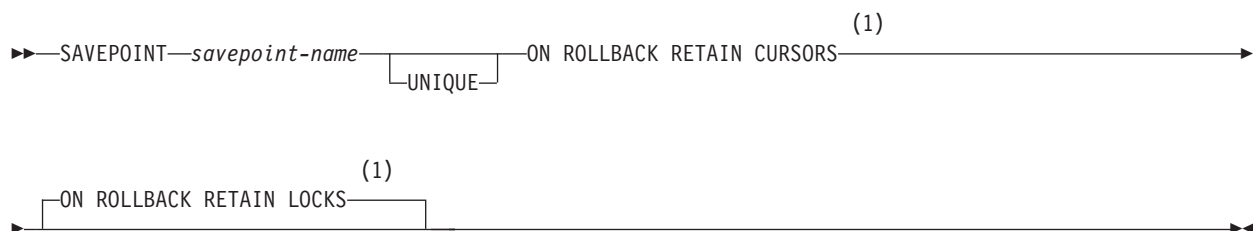
Invocation

This statement can be imbedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Notes:

- 1 These clauses can be specified in either order.

Description

savepoint-name

Names the savepoint. *savepoint-name* must not begin with 'SYS'.

UNIQUE

Specifies that the application program cannot reuse the savepoint name within the unit of recovery. An error occurs if a savepoint with the same name as *savepoint-name* already exists within the unit of recovery.

Omitting **UNIQUE** indicates that the application can reuse the savepoint name within the unit of recovery. If *svpt-name* identifies a savepoint that already exists within the unit of recovery and the savepoint was not created with the **UNIQUE** option, the existing savepoint is destroyed and a new savepoint is created. Destroying a savepoint to reuse its name for another savepoint is not the same as releasing the savepoint. Reusing a savepoint name destroys only one savepoint. Releasing a savepoint with the **RELEASE SAVEPOINT** statement releases the savepoint and all savepoints that have been subsequently set.

ON ROLLBACK RETAIN CURSORS

Specifies that any cursors that are opened after the savepoint is set are not tracked, and thus, are not closed upon rollback to the savepoint. Although these cursors remain open after rollback to the savepoint, they might not be usable. For example, if rolling back to the savepoint causes the insertion of a

row on which the cursor is positioned to be rolled back, using the cursor to update or delete the row results in an error.

ON ROLLBACK RETAIN LOCKS

Specifies that any locks that are acquired after the savepoint is set are not tracked, and thus, are not released on rollback to the savepoint.

Example

Assume that you want to set three savepoints at various points in a unit of recovery. Name the first savepoint A and allow the savepoint name to be reused. Name the second savepoint B and do not allow the name to be reused. Because you no longer need savepoint A when you are ready to set the third savepoint, reuse A as the name of the savepoint.

```
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;  
:  
SAVEPOINT B UNIQUE ON ROLLBACK RETAIN CURSORS;  
:  
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
```

SELECT

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR statement, or prepared and then referenced in a DECLARE CURSOR statement. It can also be issued interactively using SPUFI or the command line processor which causes a result table to be displayed at your terminal. In any case, the table specified by *select-statement* is the result of the fullselect.

For a description of the SELECT statement, see “select-statement” on page 669.

SELECT INTO

The SELECT INTO statement produces a result table that contains at most one row, and assigns the values in that row to host variables. If the table is empty, the statement assigns +100 to SQLCODE, '02000' to SQLSTATE, and does not assign values to the host variables. The tables or views identified in the statement can exist at the current server or at any DB2 subsystem with which the current server can establish a connection.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

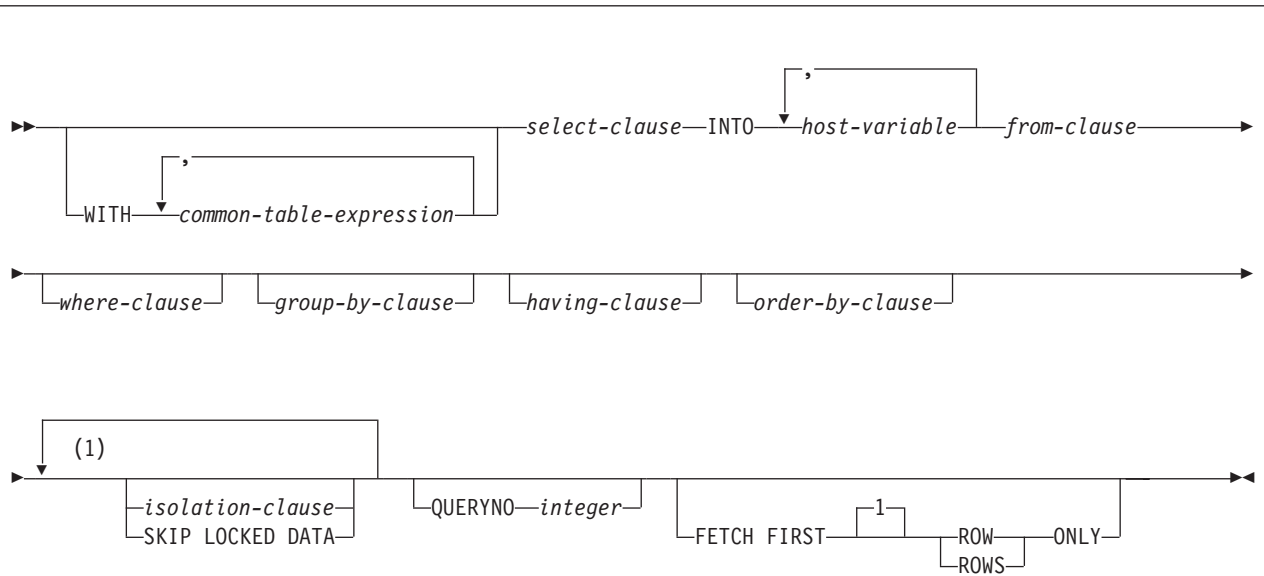
Authorization

The privileges that are held by the authorization ID of the owner of the plan or package must include at least one of the following for every table and view identified in the statement:

- The SELECT privilege on the table or view
- Ownership of the table or view
- DBADM authority for the database (tables only)
- SYSADM authority
- SYSCTRL authority (catalog tables only)

If the SELECT INTO statement includes an SQL data change statement, the privilege set must also include at least the privileges (INSERT, UPDATE, or DELETE) that are associated with that SQL data change statement on the table or view.

Syntax



Notes:

- 1 The same clause must not be specified more than once.

Description

The table is derived by evaluating the *isolation-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, and the *select-clause*, in this order. See Chapter 4, “Queries,” on page 629 for a description of these clauses.

WITH *common-table-expression*

Refer to “common-table-expression” on page 670 for information about specifying a *common-table-expression*.

INTO *host-variable*,...

Each *host-variable* must identify a structure or variable that is described in the program in accordance with the rules for declaring host structures and variables. In the operational form of the INTO clause, a reference to a structure is replaced by a reference to each of its host variables.

The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on. If the number of host variables is less than the number of column values, the value W is assigned to the SQLWARN3 field of the SQLCA. (See “SQL communication area (SQLCA)” on page 1646.)

The data type of a variable must be compatible with the value assigned to it. If the value is numeric, the variable must have the capacity to represent the integral part of the value. For a date or time value, the variable must be a character string variable of a minimum length as defined in Chapter 2, “Language elements,” on page 47. If the value is null, an indicator variable must be specified.

Each assignment to a variable is made according to the rules described in Chapter 2, “Language elements,” on page 47. Assignments are made in sequence through the list.

If an error occurs as the result of an arithmetic expression in the SELECT list of a SELECT INTO statement (division by zero or overflow) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.) If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered.

If an error occurs, no value is assigned to the host variable or to later variables, though any values that have already been assigned to variables remain assigned.

If an error occurs because the result table has more than one row, values may or may not be assigned to the host variables. If values are assigned to the host variables, the row that is the source of the values is undefined and not predictable.

isolation-clause

Specifies the isolation level at which the statement is executed and, optionally, the type of locks that are acquired.

SKIP LOCKED DATA

Specifies that rows are skipped when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. SKIP LOCKED DATA can be used only when isolation CS or RS is in effect and applies only to row level or page level locks.

SKIP LOCKED DATA is ignored if it is specified when the isolation level that is in effect is repeatable read (WITH RR) or uncommitted read (WITH UR).

QUERYNO *integer*

Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO columns of the plan tables for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the QUERYNO clause to assign unique numbers to the SQL statements in a program is helpful:

- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

FETCH FIRST ROW ONLY *integer*

The FETCH FIRST ROW ONLY clause can be used in the SELECT INTO statement when the query can result in more than a single row. The clause indicates that only one row should be retrieved regardless of how many rows might be in the result table. When a number is explicitly specified, it must be 1.

Using the `FETCH FIRST ROW ONLY` clause to explicitly limit the result table to a single row provides a way for the `SELECT INTO` statement to be used with a query that returns more than a single row. Using the clause helps you to avoid using a cursor when you know that you want to retrieve only one row. To influence which row is returned, you can use the *order-by-clause*. When you specify *order-by-clause*, the rows of the result are ordered and then the first row is returned. If the `FETCH FIRST ROW ONLY` clause is not specified and the result table contains more than a single row, an error occurs.

Notes

Number of rows inserted: If the `SELECT INTO` statement of the cursor contains an SQL data change statement, the `SELECT INTO` operation sets `SQLERRD(3)` to the number of rows inserted.

Using locators: Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to host variables with `CHAR`, `VARCHAR`, `GRAPHIC`, or `VARGRAPHIC` data types.

Examples

Example 1: Put the maximum salary in `DSN8910.EMP` into the host variable `MAXSALRY`.

```
EXEC SQL SELECT MAX(SALARY)
        INTO :MAXSALRY
        FROM DSN8910.EMP;
```

Example 2: Put the row for employee 528671, from `DSN8910.EMP`, into the host structure `EMPREC`.

```
EXEC SQL SELECT * INTO :EMPREC
        FROM DSN8910.EMP
        WHERE EMPNO = '528671'
END-EXEC.
```

Example 3: Put the row for employee 528671, from `DSN8910.EMP`, into the host structure `EMPREC`. Assume that the row will be updated later and should be locked when the query executes.

```
EXEC SQL SELECT * INTO :EMPREC
        FROM DSN8910.EMP
        WHERE EMPNO = '528671'
        WITH RS USE AND KEEP EXCLUSIVE LOCKS
END-EXEC.
```

SET CONNECTION

The SET CONNECTION statement establishes the database server of the process by identifying one of its existing connections.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required.

Syntax

<pre>➤➤—SET CONNECTION—location-name host-variable➤➤</pre>
--

Description

location-name **or** *host-variable*

Identifies the SQL connection by the specified location name or the location name contained in the host variable. If a host variable is specified:

- It must be a character string variable with a length attribute that is not greater than 16. (A C NUL-terminated character string can be up to 17 bytes.)
- It must not be followed by an indicator variable.
- The location name must be left-justified within the host variable and must conform to the rules for forming an ordinary location identifier.
- If the length of the location name is less than the length of the host variable, it must be padded on the right with blanks.

Let S denote the specified location name or the location name contained in the host variable. S must identify an existing SQL connection of the application process. If S identifies the current SQL connection, the state of S and all other connections of the application process are unchanged. The following rules apply when S identifies a dormant SQL connection.

If the SET CONNECTION statement is successful:

- SQL connection S is placed in the current state.
- S is placed in the CURRENT SERVER special register.
- Information about server S is placed in the SQLERRP field of the SQLCA. If the server is an IBM product, the information has the form *pppvrrm*, where:
 - *ppp* is:
 - ARI for DB2 Server for VSE & VM
 - DSN for DB2 for z/OS
 - QSQ for DB2 for i
 - SQL for all other DB2 products
 - *vv* is a two-digit version identifier such as '09'.

- *rr* is a two-digit release identifier such as '01'.
- *m* is a one-digit maintenance level such as '5' (Values 0, 1, 2, 3, and 4 are for maintenance levels in compatibility and enabling-new-function mode. Values 5, 6, 7, 8, and 9 are for maintenance levels in new-function mode.)

For example, if the server is Version 9 of DB2 for z/OS in new-function mode with the latest maintenance, the value of SQLERRP is 'DSN09015'.

- Any previously current SQL connection is placed in the dormant state.

If the SET CONNECTION statement is unsuccessful, the connection state of the application process and the states of its SQL connections are unchanged.

Notes

The use of CONNECT (Type 1) statements does not prevent the use of SET CONNECTION, but the statement either fails or does nothing because dormant SQL connections do not exist. The SQLRULES(DB2) bind option does not prevent the use of SET CONNECTION, but the statement is unnecessary because CONNECT (Type 2) statements can be used instead. Use the SET CONNECTION statement to conform to the SQL standard.

When an SQL connection is used, made dormant, and then restored to the current state in the same unit of work, the status of locks, cursors, and prepared statements for that SQL connection reflects its last use by the application process.

If the SET CONNECTION statement contains host variables, the contents of the host variables are assumed to be in the encoding scheme that was specified in the ENCODING parameter when the package or plan that contains the statement was bound.

Example

Execute SQL statements at TOROLAB1, execute SQL statements at TOROLAB2, and then execute more SQL statements at TOROLAB1.

```
EXEC SQL CONNECT TO TOROLAB1;

-- execute statements referencing objects at TOROLAB1

EXEC SQL CONNECT TO TOROLAB2;

-- execute statements referencing objects at TOROLAB2

EXEC SQL SET CONNECTION TOROLAB1;

-- execute statements referencing objects at TOROLAB1
```

The first CONNECT statement creates the TOROLAB1 connection, the second CONNECT statement places it in the dormant state, and the SET CONNECTION statement returns it to the current state.

SET CURRENT APPLICATION ENCODING SCHEME

The SET CURRENT APPLICATION ENCODING SCHEME statement assigns a value to the CURRENT APPLICATION ENCODING SCHEME special register. This special register allows users to control which encoding scheme will be used for dynamic SQL statements after the SET statement has been executed.

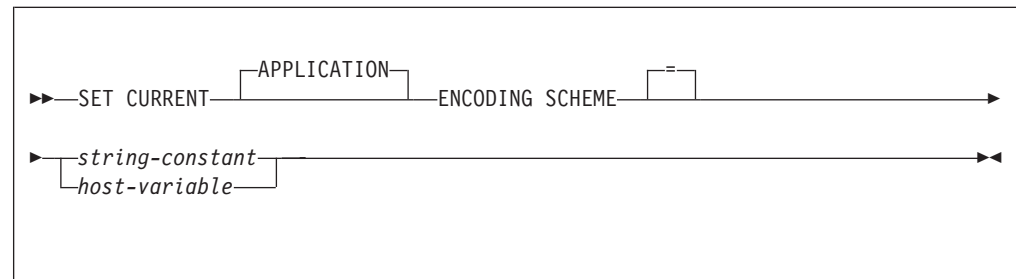
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

string-constant

A character string constant that represents a valid encoding scheme (ASCII, EBCDIC, UNICODE, or a character representation of a number between 1 and 65533).

host variable

A variable with a data type of CHAR or VARCHAR. The value of *host-variable* must not be null and must represent a valid encoding scheme or a character representation of a number between 1 and 65533). An associated indicator variable must not be provided.

The value must:

- Be left justified within the host variable
- Be padded on the right with blanks if its length is less than that of the host variable

Examples

The following examples set the CURRENT APPLICATION ENCODING SCHEME special register to 'EBCDIC' (in the second example, Host variable HV1 = 'EBCDIC').

```
EXEC SQL SET CURRENT APPLICATION ENCODING SCHEME = 'EBCDIC';  
EXEC SQL SET CURRENT ENCODING SCHEME = :HV1;
```

SET CURRENT DEBUG MODE

The SET CURRENT DEBUG MODE statement assigns a value to the CURRENT DEBUG MODE special register.

The special register sets the default value for the DEBUG MODE option for the following statements:

- CREATE PROCEDURE statements that define a native SQL or Java procedure
- ALTER PROCEDURE statements that create or replace a version of a native SQL procedure

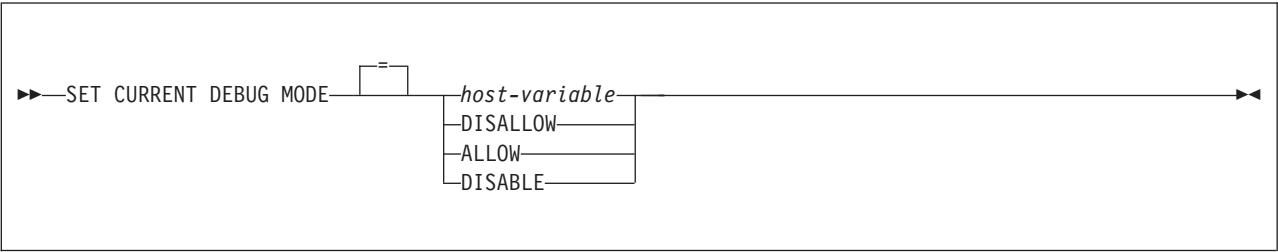
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

host-variable

Specifies a host variable that contains the debugging option. The host variable must conform to the following rules:

- Be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC variable. The actual length of the contents of the host variable must not exceed the length of the special register.
- Include a keyword value of DISALLOW, ALLOW, or DISABLE that is left justified
- Be padded on the right with blanks if the host variable is a fixed length character
- Not contain lowercase letters or characters that cannot be specified in an ordinary identifier
- Not be empty or contain only blanks
- Not be the null value

DISALLOW

Specifies that DISALLOW DEBUG MODE is the default option for CREATE PROCEDURE statements when defining an SQL or Java procedure, or ALTER PROCEDURE statements that create or replace a version of a native SQL procedure.

ALLOW

Specifies that ALLOW DEBUG MODE is the default option for CREATE PROCEDURE statements when defining an SQL or Java procedure, or ALTER PROCEDURE statements that create or replace a version of a native SQL procedure.

DISABLE

Specifies that DISABLE DEBUG MODE is the default option for CREATE PROCEDURE statements when defining an SQL or Java procedure, or ALTER PROCEDURE statements that add a version of a native SQL procedure.

Examples

Example: The following statement sets the CURRENT DEBUG MODE special register so that the default option for CREATE PROCEDURE statements will be ALLOW DEBUG MODE:

```
SET CURRENT DEBUG MODE = ALLOW;
```

SET CURRENT DECFLOAT ROUNDING MODE

The SET CURRENT DECFLOAT ROUNDING MODE statement assigns a value to the CURRENT DECFLOAT ROUNDING MODE special register. The special register sets the default rounding mode that is used with decimal floating point values (DECFLOAT).

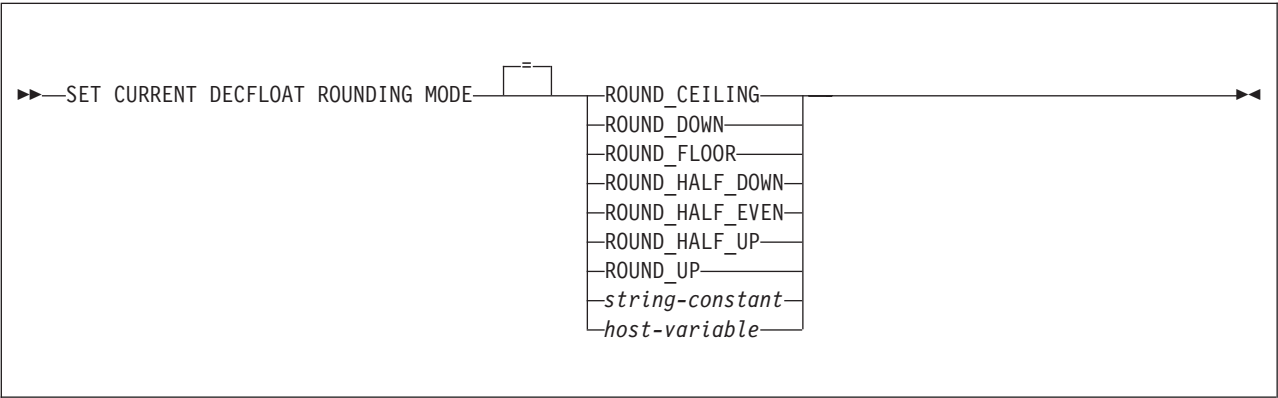
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

ROUND_CEILING

Round towards positive infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by 1 (round up).

ROUND_DOWN

Round towards 0 (truncation). The discarded digits are ignored.

ROUND_FLOOR

Round towards negative infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1 (round down).

ROUND_HALF_DOWN

Round to nearest value; if values are equidistant, rounds down. If the discarded digits represent greater than half (0.5) of the value of a number in the next left position, the result coefficient is incremented by 1 (round up). Otherwise, the discarded digits are ignored. This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.

ROUND_HALF_EVEN

Round to nearest value; if values are equidistant, round so that the final digit

is even. If the discarded digits represent greater than half (0.5) of the value of a number in the next left position, the result coefficient is incremented by 1 (round up). If the discarded digits represent less than half of the value, the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise, the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 (round up) if its rightmost digit is odd (to make an even digit).

ROUND_HALF_UP

Round to nearest value; if values are equidistant, round up. If the discarded digits represent greater than or equal to half (0.5) of the value of a number in the next left position, the result coefficient is incremented by 1 (round up). Otherwise the discarded digits are ignored.

ROUND_UP

Round away from 0. If all of the discarded digits are zero, the result is unchanged other than the removal of discarded digits. Otherwise, the result coefficient is incremented by 1 (round up). This rounding mode is not recommended when creating a portable application because it is not supported by the IEEE draft standard for floating-point arithmetic.

string-constant

Specifies a string constant that contains a specification of the rounding mode. The string-constant must have the following characteristics:

- Must be a string constant. The actual length of the contents of the string constant, after trailing blanks have been removed, must not exceed 19 characters.
- Must not be the null value.
- Must not contain lower case letters or characters that cannot be specified in an ordinary identifier.
- Must specify one of the seven rounding mode keywords as a string constant.

host-variable

Specifies a variable that contains a specification of the rounding mode. The variable must have the following characteristics:

- Must have a length, after trailing blanks have been removed, that does not exceed 19 bytes.
- Must not be followed by an indicator variable.
- Must not be a CLOB or DBCLOB.
- Must include a rounding mode that is left justified and conforms to the rules for forming an ordinary identifier.
- Must not contain lower case letters or characters that cannot be specified in an ordinary identifier.
- Must be padded on the right with blanks if the variable is a fixed length string.
- Must contain one of the seven rounding mode keywords.

Examples

Example: The following statement sets the CURRENT DECFLOAT ROUNDING MODE to ROUND_CEILING, using a string constant and a keyword.

```
SET CURRENT DECFLOAT ROUNDING MODE = ROUND_CEILING;
```

SET CURRENT DEGREE

The SET CURRENT DEGREE statement assigns a value to the CURRENT DEGREE special register.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

<pre>▶▶—SET CURRENT DEGREE—=— <i>string-constant</i> <i>host-variable</i> └──▶▶</pre>

Description

The value of CURRENT DEGREE is replaced by the value of the string constant or host variable. The value must be a character string that is not longer than 3 bytes and the value must be 'ANY', '1', or '1 '.

Notes

If the value of CURRENT DEGREE is '1' when a query is dynamically prepared, the execution of that query will not use parallel operations. If the value of CURRENT DEGREE is 'ANY' when a query is dynamically prepared, the execution of that query can involve parallel operations.

For distributed applications, the default value at the server is used unless the requesting application issues the SQL statement SET CURRENT DEGREE. For requests using DRDA, the SET CURRENT DEGREE statement must be within the scope of the CONNECT statement.

The value specified in the SET CURRENT DEGREE statement remains in effect until it is changed by the execution of another SET CURRENT DEGREE statement or until deallocation of the application process. For applications that connect to DB2 using the call attachment facility, the value of register CURRENT DEGREE can be requested to remain in effect for a longer duration. For more information, see the description of the call attachment facility CONNECT statement in *DB2 Application Programming and SQL Guide*.

Examples

Example 1: The following statement inhibits parallel operations:

```
SET CURRENT DEGREE = '1';
```

Example 2: The following statement allows parallel operations:

```
SET CURRENT DEGREE = 'ANY';
```

SET CURRENT LOCALE LC_CTYPE

The SET CURRENT LOCALE LC_CTYPE statement assigns a value to the CURRENT LOCALE LC_CTYPE special register. The special register allows control over the LC_CTYPE locale for statements that use a built-in function that refers to a locale, such as LCASE, UCASE, and TRANSLATE (with a single argument).

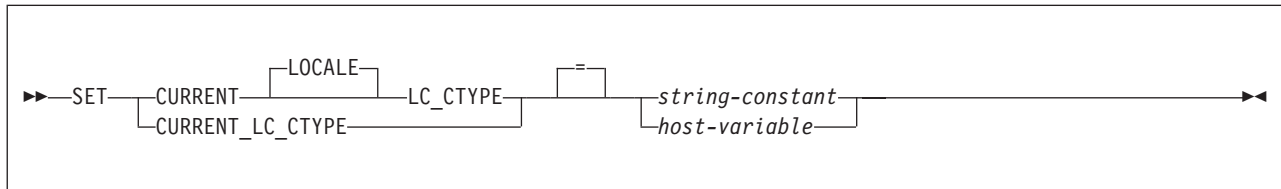
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of CURRENT LOCALE LC_CTYPE is replaced by the string constant or host variable specified. The value must be a CHAR or VARCHAR character string that is no longer than 50 bytes.

The value of CURRENT LOCALE LC_CTYPE is replaced by the value specified. The value must not be longer than 50 bytes and must be a valid locale.

string-constant

A character string constant that must not be longer than 50 bytes and must represent a valid locale.

host-variable

A variable with a data type of CHAR or VARCHAR and a length that is not longer than 50 bytes. The value of *host-variable* must not be null and must represent a valid locale. If the host variable has an associated indicator variable, the value of the indicator variable must not indicate a null value.

The locale must:

- Be left justified within the host variable
- Be padded on the right with blanks if its length is less than that of the host variable

A locale can be specified in uppercase characters, lowercase characters, or a combination of the two. For information on locales and their naming conventions, see *z/OS C/C++ Programming Guide*. Some examples of locales include:

Fr_BE
Fr_FR@EURO
En_US
Ja_JP

For Unicode data, there are three options for this special register:

- blank — simple casting on A-Z, a-z, and fullwidth Latin lowercase letters a-z and fullwidth Latin capital letters A-Z. Characters with diacritics are not affected.
- "UNI" — If the value "UNI" is specified, casting will use both the "NORMAL" and "SPECIAL" casing capabilities as described in *z/OS Support for Unicode™: Using Conversion Services*.
- a locale — In this case, locale specific casing will be performed using the "LOCALE" casing capabilities as described in *z/OS Support for Unicode™: Using Conversion Services*.. See the preceding paragraph for examples of specifying a locale.

Examples

Example 1: Set the CURRENT LOCALE LC_CTYPE special register to the locale 'En_US'.

```
EXEC SQL SET CURRENT LOCALE LC_CTYPE = 'En_US';
```

Example 2: Set the CURRENT LOCALE LC_CTYPE special register to the value of host variable HV1, which contains 'Fr_FR@EURO'.

```
EXEC SQL SET CURRENT LOCALE LC_CTYPE = :HV1;
```

SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement changes the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.

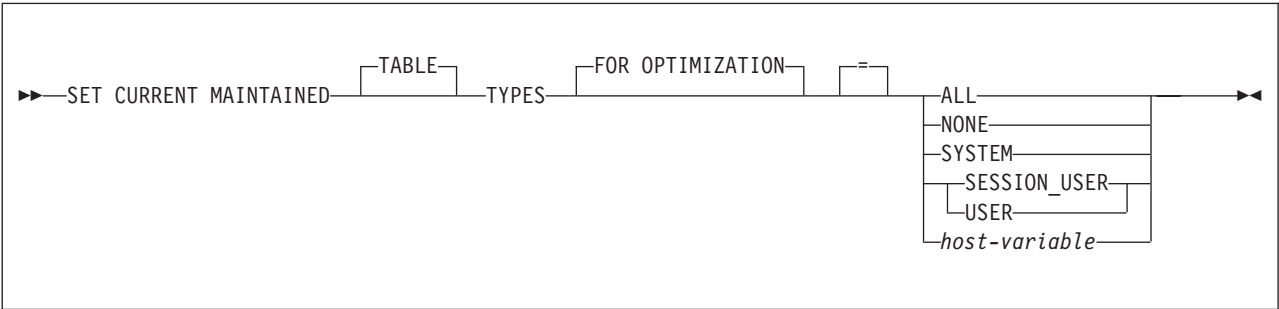
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value indicates which materialized query tables that are enabled for optimization are considered when optimizing the processing of dynamic SQL queries.

ALL

Indicates that all materialized query tables will be considered.

NONE

Indicates that no materialized query tables will be considered.

SYSTEM

Indicates that only system-maintained materialized query tables that are refresh deferred will be considered.

SESSION_USER or USER

Indicates that only user-maintained materialized query tables that are refresh deferred will be considered.

host-variable

A variable of type CHAR or VARCHAR. The length of the contents of *host-variable* must not exceed 255 bytes. It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value.

The characters of *host-variable* must be left justified. The content of the host variable must be a string that would match what can be specified as keywords for the special register in the exact case intended as there is no conversion to uppercase characters.

Notes

The CURRENT REFRESH AGE special register needs to be set to a value other than zero in order for the specified types of objects to be considered for optimizing the processing of dynamic SQL queries.

The CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register affects dynamic statement cache matching.

Examples

Example 1: The following statement sets the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register:

```
SET CURRENT MAINTAINED TABLE TYPES ALL;
```

Example 2: The following example retrieves the current value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register into the host variable called CURMAINTYPES.

```
EXEC SQL VALUES (CURRENT MAINTAINED TABLE TYPES) INTO :CURMAINTYPES;
```

The value would be ALL if set by the previous example.

Example 3: The following example resets the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register so that no materialized query tables can be considered to optimize the processing of dynamic SQL queries.

```
SET CURRENT MAINTAINED TABLE TYPES NONE;
```

SET CURRENT OPTIMIZATION HINT

The SET CURRENT OPTIMIZATION HINT statement assigns a value to the CURRENT OPTIMIZATION HINT special register.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

<pre>▶▶—SET CURRENT OPTIMIZATION HINT—=— string-constant host-variable —————▶▶</pre>

Description

The value of special register CURRENT OPTIMIZATION HINT is replaced by the value of the string constant or host variable. The value must be a character string that is not longer than 128 bytes.

Notes

Using the OPTIMIZATION HINT special register: The CURRENT OPTIMIZATION HINT special register specifies whether optimization hints are used in determining the access path of dynamic statements. See “CURRENT OPTIMIZATION HINT” on page 141 for additional information. An empty string or all blanks indicates that DB2 uses normal optimization techniques and ignores optimization hints.

Example

Example 1: Assume that string constant 'NOHYB' identifies a user-defined optimization hint in owner.PLAN_TABLE. Set the CURRENT OPTIMIZATION HINT special register so that DB2 uses this optimization hint to generate the access path for dynamic statements.

```
SET CURRENT OPTIMIZATION HINT = 'NOHYB';
```

If you set the register this way, DB2 validates and considers information in the rows in owner.PLAN_TABLE where the value in the OPTHINT column matches 'NOHYB' for dynamic SQL statements.

Example 2: Clear the CURRENT OPTIMIZATION HINT special register by specifying an empty string.

```
SET CURRENT OPTIMIZATION HINT = '';
```

SET CURRENT PACKAGE PATH

The SET CURRENT PACKAGE PATH statement assigns a value to the CURRENT PACKAGE PATH special register.

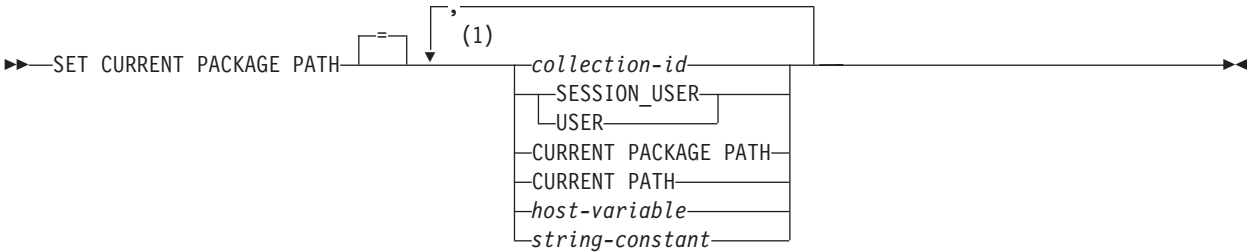
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Notes:

- 1 *SESSION_USER* (or *USER*), *CURRENT PACKAGE PATH*, and *CURRENT PATH* can each be specified only once on the right side of the statement.

Description

The value of CURRENT PACKAGE PATH is replaced by the values specified.

collection-id

Identifies a collection. *collection-id* must not be a delimited identifier that is empty or contains only blanks.

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register.

CURRENT PACKAGE PATH

Specifies the value of the CURRENT PACKAGE PATH special register before the execution of the SET CURRENT PACKAGE PATH statement.

CURRENT PATH

Specifies the value of the CURRENT PATH special register.

host-variable

Specifies a host variable that contains one or more collection IDs, separated by commas. The host variable must:

- Have a data type of CHAR or VARCHAR. The actual length of the contents of the host variable must not exceed the maximum length of the CURRENT PACKAGE PATH special register.
- Not be the null value if an indicator variable is provided.

- Contain an empty or blank string, or one or more collection IDs that are separated by commas.
- Be padded on the right with blanks if the host variable is fixed-length, or if the actual length of the host variable is longer than the content.
- Not contain a delimited identifier that is empty or contains only blanks.

string-constant

Specifies a string constant that contains one or more collection IDs, separated by commas. The string constant must:

- Have a length that does not exceed the maximum length of the CURRENT PACKAGE PATH special register.
- Contain an empty or blank string, or one or more collection IDs separated by commas.
- Not contain a delimited identifier that is empty or contains only blanks.

Notes

Contents of host variable or string constant: The contents of a host variable or string constant are interpreted as a list of collection IDs if the value contains at least one comma. If multiple collection IDs are specified, they must be separated by commas. Each collection ID in the list must conform to the rules for forming an ordinary identifier or be specified as a delimited identifier.

Checking for the existence of collections: No validation that the collections exist is made at the time that the CURRENT PACKAGE PATH special register is set. For example, a collection ID that is misspelled is not detected, which could affect the way subsequent SQL operates. At package execution time, authorization to the specific package is checked, and if this authorization check fails, an error is issued.

Resulting contents of the special register: The special register string is built by taking each collection ID specified and removing trailing blanks, delimiting with double quotation marks, doubling any double quotation marks within the collection ID as necessary, and then separating each collection ID by a comma. If the same collection ID appears more than once in the list, the first occurrence of the collection is used, and a warning is issued. The length of the resulting list cannot exceed the length of the special register. For example, assume that the following statements are issued:

```
SET CURRENT PACKAGE PATH = MYPKGS, "ABC E", SYSIBM
SET :HVPKLIST = CURRENT PACKAGE PATH
```

These statements result in the value of the host variable being set to: "MYPKGS", "ABC E", "SYSIBM".

A collection ID that does not conform to the rules for an ordinary identifier must be specified as a delimited collection ID and must not be specified within a host variable or string constant.

Considerations for keywords: A difference exists between specifying a single keyword, such as SESSION_USER, as a single keyword or as a delimited identifier. To indicate that the current value of a special register that is specified as a single keyword should be used in the package path, specify the name of the special register as a keyword. If you specify the name of the special register as a delimited identifier, it is interpreted as a collection ID of that value. For example, assume that the current value of the SESSION_USER special register is SMITH and that the following statement is issued:

```
SET CURRENT PACKAGE PATH = SYSIBM, SESSION_USER, "USER"
```

The result is that the value of the CURRENT PACKAGE PATH special register is set to: "SYSIBM, "SMITH", "USER".

Specifying a collection ID in an SQL procedure: Because a host variable (SQL variable) in an SQL procedure does not begin with a colon, DB2 uses the following rules to determine whether a value that is specified in a SET PACKAGE PATH = *name* statement is a variable or a collection ID:

- If *name* is the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a parameter or SQL variable and assigns the value in *name* to the package path.
- If *name* is not the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a collection ID and assigns and the value in *name* is the package path.

DRDA classification: The SET CURRENT PACKAGE PATH statement is executed by the database server and, therefore, is classified as a non-local SET statement in DRDA. The SET CURRENT PACKAGE PATH statement requires a new level of DRDA support. If SET CURRENT PACKAGE PATH is issued when connected to the local server, the SET CURRENT PACKAGE PATH special register at the local server is set. Otherwise, when SET CURRENT PACKAGE PATH is issued when connected to a remote server, the SET CURRENT PACKAGE PATH special register at the remote server is set.

Examples

Example 1: Set the CURRENT PACKAGE PATH special register to the list of collections COLL4 and COLL5, where :hvar1 contains the value COLL4,COLL5:

```
SET CURRENT PACKAGE PATH :hvar1;
```

The value of CURRENT PACKAGE PATH is set to the following two collection IDs: "COLL4","COLL5".

Example 2: Set the CURRENT PACKAGE PATH special register to the list of collections: COLL1, COLL#2, COLL3, COLL4, and COLL5, where :hvar1 contains the value COLL4,COLL5:

```
SET CURRENT PACKAGE PATH = "COLL1","COLL#2","COLL3", :hvar1;
```

The value of CURRENT PACKAGE PATH is set to the following five collection IDs: "COLL1","COLL#2","COLL3","COLL4","COLL5".

Example 3: Clear the CURRENT PACKAGE PATH special register.

```
SET CURRENT PACKAGE PATH = ' ';
```

Example 4: In preparation of calling a stored procedure that is named SUMARIZE, temporarily add two collections, COLL_PROD1" and "COLL_PROD2, to the end of the CURRENT PACKAGE PATH special register (the values of the collections are in host variables :prodcoll1 and prodcoll2, respectively). Because the stored procedure SUMARIZE is not defined with a COLLID value and is defined with INHERIT SPECIAL REGISTERS, the stored procedure will inherit the value of CURRENT PACKAGE PATH. When the stored procedure returns, set the value of the CURRENT PACKAGE PATH special register back to its original value.

```
SET :oldCPP = CURRENT PACKAGE PATH;  
SET CURRENT PACKAGE PATH = CURRENT PACKAGE PATH, :prodcol11, :prodcol12;  
CALL SUMARIZE(:V1,:V2);  
SET CURRENT PACKAGE PATH = :oldCPP;
```

SET CURRENT PACKAGESET

The SET CURRENT PACKAGESET statement assigns a value to the CURRENT PACKAGESET special register.

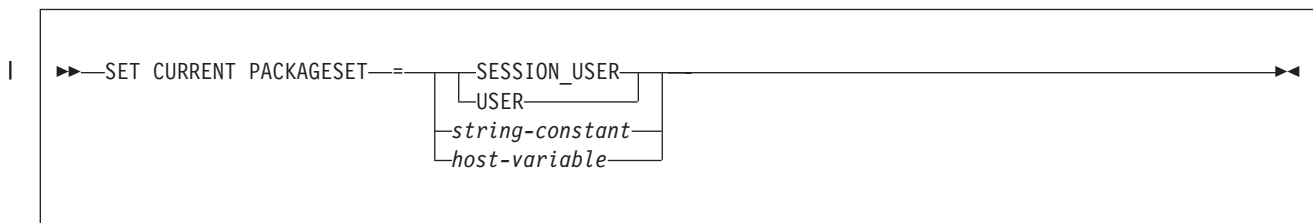
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of CURRENT PACKAGESET is replaced by the value of the SESSION_USER special register, *string-constant*, or *host-variable*. The value specified by *string-constant* or *host-variable* must be a character string that is not longer than 128 bytes.

Notes

Selection of plan elements: A *plan element* is a DBRM that has been bound into the plan or a package that is implicitly or explicitly identified in the package list of the plan. Plan elements contain the control structures used to execute certain SQL statements.

Since a plan can have many elements, one of the first steps involved in the execution of an SQL statement that requires a control structure is the selection of the plan element that contains its control structure. The information used by DB2 to select plan elements includes the value of CURRENT PACKAGESET.

SET CURRENT PACKAGESET is used to specify the collection ID of a package that exists at the current server. SET CURRENT PACKAGESET is optional and should not be used without an understanding of the following rules for selecting a plan element.

If the CURRENT PACKAGESET special register is an empty string, DB2 searches for a DBRM or a package in one of these sequences:

At the local location (if CURRENT SERVER is blank or explicitly names that location), the order is:

1. All DBRMs bound directly to the plan
2. All packages that have already been allocated for the application process

3. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan. The order of search is the order those packages are named in the package list.

At a remote location, the order is:

1. All packages that have already been allocated for the application process at that location
2. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan, whose locations match the value of CURRENT SERVER. The order of search is the order those packages are named in the package list.

If the special register CURRENT PACKAGESET is set, DB2 skips the check for programs that are part of the plan and uses the value of CURRENT PACKAGESET as the collection. For example, if CURRENT PACKAGESET contains COL5, then DB2 uses COL5.PROG1.timestamp for the search. For additional information, see *DB2 Application Programming and SQL Guide*.

DRDA classification: SET CURRENT PACKAGESET is executed by the requester and is therefore classified as a local SET statement in DRDA.

CURRENT PACKAGESET special register with stored procedures and user-defined functions: The initial value of the CURRENT PACKAGESET special register in a stored procedure or user-defined function is the value of the COLLID parameter with which the stored procedure or user-defined function was defined. If the routine was defined without a value for the COLLID parameter, the value of the special register is inherited from the calling program. A stored procedure or user-defined function can use the SET CURRENT PACKAGESET statement to change the value of the special register. This allows the routine to select the version of the DB2 package that is used to process the SQL statements in a called routine that is not defined with a COLLID value.

When control returns from the stored procedure to the calling program, the special register CURRENT PACKAGESET is restored to the value it contained before the stored procedure was called.

Examples

Example 1: Limit the plan element selection to packages in the PERSONNEL collection at the current server.

```
EXEC SQL SET CURRENT PACKAGESET = 'PERSONNEL';
```

Example 2: Eliminate collections as a factor in plan element selection.

```
EXEC SQL SET CURRENT PACKAGESET = '';
```

SET CURRENT PRECISION

The SET CURRENT PRECISION statement assigns a value to the CURRENT PRECISION special register.


Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



```

  >>—SET CURRENT PRECISION—=—
                                |
                                | string-constant
                                |
                                | host-variable
                                |
                                <<

```

Description

This statement replaces the value of the CURRENT PRECISION special register with the value of the string constant or host variable. The value must be a character string 5 bytes in length. The value must be 'DEC15,' 'DEC31,' or 'Dpp.s', where 'pp' is either 15 or 31 and 's' is a number between 1 and 9. If the form 'Dpp.s' is used, 'pp' represents the precision that will be used with the rules that are used for DEC15 or DEC31, and 's' represents the minimum divide scale to use for division operations. The separator used in the form 'Dpp.s' can be either the '.' or the ',' character, regardless of the setting of the default decimal point.

Example

Set the CURRENT PRECISION special register so that subsequent statements that are prepared use DEC15 rules for decimal arithmetic.

```
EXEC SQL SET CURRENT PRECISION = 'DEC15';
```


SET CURRENT REFRESH AGE

The SET CURRENT REFRESH AGE statement changes the value of the CURRENT REFRESH AGE special register.

The CURRENT REFRESH AGE value corresponding to ANY (99 999 999 999 999) cannot be used in timestamp arithmetic operations because the result would be outside the valid range of dates.

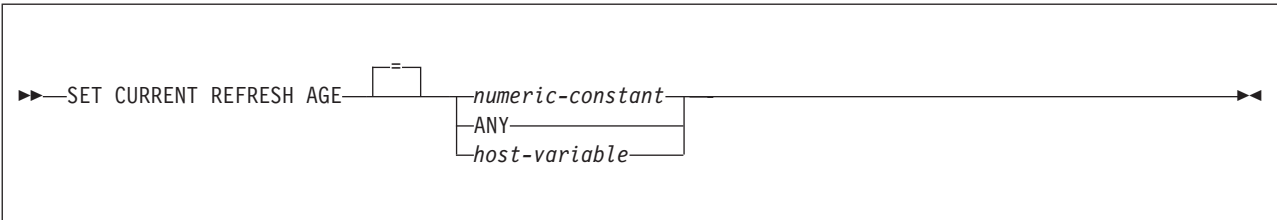
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

numeric-constant

A DECIMAL(20,6) value representing a timestamp duration. The value must be 0 or 99 999 999 999 999, the partial seconds of which is ignored and thus can be any value.

0 Indicates that query optimization using materialized query tables will not be attempted.

9999999999999999

Indicates that any materialized query tables identified by the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register may be used to optimize the processing of a query. This value represents 9999 years, 99 months, 99 days, 99 hours, 99 minutes, and 99 seconds.

ANY

Shorthand for 9999999999999999.

host-variable

A variable of type DECIMAL(20,6) or other type that is assignable to DECIMAL(20,6). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value. The value of *host-variable* must be 0 or 99 999 999 999 999, the partial seconds of which is ignored and thus can be any value.

Notes

Materialized query tables created or altered with `DISABLE QUERY OPTIMIZATION` specified are not eligible for automatic query rewrite. Thus, they are not affected by the setting of this special register.

Setting the `CURRENT REFRESH AGE` special register to a value other than zero should be done with caution. Allowing a materialized query table that may not represent the values of the underlying base table to be used to optimize the processing of a query may produce results that do not accurately represent the data in the underlying table. This situation may be acceptable when you know the underlying data has not changed or you are willing to accept the degree of error in the results based on your knowledge of the data.

Examples

Example: Set the `CURRENT REFRESH AGE` special register to 99 999 999 999 999 to indicate that any materialized query tables identified by the `CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION` special register can be used to optimize the processing of a query.

```
SET CURRENT REFRESH AGE ANY;
```

SET CURRENT ROUTINE VERSION

The SET CURRENT ROUTINE VERSION statement assigns a value to the CURRENT ROUTINE VERSION special register. The special register sets the override value for the version identifier of native SQL procedures when they are invoked.

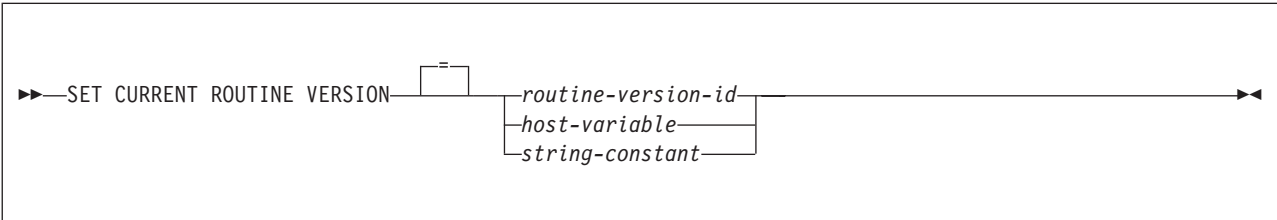
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

routine-version-id
Specifies a routine version identifier.

host-variable
Specifies a host variable that contains a version identifier. The host variable must conform to the following rules:

- Be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC variable. The actual length of the contents of the host variable must not exceed the length of a version identifier.
- Include a routine version identifier that is left justified and conforms to the rules for forming an ordinary identifier or a delimited identifier, or must be blank or empty.
- Be padded on the right with blanks if the host variable is a fixed length character.
- Not be empty or contain only blanks if the identifier is delimited.
- Not be the null value.

string-constant
Specifies a string constant that contains a version identifier. The string constant must conform to the following rules:

- Have a length that does not exceed the length of a *routine-version-id*.
- Include a routine version identifier that is left justified and conforms to the rules for forming an ordinary identifier or a delimited identifier, or must be blank or an empty string
- Not be empty or contain only blanks if the identifier is delimited

Notes

Resetting the special register: To reset the special register, specify an empty string constant, a string of blanks, or a host variable that is empty or contains only blanks. A routine version override is not in effect when the special register is reset.

Implications of using the special register: Setting the CURRENT ROUTINE VERSION special register to a version identifier will affect all SQL procedures that are subsequently invoked using CALL statements that specify the name of the procedure using a host variable, until the value of CURRENT ROUTINE VERSION is changed. If a version of the procedure that is identified by the version identifier in the special register exists for an SQL procedure that is being invoked, that version of the procedure is used. Otherwise, the currently active version of the procedure (as noted in the catalog) is used.

When you use the CURRENT ROUTINE VERSION special register to test a version of one or more native SQL procedures, you should use a routine version identifier that is a value other than the default value (V1) on the CREATE PROCEDURE statement. This will avoid having the special register affect more procedures that you intend when testing a new version of a procedure. For example, assume that you want to run version VER2 of procedure P1, and procedure P1 invokes another procedure, P2. If a version exists for both procedures P1 and P2 with the routine version identifier VER2, that version will be used for both procedures.

Examples

Example: The following statement sets the CURRENT ROUTINE VERSION special register so that the override value for the version identifier of native SQL procedures will be the value that is specified in the host variable *rvid*:

```
SET CURRENT ROUTINE VERSION = :rvid;
```

SET CURRENT RULES

The SET CURRENT RULES statement assigns a value to the CURRENT RULES special register.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

<pre>» SET CURRENT RULES = <i>string-constant</i> <i>host-variable</i> «</pre>
--

Description

This statement replaces the value of the CURRENT RULES special register with the value of the string constant or host variable. The value must be a character string that is 3 bytes in length, and the value must be 'DB2' or 'STD'.

Notes

For the effect of the values 'DB2' and 'STD' on the execution of certain SQL statements, see "CURRENT RULES" on page 145.

Example

Set the SQL rules to be followed to DB2.

```
EXEC SQL SET CURRENT RULES = 'DB2';
```

SET CURRENT SQLID

The SET CURRENT SQLID statement assigns a value to the CURRENT SQLID special register.

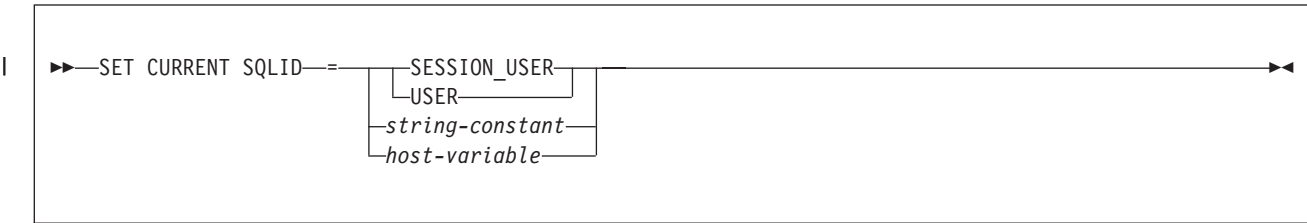
Invocation

| This statement can be embedded in an application program or issued interactively.
| It is an executable statement that can be dynamically prepared. The value to which
| special register CURRENT SQLID is set is used as the SQL authorization ID for
| dynamic SQL statements only if DYNAMICRULES run behavior is in effect. The
| CURRENT SQLID value is ignored for the other DYNAMICRULES behaviors.

Authorization

| If any of the authorization IDs of the process has SYSADM authority, CURRENT
| SQLID can be set to any value . Otherwise, the specified value must be equal to
| one of the authorization IDs of the application process. This rule always applies,
| even when SET CURRENT SQLID is a static statement.CURRENT SQLID cannot
| be set to the name of a role.

Syntax



Description

| The value of CURRENT SQLID is replaced by the value of SESSION_USER,
| string-constant, or host-variable. The value specified by a string-constant or
| host-variable must be a character string that contains 8 characters or less. Unless
| some authorization ID of the process has SYSADM authority, the value must be
| equal to one of the authorization IDs of the process.

Notes

Effect on authorization IDs: SET CURRENT SQLID does not change the primary authorization ID of the process.

If the SET CURRENT SQLID statement is executed in a stored procedure or user-defined function package that has a dynamic SQL behavior other than run behavior, the SET CURRENT SQLID statement does not affect the authorization ID that is used for dynamic SQL statements in the package. The dynamic SQL behavior determines the authorization ID. For more information, see the discussion of DYNAMICRULES in *DB2 Command Reference*.

Effect on special register CURRENT PATH: When the value of the PATH special register depends on the value of the CURRENT SQLID special register, any changes to the CURRENT SQLID special register are not reflected in the value of

the PATH special register until a commit operation is performed or a SET PATH statement is issued to change the SQL path to use the new value of the CURRENT SQLID.

DRDA classification: SET CURRENT SQLID is executed by the database server and is therefore classified as a non-local SET statement in DRDA.

Examples

Example 1: Set the CURRENT SQLID to the primary authorization ID.

```
SET CURRENT SQLID = SESSION_USER;
```

SET ENCRYPTION PASSWORD

The SET ENCRYPTION PASSWORD statement sets the value of the encryption password and, optionally, the password hint. The encryption and decryption built-in functions use this password and password hint for data encryption unless the functions are invoked with an explicitly specified password and hint. The password is not tied to DB2 authentication and is used only for data encryption.

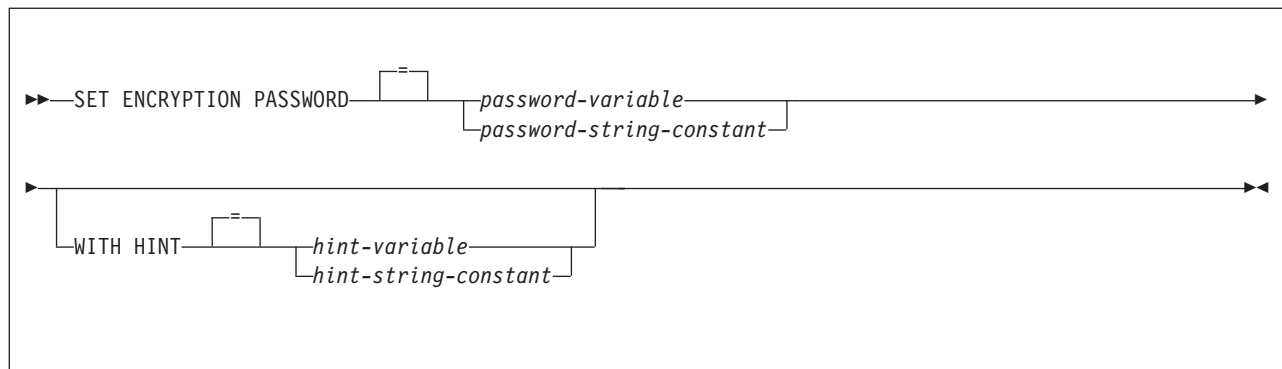
Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

No authorization is required to execute this statement.

Syntax



Description

password-variable

Specifies a variable that contains an encryption password. The variable:

- Must be a CHAR or VARCHAR variable. The actual length of the contents of the variable must be between 6 and 127 inclusive or must be an empty string. If an empty string is specified, the default encryption password is set to no value.
- Must not be the null value.
- All characters are case-sensitive and are not converted to uppercase characters.

password-string-constant

A character constant that contains an encryption password. The length of the constant must be between 6 and 127 inclusive or must be an empty string. If an empty string is specified, the default encryption password is set to no value. All characters are case-sensitive and are not converted to uppercase characters.

WITH HINT

Indicates that a value is specified that will help you remember passwords (for example, 'Ocean' as a hint to remember 'Pacific'). If a hint value is specified, the hint is used as the default for encryption functions. The hint can subsequently be retrieved for an encrypted value using the GETHINT function.

If this clause is not specified and a hint is not explicitly specified on the encryption function, no hint will be embedded in encrypted data result.

hint-variable

Specifies a variable that contains an encryption password hint. The variable:

- Must be a CHAR or VARCHAR variable. The actual length of the contents of the variable must not be greater than 32. If an empty string is specified, the default encryption password hint is set to an empty string.
- Must not be the null value.
- All characters are case-sensitive and are not converted to uppercase characters.

hint-string-constant

A character string constant that contains an encryption password hint. The length of the constant must not be greater than 32. If the value is an empty string, the default encryption password hint is set to an empty string.

Notes

Normal DB2 mechanisms are used to transmit the host variable or constant to the database server.

For more information about using this statement, see “ENCRYPT_TDES” on page 363 and “DECRYPT_BINARY, DECRYPT_BIT, DECRYPT_CHAR, and DECRYPT_DB” on page 346..

Examples

Example 1: Set the ENCRYPTION PASSWORD to the value in :hv1. Do not specify a hint for the password.

```
SET ENCRYPTION PASSWORD = :hv1
```

Example 2: Set the ENCRYPTION PASSWORD to the value in :hv1. Specify the value in :hv2 as the hint for the password.

```
SET ENCRYPTION PASSWORD = :hv1 WITH HINT :hv2
```

SET host-variable assignment

The SET host-variable assignment statement assigns values, either of expressions or NULL values, to host variables.

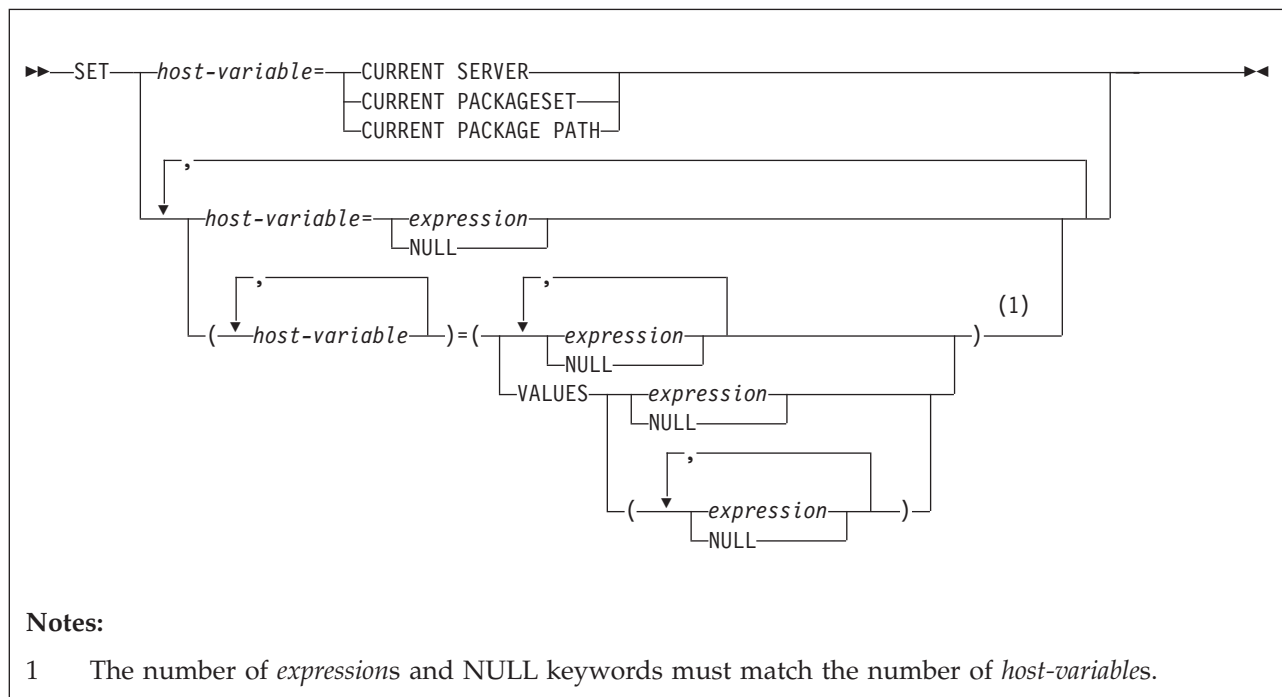
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The privileges that are held by the current authorization ID must include those required to execute any of the expressions.

Syntax



Description

host-variable

Identifies one or more host variables or transition variables that are used to receive the corresponding *expression* or NULL value on the right side of the statement.

For the triggered action of a CREATE TRIGGER statement, use the SET *transition-variable* instead.

The value to be assigned to each *host-variable* can be specified immediately following the item reference, for example, *host-variable* = *expression*, *host-variable*=*expression*. Or, sets of parentheses can be used to specify all the *host-variables* and then all the values, for example, (*host-variable*, *host-variable*) = (*expression*, *expression*).

host-variable

Identifies one or more host variables that are used to receive the corresponding *expression* or NULL value on the right side of the statement. Each host variable must be defined in the program as described under the rules for declaring host variables. A parameter marker must not be specified in place of *host-variable*.

The value to be assigned to each *host-variable* can be specified immediately following the host variable, for example, *host-variable* = *expression*, *host-variable* = *expression*. Or, sets of parentheses can be used to specify all the *host-variables* and then all the values, for example, (*host-variable*, *host-variable*) = (*expression*, *expression*).

expression

Specifies the value to be assigned to the corresponding *host-variable*. The expression is any expression of the type described in “Expressions” on page 180, except it cannot contain a reference to CURRENT PACKAGE PATH or to *local* special registers (CURRENT SERVER or CURRENT PACKAGESET).

All expressions are evaluated before any result is assigned to a host variable. If an expression refers to a host variable that is used in the host variable list, the value of the variable in the expression is the value of the variable prior to any assignments.

Each assignment to a host variable is made according to the assignment rules described in “Assignment and comparison” on page 102. Assignments are made in sequence through the list. When the *host-variables* are enclosed within parentheses, for example, (*host-variable*, *host-variable*, ...) = (*expression*, *expression*, ...), the first value is assigned to the first host variable in the list, the second value to the second host variable in the list, and so on.

NULL

Specifies the null value and can only be specified for host variables that have an associated indicator variable.

VALUES

Specifies the value to be assigned to the corresponding host variable. When more than one value is specified, the values must be enclosed in parentheses. Each value can be an expression or NULL, as described above. The following syntax is equivalent:

- (*host-variable*, *host-variable*) = (VALUES(*expression*, NULL))
- (*host-variable*, *host-variable*) = (*expression*, NULL)

Local special registers and the CURRENT PACKAGE PATH special register can be referenced only in a VALUES host-variable statement that results in the assignment of a single host variable and not those that result in setting more than one value.

Notes

The default encoding scheme for the data is the value in the bind option ENCODING, which is the option for application encoding. If this statement is used with functions such as LENGTH or SUBSTRING that are operating on LOB locators, and the LOB data that is specified by the locator is in a different encoding scheme from the ENCODING bind option, LOB materialization and character conversion occur. To avoid LOB materialization and character conversion, select the LOB data from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table.

Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to host variables with other data types. For more information on using locators, see *DB2 Application Programming and SQL Guide*.

Examples

Example 1: Set the host variable HVL to the value of the CURRENT PATH special register.

```
SET :HVL = CURRENT PATH;
```

Example 2: Set the host variable PATH to the contents of the SQL PATH special register, the host variable XTIME to the local time at the current server, and the host variable MEM to the current member of the data sharing environment.

```
SET :SERVER = CURRENT PATH,  
    :XTIME = CURRENT TIME,  
    :MEM = CURRENT MEMBER;
```

Example 3: Set the host variable DETAILS to a portion of a LOB value, using a LOB expression with a LOB locator to refer the extracted portion of the value.

```
SET :DETAILS = SUBSTR(:LOCATOR,1,35);
```

If the LOB data that is specified by the LOB locator LOCATOR is in a different encoding scheme from the value of the ENCODING bind option, and you want to avoid LOB materialization and character conversion, use the following statement instead of the SET statement:

```
SELECT SUBSTR(:LOCATOR,1,35)  
      INTO :DETAILS  
      FROM SYSIBM.SYSDUMMYU;
```

Example 4: Set host variable HV1 to the results of external function CALC_SALARY and host variable HV2 to the value of special register CURRENT PATH. Use an indicator value with HV1 in case CALC_SALARY returns a null value.

```
SET (:HV1:IND1, :HV2) =  
    (CALC_SALARY(:HV3, :HF4), CURRENT PATH);
```

SET PATH

The SET PATH statement assigns a value to the CURRENT PATH special register.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

SET

CURRENT

PATH

=

(1)

schema-name

SYSTEM PATH

SESSION_USER

USER

CURRENT

PATH

CURRENT PACKAGE PATH

host-variable

string-constant

Notes:

1SYSTEM PATH, SESSION_USER or USER, and CURRENT PATH can be specified only once each.

Description

The value of PATH is replaced by the values specified.

schema-name

Identifies a schema. DB2 does not verify that the schema exists. For example, a schema name that is misspelled is not detected, which could affect the way subsequent SQL operates.

SYSTEM PATH

Specifies the schema names "SYSIBM", "SYSFUN", or "SYSPROC".

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register.

PATH

Specifies the value of the CURRENT PATH special register before the execution of this statement.

CURRENT PACKAGE PATH

Specifies the value of the CURRENT PACKAGE PATH special register.

host-variable

A variable with a data type of CHAR or VARCHAR. The value of *host-variable* must not be null and must represent a valid schema name.

The schema name must:

- Be left justified within the host variable
- Be padded on the right with blanks if its length is less than that of the host variable

string-constant

A character string constant that represents a valid schema name.

If the schema name specified in *string-constant* will also be specified in other SQL statements and the schema name does not conform to the rules for ordinary identifiers, the schema name must be specified as a delimited identifier in the other SQL statements.

Notes

Restrictions on SET PATH: These restrictions apply to the SET PATH statement:

- If the same schema name appears more than once in the path, the first occurrence of the name is used and a warning is issued.
- The length of the CURRENT PATH special register limits the number of schema names that can be specified. The special register string is built by taking each schema name that is specified and removing trailing blanks, delimiting with double quotes, changing each double quote character to two double quote characters within the schema name as necessary, and then separating each schema name with a comma. The length of the resulting string cannot exceed 2048 bytes.

Specifying SYSIBM, SYSPROC, and SYSFUN: Schemas SYSIBM, SYSPROC, and SYSFUN do not need to be specified in the special register. If these schemas are not explicitly specified in the CURRENT PATH special register, each schema is implicitly assumed at the front of the SQL path; if any of these schemas are not specified, they are assumed in the order of SYSIBM, SYSPROC, SYSFUN (see “SQL path” on page 56 for an example). Only the schemas that are explicitly specified in the CURRENT PATH register are included in the 2048 byte limit.

To avoid having SYSIBM, SYSPROC, or SYSFUN implicitly added to the front of the SQL path, explicitly specify them in the path when setting the value of the register. If you specify them at the end of the path, DB2 will check all the other schemas in the path first.

Specifying keywords versus delimited identifiers: There is a difference between specifying a keyword and specifying a delimited identifier. For example, specifying SESSION_USER with and without escape characters. To indicate that the value of the SESSION_USER special register should be used in the SQL path, specify the keyword SESSION_USER. If you specify SESSION_USER is as a delimited identifier instead (for example, "SESSION_USER"), it is interpreted as a schema name of 'SESSION_USER'. For example, assume that the current value of the SESSION_USER special register is SMITH and that the following statement is issued:

```
SET PATH = SYSIBM, SYSPROC, SESSION_USER, "SESSION_USER"
```

The result is that the value of the SQL path is set to:
"SYSIBM","SYSPROC","SMITH","SESSION_USER".

Specifying a schema name in an SQL procedure: Because a host variable (SQL variable) in an SQL procedure does not begin with a colon, DB2 uses the following rules to determine whether a value that is specified in a SET PATH=*name* statement is a variable or a *schema-name*:

- If *name* is the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a parameter or SQL variable and assigns the value in *name* to PATH.
- If *name* is not the same as a parameter or SQL variable in the SQL procedure, DB2 uses *name* as a *schema-name* and assigns the value *name* to PATH.

The use of the path to resolve object names: For information on when the SQL path is used to resolve unqualified data type, function, and procedure names and when the CURRENT PATH special register provides the SQL path, see “SQL path” on page 56.

DRDA classification: The SET PATH statement is executed by the database server and, therefore, is classified as a non-local SET statement in DRDA.

Alternative syntax and synonyms: For compatibility with previous releases of DB2 or other products in the DB2 family, DB2 supports CURRENT FUNCTION PATH or CURRENT_PATH as a synonym for CURRENT PATH. CURRENT_PATH is consistent with the SQL standard name of the special register.

Examples

Example 1: Set the CURRENT PATH special register to the list of schemas: "SCHEMA1", "SCHEMA#2", "SYSIBM".

```
SET PATH = SCHEMA1,"SCHEMA#2", SYSIBM;
```

When the special register provides the SQL path, SYSPROC which was not explicitly specified in the special register, is implicitly assumed at the front of the SQL path, making the effective value of the path:

```
SYSPROC, SCHEMA1, SCHEMA#2, SYSIBM
```

Example 2: Add schema SMITH and SYSPROC to the value of the CURRENT PATH special register that was set in Example 1.

```
SET PATH = CURRENT PATH, SMITH, SYSPROC;
```

The value of the special register becomes:

```
SCHEMA1, SCHEMA#2, SYSIBM, SMITH, SYSPROC
```

SET SCHEMA

The SET SCHEMA statement assigns a value to the CURRENT SCHEMA special register. If the package is bound with the DYNAMICRULES BIND option, this statement does not affect the qualifier that is used for unqualified database object references.

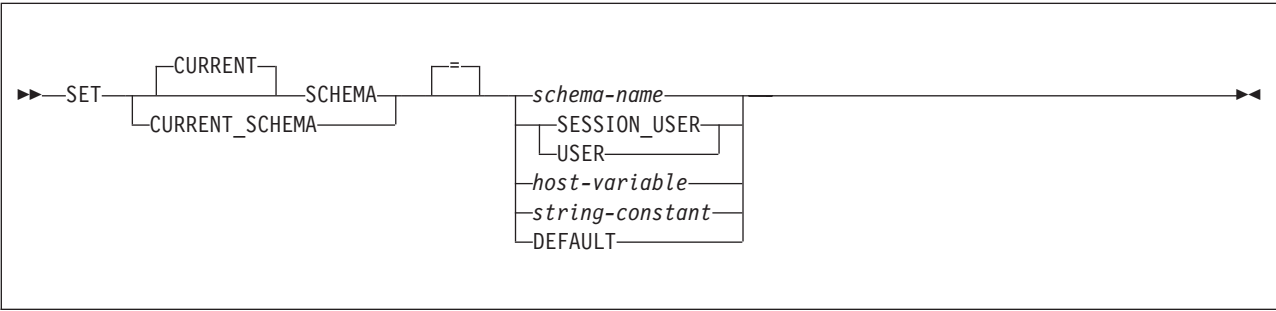
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

schema-name

Identifies a schema. No validation that the schema exists is made at the time the CURRENT SCHEMA is set. For example, if a schema name is misspelled, it could affect the way subsequent SQL operates.

SESSION_USER or USER

Specifies the value of the SESSION_USER (USER) special register.

host-variable

Specifies a host variable that contains a schema name. The content is not folded to uppercase.

The host variable must:

- Be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC variable. The actual length of the contents of the *host-variable* must not exceed the length of a schema name.
- Not be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value.
- Include a schema name that is left justified and conforms to the rules for forming an ordinary identifier or delimited identifier. If the identifier is delimited, it must not be empty or contain only blanks.
- Be padded on the right with blanks if the host variable is fixed length.
- Not contain SESSION_USER, USER, or DEFAULT.

string-constant

Specifies a string constant that contains a schema name. The content is not folded to uppercase.

The string constant must:

- Have a length that does not exceed the maximum length of a schema name.
- Include a schema name that is left justified and conforms to the rules for forming an ordinary identifier or delimited identifier. If the identifier is delimited, it must not be empty or contain only blanks.
- Not contain SESSION_USER, USER, or DEFAULT.

DEFAULT

Specifies that CURRENT SCHEMA is to be set to its initial value, as if it had never been explicitly set during the application process. For information about the initial value of CURRENT SCHEMA, see “CURRENT SCHEMA” on page 147.

Notes

Considerations for keywords: There is a difference between specifying a single keyword (such as SESSION_USER or DEFAULT) as a single keyword or as a delimited identifier. To indicate that the current value of the SESSION_USER special register should be used for setting the current schema, specify SESSION_USER as a keyword. To indicate that the special register should be set to its default value, specify DEFAULT as a keyword. If SESSION_USER or DEFAULT is specified as a delimited identifier instead (for example, "SESSION_USER"), it is interpreted as a schema name of that value ("SESSION_USER").

Transaction considerations: The SET SCHEMA statement is not a committable operation. ROLLBACK has no effect on CURRENT SCHEMA.

Usage of the assigned value: The value of the CURRENT SCHEMA special register, as set by this statement, is used as the schema name in all dynamic SQL statements. The QUALIFIER bind option specifies the schema name for use as the qualifier for unqualified database object names in static SQL statements.

Impact on other special registers: Setting the CURRENT SCHEMA special register does not affect any other special register. Therefore, the CURRENT SCHEMA is not included in the SQL path that is used to resolve the schema name for unqualified references to function, procedures and user-defined types in dynamic SQL statements. To include the current schema value in the SQL path, whenever the SET SCHEMA statement is issued, also issue the SET PATH statement including the schema name from the SET SCHEMA statement.

Examples

Example 1: The following statement sets the CURRENT SCHEMA special register.

```
EXEC SQL SET SCHEMA RICK;
```

Example 2: The following example retrieves the current value of the CURRENT SCHEMA special register into the host variable called CURSCHEMA.

```
EXEC SQL SELECT CURRENT SCHEMA INTO :CURSCHEMA  
FROM SYSIBM.SYSDUMMY;
```

The value of the host variable is RICK.

Example 3: Assume that the following statements are issued:

```
SET CURRENT SQLID = 'USRT001';  
SET CURRENT SCHEMA = 'USRT002';
```

At this point, the two special registers contain different values. Any subsequent CREATE statements will use USRT002 as the implicit qualifier, but the owner of the newly created objects is USRT001.

Example 4: Assume that the value of CURRENT SCHEMA is 'Jane' and that the default value of the PATH special register was established using that value (that is, the value of PATH is "SYSIBM", "SYSFUN", "SYSPROC", "Jane"). Change the value of the CURRENT SCHEMA special register to 'John'.

```
SET CURRENT SCHEMA = 'JOHN';
```

To change the SQL path to use the updated CURRENT SCHEMA value of 'John', issue a SET PATH statement to change the value of the PATH special register. Alternatively, a commit would cause PATH to be re-initialized. Otherwise, the path remains "SYSIBM", "SYSFUN", "SYSPROC", "Jane"), which might cause unqualified object names to resolve to 'Jane' when you want them to resolve to 'John'.

SET transition-variable assignment

The SET transition-variable assignment statement assigns values, either of expressions or NULL values, to transition variables.

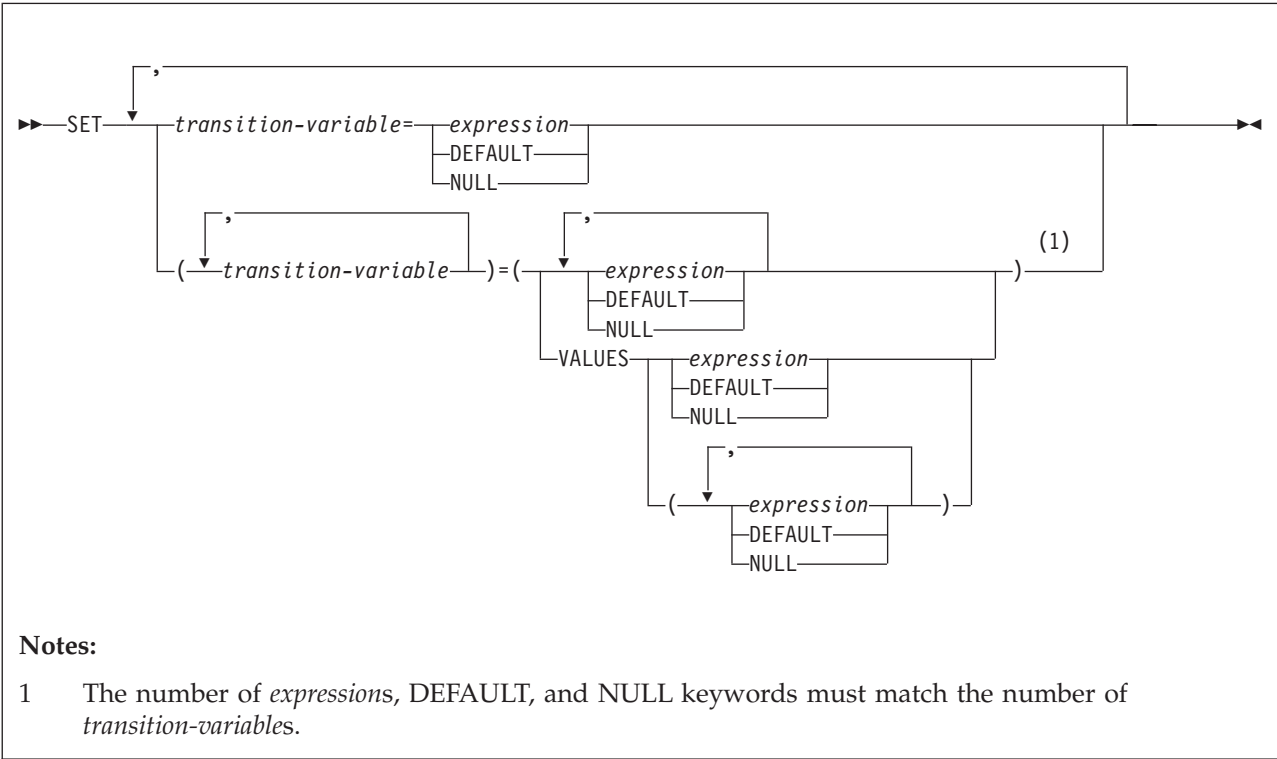
Invocation

This statement can be used as a triggered SQL statement in the triggered action of a before trigger whose granularity is FOR EACH ROW.

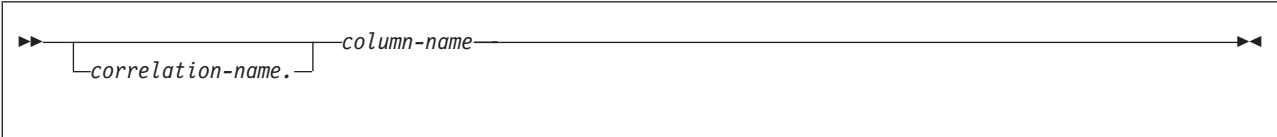
Authorization

The privileges that are held by the current authorization ID must include those required to execute any of the expressions or assignments to transition variables.

Syntax



transition-variable:



Description

transition-variable

Identifies a column in the set of affected rows for the trigger that is used to receive the corresponding *expression*, default value, or NULL value on the right side of the statement.

The value to be assigned to each *transition-variable* can be specified immediately following the transition variable, for example, *transition-variable = expression*, *transition-variable=expression*. Or, sets of parentheses can be used to specify all the *transition-variables* and then all the values, for example, *(transition-variable, transition-variable) = (expression, expression)*.

correlation-name

Identifies the correlation name given for referencing the NEW transition variables. The name must match the correlation name specified following NEW in the REFERENCING clause of the CREATE TRIGGER statement.

If OLD is not specified in the REFERENCING clause, *correlation-name* defaults to the correlation name following NEW.

column-name

Identifies the column to be updated. The name must identify a column of the subject table. The name can identify a column that is defined as GENERATED BY DEFAULT but not one defined as GENERATED ALWAYS, unless the DEFAULT keyword is specified on the right side of the statement. You must not specify the same column more than once.

The effect of a SET *transition-variable* statement is equivalent to the effect of an SQL UPDATE statement.

expression

Specifies the value to be assigned to the corresponding *transition-variable*. The expression is any expression of the type described in “Expressions” on page 180. A reference to a *local special register* is the value of that special register at the *server* when the trigger body is activated. If the expression contains a scalar fullselect, the scalar fullselect cannot reference columns of the triggering table. The expression cannot include an aggregate function except when it occurs within a scalar fullselect.

An expression can contain references to OLD and NEW transition variables that are qualified with a correlation name.

All expressions are evaluated before any result is assigned to a transition variable. If an expression refers to a transition variable that is used in the list of transition variables, the value of the variable in the expression is the value of the variable prior to any assignments.

Each assignment to a transition variable column is made according to the assignment rules described in “Assignment and comparison” on page 102. Assignments are made in sequence through the list. When the *transition-variables* are enclosed within parentheses, for example, *(transition-variable, transition-variable, ...) = (expression, expression, ...)*, the first value is assigned to the first transition variable in the list, the second value to the second transition variable in the list, and so on.

DEFAULT

Specifies that the default value is used based on how the corresponding column is defined in the table. The value that is assigned depends on how the column is defined.

- If the column is defined using the IDENTITY clause, the column is generated by the DB2 system.
- If the column is defined as a row change timestamp column, the column value is generated by the DB2 system.

- If the column is defined using the WITH DEFAULT clause, the value is set to the default that is defined for the column.
- If the column is defined without specifying the WITH DEFAULT clause, the GENERATED clause, or the NOT NULL clause, the value is NULL
- If the column is specified in the INCLUDE column list, the column value is set to null.

A ROWID column must not be set to the DEFAULT keyword.

An identity column or a row change timestamp column that is defined as GENERATED ALWAYS can be set only to the DEFAULT keyword.

If the column is defined using the NOT NULL clause and the GENERATED clause is not used, or the WITH DEFAULT clause is not used, the DEFAULT keyword cannot be specified for that column.

NULL

Specifies the null value and can only be specified for nullable transition variables.

VALUES

Specifies the value to be assigned to the corresponding transition variable. When more than one value is specified, the values must be enclosed in parentheses. Each value can be an expression or NULL, as described above. The following syntax is equivalent:

- *(transition-variable, transition-variable) = (VALUES(expression, NULL))*
- *(transition-variable, transition-variable) = (expression, NULL)*

Examples

Example 1: Assume that you want to create a before trigger that sets the salary and commission columns to default values for newly inserted rows in the EMPLOYEE table and that you will define the trigger only with NEW in the REFERENCING clause. Write the SET transition-variable statement that assigns the default values to the SALARY and COMMISSION columns.

```
SET (SALARY, COMMISSION) = (50000, 8000);
```

Example 2: Assume that you want to create a before trigger that detects any commission increases greater than 10% for updated rows in the EMPLOYEE table and limits the commission increase to 10%. You will define the trigger with both OLD and NEW in the REFERENCING clause. Write the SET transition-variable statement that limits an increase to the COMMISSION column to 10%.

```
SET NEWROW.COMMISSION = 1.1 * OLDROW.COMMISSION;
```

SIGNAL

The `SIGNAL` statement is used to signal an error. It causes an error to be returned with the specified `SQLSTATE` and error description.

For a description of the statement, see “`SIGNAL` statement” on page 1582.

TRUNCATE

The DB2 TRUNCATE statement deletes all rows for either base tables or declared global temporary tables. The base table can be in a simple table space, a segmented table space, a partitioned table space, or a universal table space. If the table contains LOB or XML columns, the corresponding table spaces and indexes are also truncated.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privilege set that is defined below must include at least one of the following privileges:

- The DELETE privilege for the table
- Ownership of the table
- DBADM authority for the database
- SYSADM authority

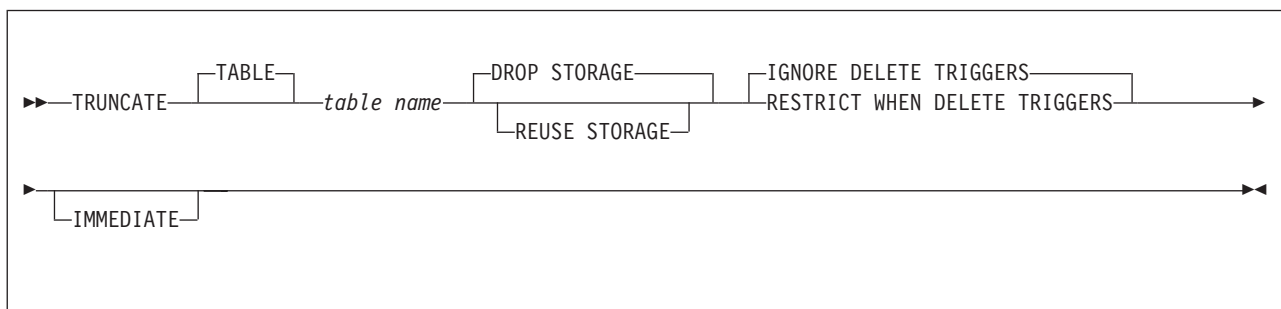
If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

Additionally, if the IGNORE DELETE TRIGGERS option is specified, the privilege set must include at least one of the following privileges:

- The ALTER privilege for the table
- Ownership of the table
- DBADM authority for the database
- SYSADM authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more details on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64.)

Syntax



Description

table-name

Identifies the table that is to be truncated. The name must identify a table that exists at the current server. The name must not identify the following objects:

- a view
- an auxiliary table
- an XML table
- a catalog table

If *table-name* is a base table of a table space, all tables that are defined under the table will also be truncated (for example: auxiliary LOB table spaces and XML table spaces), and all of its associated indexes will also be truncated.

DROP STORAGE or REUSE STORAGE

Specifies whether to drop or reuse the existing storage that is allocated for the table.

DROP STORAGE

Specifies that all storage that is allocated for the table is released and made available for use for the same table or any other table that resides in the table space. The scope of **DROP STORAGE** is always at the table space level and the deallocated space is always available for reuse by all tables in the table space.

DROP STORAGE is the default.

REUSE STORAGE

Specifies that all storage that is allocated for the table will be emptied, but will continue to be allocated for the table. **REUSE STORAGE** is ignored for a table in a simple table space and the statement is processed as if **DROP STORAGE** is specified.

RESTRICT WHEN DELETE TRIGGERS or IGNORE DELETE TRIGGERS

Specifies what to do when delete triggers are defined on the table.

RESTRICT WHEN DELETE TRIGGERS

Specifies that an error is returned if delete triggers are defined on the table.

IGNORE DELETE TRIGGERS

Specifies that any delete triggers that are defined for the table are not activated by the truncate operation.

IGNORE DELETE TRIGGERS is the default.

IMMEDIATE

Specifies that the truncate operation is processed immediately and cannot be undone. If the **IMMEDIATE** option is specified, the table must not contain any uncommitted updates. In the case of a table in a simple table space, the entire table space must not contain any uncommitted updates or the truncate operation will fail. Also, if any uncommitted CREATE or ALTER statement has been executed, the truncate operation will fail.

The truncated table is immediately available for use in the same unit of work. Although a ROLLBACK statement is allowed to execute after a TRUNCATE statement, the truncate operation is not undone, and the table remains in a truncated state. For example, if another data change operation is done on the table after the TRUNCATE IMMEDIATE statement and then the ROLLBACK statement is executed, the truncate operation will not be undone, but all other data change operations are undone.

If **IMMEDIATE** is not specified, a **ROLLBACK** statement can undo the truncate operation.

The **IMMEDIATE** option can be specified for a table in a segmented table space or a universal table space which allows deallocated spaces to be reclaimed immediately for subsequent insert operations in the same unit of work without committing the truncate operation.

Notes

Rules and restrictions: The truncate operation cannot be executed if the table is a parent table in a defined referential constraint. The DB2 subsystem issues an error when it detects the existence of rule violations. Therefore, if the referential integrity constraint exists, the **TRUNCATE** statement will be restricted regardless of whether the child table contains rows.

If the **TRUNCATE** statement is used on a tables where any of the following conditions is true, the truncate operation will perform in a similar way to a mass delete operation:

- Tables with Change Data Capture (CDC) attribute

The DB2 subsystem allows a table with the CDC-enabled attribute to be truncated without imposing any new restrictions.

- Tables with multi-level security

If the table contains a column that is defined as a security label, the truncate operation needs to examine each row to determine if the security label of the authorization ID or role has the authority to delete that row. However, if the primary authorization ID or role has write-down privilege, verification of each row in the table is not necessary.

- Tables with a **VALIDPROC** attribute

If a **VALIDPROC** is defined for the table, the truncate operation needs to verify the validity of each row in the table.

TRUNCATE and table spaces that are not logged: The **TRUNCATE TABLE** statement can be used to remove a table space from the logical page list and to reset recover-pending status. When the table space is segmented or universal, the table is the only table in the table space, and the table does not have a **VALIDPROC**, referential constraints, delete triggers, or a **SECURITY LABEL** column, use the **TRUNCATE TABLE** statement to empty the table and the table space will be removed from the LPL and recover-pending status will be reset.

Examples

Example 1: Empty an unused inventory table regardless of any existing triggers and return its allocated space.

```
TRUNCATE TABLE INVENTORY
DROP STORAGE
IGNORE DELETE TRIGGERS;
```

Example 2: Empty an unused inventory table regardless of any existing delete triggers but preserve its allocated space for later reuse.

```
TRUNCATE TABLE INVENTORY
REUSE STORAGE
IGNORE DELETE TRIGGERS;
```

| *Example 3:* Empty an unused inventory table permanently (a ROLLBACK statement
| cannot undo the truncate operation when the **IMMEDIATE** option is specified)
| regardless of any existing delete triggers and preserve its allocated space for
| immediate use.

| TRUNCATE TABLE INVENTORY
| REUSE STORAGE
| IGNORE DELETE TRIGGERS
| IMMEDIATE;

|

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table or view or activates an instead of update trigger. Updating a row of a view updates a row of the table on which the view is based if no instead of update trigger is defined for the update operation on the view. If such a trigger is defined, the trigger is activated instead of the UPDATE statement. The table or view can exist at the current server or at any DB2 subsystem with which the current server can establish a connection.

There are two forms of this statement:

- The *searched* UPDATE form is used to update one or more rows optionally determined by a search condition.
- The *positioned* UPDATE form specifies that one or more rows corresponding to the current cursor position are to be updated.

Invocation

This statement can be embedded in an application program or issued interactively. A positioned UPDATE can be embedded in an application program. Both forms are executable statements that can be dynamically prepared.

Authorization

Authority requirements depend on whether the object identified in the statement is a user-defined table, a catalog table for which updates are allowed, or a view, and whether SQL standard rules are in effect:

When a user-defined table is identified: The privilege set must include at least one of the following:

- The UPDATE privilege on the table
- The UPDATE privilege on each column to be updated
- Ownership of the table
- DBADM authority on the database that contains the table
- SYSADM authority

If the database is implicitly created, the database privileges must be on the implicit database or on DSNDB04.

When a catalog table is identified: The privilege set must include at least one of the following:

- The UPDATE privilege on each column to be updated
- DBADM authority on the catalog database
- SYSCTRL authority
- SYSADM authority

When a view is identified: The privilege set must include at least one of the following:

- The UPDATE privilege on the view
- The UPDATE privilege on each column to be updated
- SYSADM authority

When SQL standard rules are in effect: If SQL standard rules are in effect and an expression in the SET clause contains a reference to a column of the table or view, or if the search-condition in a searched UPDATE contains a reference to a column of the table or view, the privilege set must include at least one of the following:

- The SELECT privilege on the table or view
- SYSADM authority

SQL standard rules are in effect as follows:

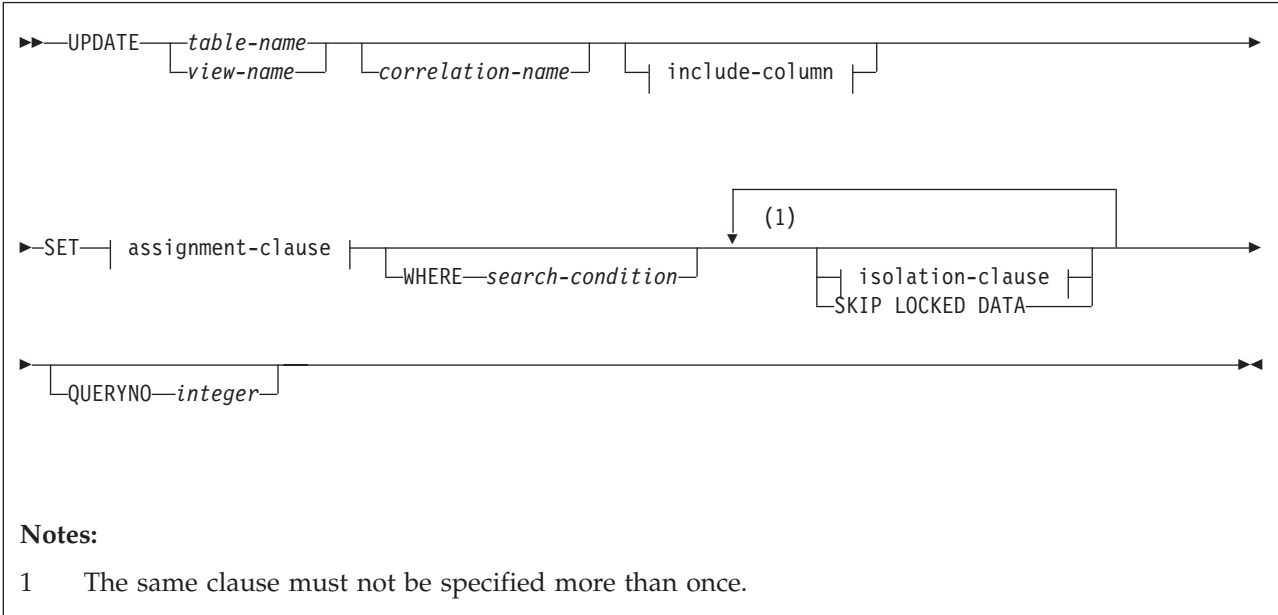
- For static SQL statements, if the SQLRULES(STD) bind option was specified
- For dynamic SQL statements, if the CURRENT RULES special register is set to 'STD'

The owner of a view, unlike the owner of a table, might not have UPDATE authority on the view (or might have UPDATE authority without being able to grant it to others). The nature of the view itself can preclude its use for UPDATE. For more information, see the discussion of authority in “CREATE VIEW” on page 1184.

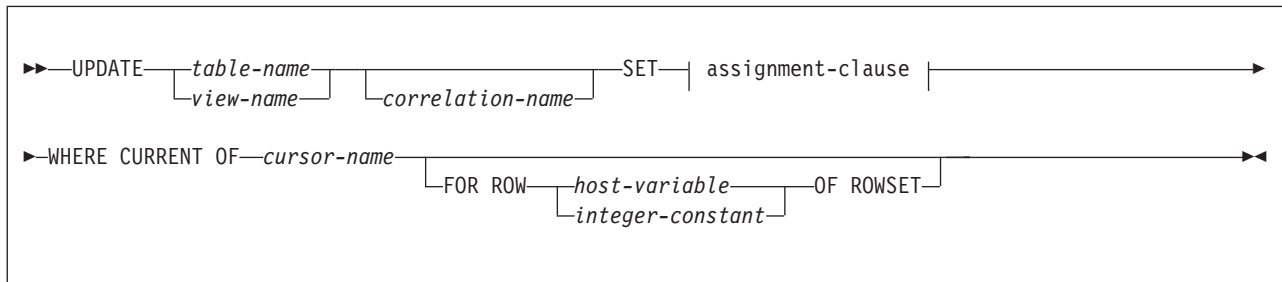
If a fullselect is specified, the privilege set must include authority to execute the fullselect. For more information about the authorization rules, see “Authorization” on page 630.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the owner of the plan or package. If the statement is dynamically prepared, the privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 81 on page 691. (For more information on these behaviors, including a list of the DYNAMICRULES bind option values that determine them, see “Authorization IDs and dynamic SQL” on page 64).

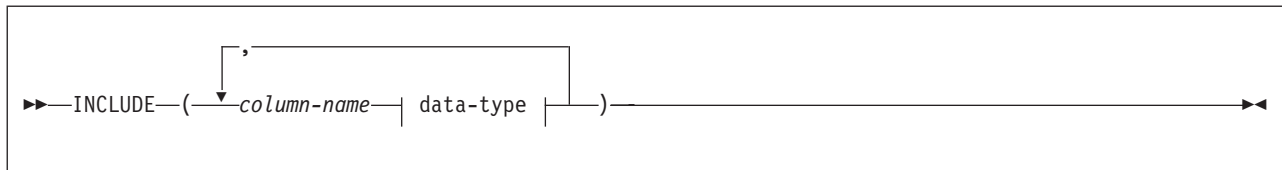
searched update:



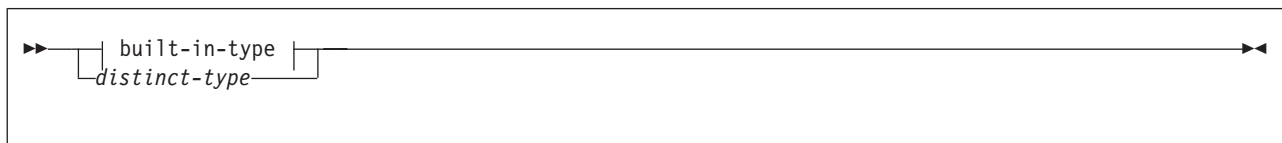
positioned update:



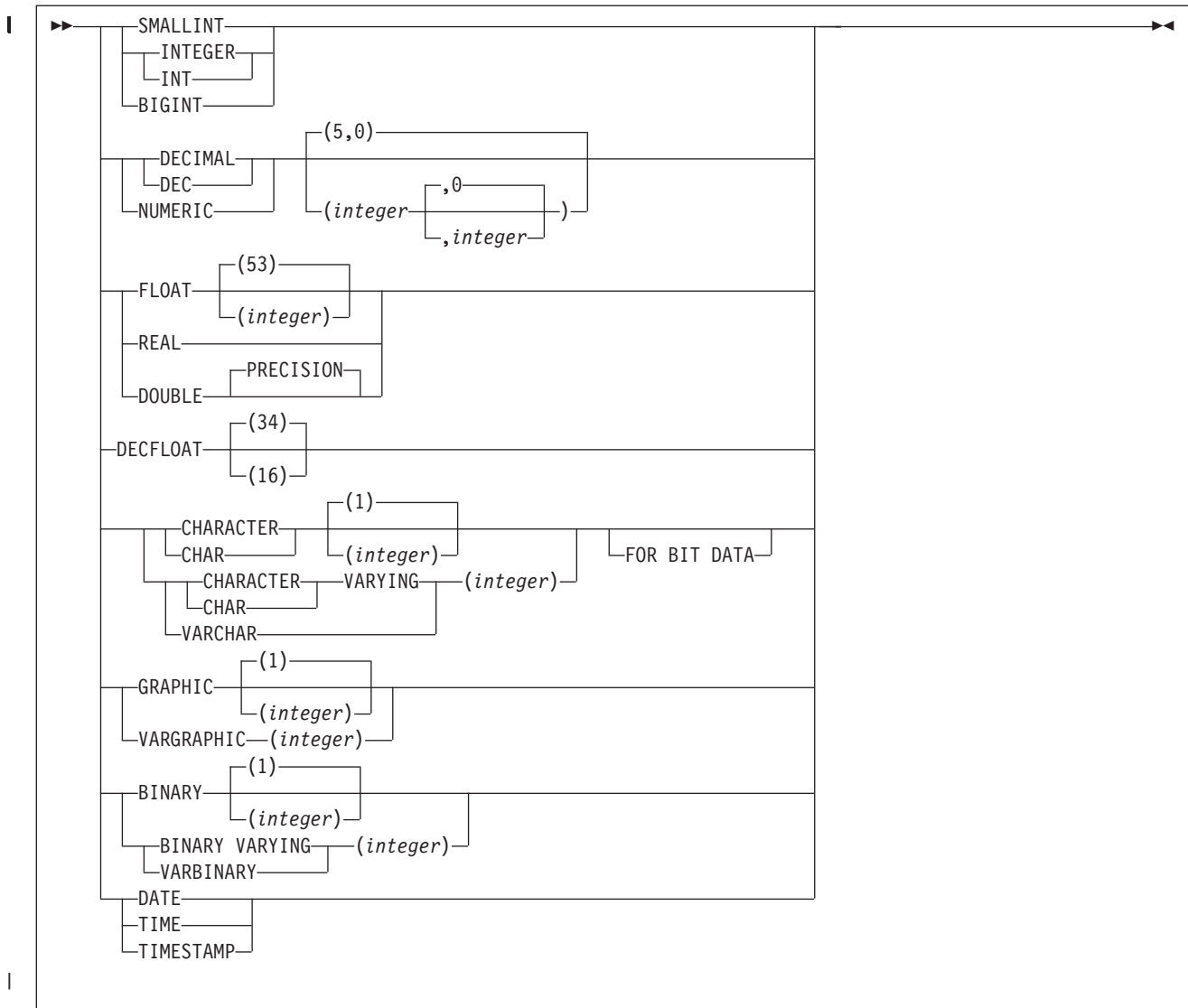
include-column:



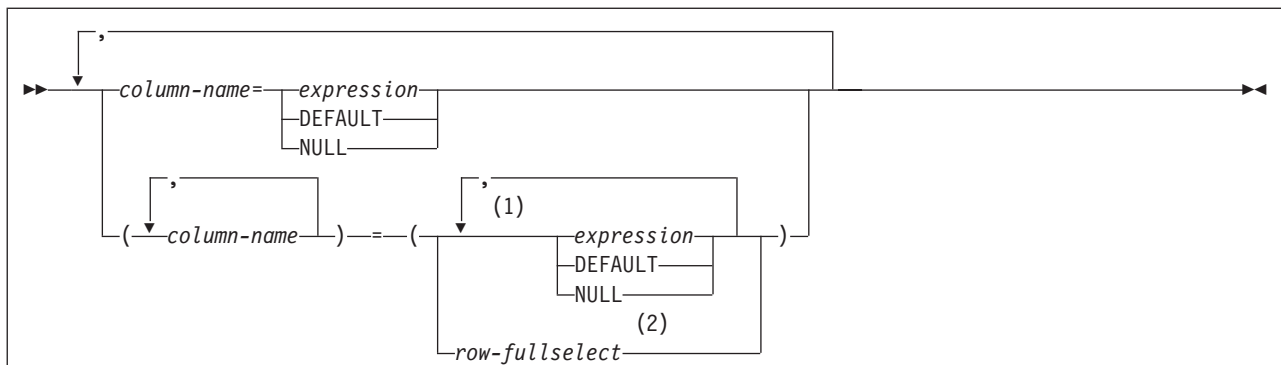
data-type:



built-in-type:



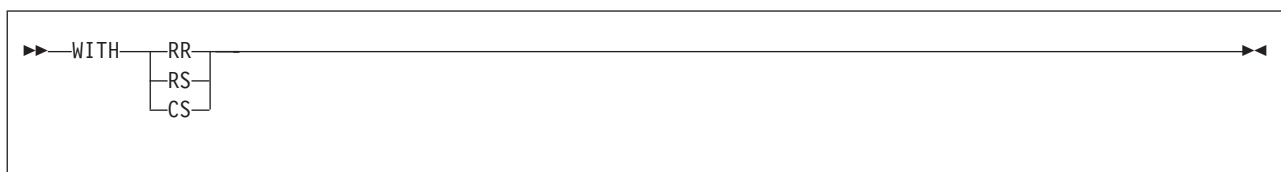
assignment clause:



Notes:

- 1 The number of *expressions*, *DEFAULT*, and *NULL* keywords must match the number of *column-names*.
- 2 The number of columns in the select list must match the number of *column-names*.

isolation-clause:



Description

table-name or *view-name*

Identifies the object of the UPDATE statement. The name must identify a table or view that exists at the DB2 subsystem identified by the implicitly or explicitly specified location name. The name must not identify:

- An auxiliary table
- A created temporary table or a view of a created temporary table
- A catalog table with no updatable columns or a view of a catalog table with no updatable columns
- A read-only view that has no INSTEAD OF trigger defined for its update operations. (For a description of a read-only view, see “CREATE VIEW” on page 1184.)
- A system-maintained materialized query table
- A table that is implicitly created for an XML column

In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must be a remote server that supports two-phase commit.

A catalog table or a view of a catalog table can be identified if every column identified in the SET clause is an updatable column. If a column of a catalog table is updatable, its description in “DB2 catalog tables” on page 1677 indicates that the column can be updated. If the object table is SYSIBM.SYSSTRINGS, any column other than IBMREQD can be updated, but the rows selected for update must be rows provided by the user (the value of the IBMREQD column is N) and only certain values can be specified as explained in *DB2 Administration Guide*.

correlation-name

Can be used within *search-condition* or *assignment-clause* to designate the table or view. (For an explanation of *correlation-name*, see “Correlation names” on page 154.)

include-column

Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the result table of the UPDATE statement when it is nested in the FROM clause of the outer fullselect that is used in a subselect, SELECT statement, or in a SELECT INTO statement. The included columns are appended to the end of the list of columns that is identified by *table-name* or *view-name*. If no value is assigned to a column that is specified by an *include-column*, a NULL value is returned for that column.

INCLUDE

Introduces a list of columns that are to be included in the result table of the UPDATE statement. The included columns are only available if the UPDATE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement.

column-name

Specifies the name for a column of the result table of the UPDATE statement that is not the same name as another included column nor a column in the table or view that is specified in *table-name* or *view-name*.

data-type

Specifies the data type of the included column. The included columns are nullable.

built-in-type

Specifies a built-in data type. See “CREATE TABLE” on page 1079 for a description of each built-in type.

distinct-type

Specifies a distinct type. Any length, precision, or scale attributes for the column are those of the source type of the distinct type as specified by using the CREATE TYPE statement.

SET

Introduces the assignment of values to column names.

assignment-clause

If *row-fullselect* is specified, the number of columns in the result of *row-fullselect* must match the number of *column-names* that are specified. If *row-fullselect* is not specified, the number of expressions, and NULL and DEFAULT keywords must match the number of *column-names* that are specified.

column-name

Identifies a column that is to be updated. *column-name* must identify a column of the specified table or view, and that column must be updatable. The column must not identify a generated column or a view column where the column is derived from a scalar function, constant, or expression. *column-name* can also identify an INCLUDE column that must not be qualified. The same column must not be specified more than one time.

Assignments to included columns are only processed when the UPDATE statement is nested in the FROM clause of a SELECT statement or a SELECT INTO statement. There must be at least one assignment clause

that specifies a *column-name* that is not an **INCLUDE** column. The null value is returned for an included column that is not set by using an explicit **SET** clause.

For a positioned update, allowable column names can be further restricted to those in a certain list. This list appears in the **FOR UPDATE** clause of the **SELECT** statement for the associated cursor. The clause can be omitted by using the conditions that are described in “Positioned updates of columns” on page 256.

A view column that is derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same **UPDATE** statement.

expression

Indicates the new value of the column. The *expression* is any expression of the type described in “Expressions” on page 180. It must not include an aggregate function.

A *column-name* in an expression must identify a column of the table or view. For each row that is updated, the value of the column in the expression is the value of the column in the row before the row is updated.

DEFAULT

Specifies that the default value is used based on how the corresponding column is defined in the table. The value that is assigned depends on how the column is defined.

- If the column is defined using the **IDENTITY** clause, the column is generated by the DB2 system.
- If the column is defined as a row change timestamp column, the column value is generated by the DB2 system.
- If the column is defined using the **WITH DEFAULT** clause, the value is set to the default that is defined for the column.
- If the column is defined without specifying the **WITH DEFAULT** clause, the **GENERATED** clause, or the **NOT NULL** clause, the value is **NULL**.
- If the column is specified in the **INCLUDE** column list, the column value is set to null.

A **ROWID** column must not be set to the **DEFAULT** keyword.

An identity column or a row change timestamp column that is defined as **GENERATED ALWAYS** can be set only to the **DEFAULT** keyword.

If the column is defined using the **NOT NULL** clause and the **GENERATED** clause is not used, or the **WITH DEFAULT** clause is not used, the **DEFAULT** keyword cannot be specified for that column.

NULL

Specifies the null value as the new value of the column. Specify **NULL** only for nullable columns.

row-fullselect

Specifies a fullselect that returns a single row. The column values are assigned to each of the corresponding *column-names*. If the fullselect returns no rows, the null value is assigned to each column; an error occurs if any column to be updated is not nullable. An error also occurs if there is more than one row in the result.

For a positioned update, if the table or view that is the object of the UPDATE statement is used in the fullselect, a column from the instance of the table or view in the fullselect cannot be the same as *column-name*, a column being updated.

If the fullselect refers to columns to be updated, the value of such a column in the fullselect is the value of the column in the row before the row is updated.

WHERE

Specifies the rows to be updated. You can omit the clause, give a search condition, or specify a cursor. If you omit the clause, all rows of the table or view are updated.

search-condition

Specifies any search condition described in Chapter 2, “Language elements,” on page 47. Each *column-name* in the search condition, other than in a subquery, must identify a column of the table or view.

The *search-condition* is applied to each row of the table or view and the updated rows are those for which the result of the *search-condition* is true. If the unique key or primary key is a parent key, the constraints are effectively checked at the end of the operation.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, a subquery with no correlated references is executed just once, whereas it is possible that a subquery with a correlated reference must be executed once for each row.

WHERE CURRENT OF *cursor-name*

Identifies the cursor to be used in the update operation. *cursor-name* must identify a declared cursor as explained in the description of the DECLARE CURSOR statement in “DECLARE CURSOR” on page 1191. If the UPDATE statement is embedded in a program, the DECLARE CURSOR statement must include *select-statement* rather than *statement-name*.

The object of the UPDATE statement must also be identified in the FROM clause of the SELECT statement of the cursor. The columns to be updated can be identified in the FOR UPDATE clause of that SELECT statement though they do not have to be identified. If the columns are not specified, the columns that can be updated include all the updatable columns of the table or view that is identified in the first FROM clause of the fullselect.

The result table of the cursor must not be read-only. For an explanation of read-only result tables, see Read-only cursors. Note that the object of the UPDATE statement must not be identified as the object of the subquery in the WHERE clause of the SELECT statement of the cursor.

When the UPDATE statement is executed, the cursor must be open and positioned on a row or rowset of the result table.

- If the cursor is positioned on a single row, that row is the one updated.
- If the cursor is positioned on a rowset, all rows corresponding to the rows of the current rowset are updated.

A positioned UPDATE must not be specified for a cursor that references a view on which an instead of update trigger is defined, even if the view is an updatable view.

FOR ROW *n* OF ROWSET

Specifies which row of the current rowset is to be updated. The corresponding row of the rowset is updated, and the cursor remains positioned on the current rowset.

host-variable or *integer-constant* is assigned to an integral value *k*. If *host-variable* is specified, it must be an exact numeric type with scale zero, must not include an indicator variable, and *k* must be in the range of 1 to 32767.

The cursor must be positioned on a rowset, and the specified value must be a valid value for the set of rows most recently retrieved for the cursor. If the specified row cannot be updated, an error is returned. It is possible that the specified row is within the bounds of the rowset most recently requested, but the current rowset contains less than the number of rows that were implicitly or explicitly requested when that rowset was established.

If this clause is not specified, the cursor position determines the rows that will be affected. If the cursor is positioned on a single row, that row is the one updated. In the case where the most recent FETCH statement returned multiple rows of data (but not as a rowset), this position would be on the last row of data that was returned. If the cursor is positioned on a rowset, all rows corresponding to the current rowset are updated. The cursor position remains unchanged.

It is possible for another application process to update a row in the base table of the SELECT statement so that the specified row of the cursor no longer has a corresponding row in the base table. An attempt to update such a row results in an error.

isolation-clause

Specifies the isolation level used when locating the rows to be updated by the statement.

WITH

Introduces the isolation level, which may be one of the following:

- RR** Repeatable read
- RS** Read stability
- CS** Cursor stability

The default isolation level of the statement is the isolation level of the package or plan in which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

SKIP LOCKED DATA

Specifies that rows are skipped when incompatible locks are held on the row by other transactions. These rows can belong to any accessed table that is specified in the statement. **SKIP LOCKED DATA** can be used only when isolation CS or RS is in effect and applies only to row level or page level locks.

SKIP LOCKED DATA can be specified only in the searched UPDATE statement (or the searched update operation of a MERGE statement). **SKIP LOCKED DATA** is ignored if it is specified when the isolation level that is in effect is repeatable read (**WITH RR**) or uncommitted read (**WITH UR**). The default isolation level of the statement depends on the isolation level of the package or plan with which the statement is bound, with the package isolation taking precedence over the plan isolation. When a package isolation is not specified, the plan isolation is the default.

QUERYNO *integer*

Specifies the number to be used for this SQL statement in EXPLAIN output and trace records. The number is used for the QUERYNO column of the plan table for the rows that contain information about this SQL statement. This number is also used in the QUERYNO column of the SYSIBM.SYSSTMT and SYSIBM.SYSPACKSTMT catalog tables.

If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation. Thus, if the application program is changed and then precompiled, that statement number might change.

Using the **QUERYNO** clause to assign unique numbers to the SQL statements in a program is helpful:

- For simplifying the use of optimization hints for access path selection
- For correlating SQL statement text with EXPLAIN output in the plan table

For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, and for information on accessing the plan table, see *DB2 Performance Monitoring and Tuning Guide*.

Notes

Update rules: Update values must satisfy the following rules. If they do not, or if other errors occur during the execution of the UPDATE statement, no rows are updated and the position of the cursors are not changed.

- *Assignment.* Update values are assigned to columns using the assignment rules described in Chapter 2, “Language elements,” on page 47.
- *Validity.* Updates must obey the following rules. If they do not, or if any other errors occur during the execution of the UPDATE statement, no rows are updated.
 - *Fullselects:* The row-fullselect and expressions that contain a *scalar-fullselect* must return no more than one row.
 - *Unique constraints and unique indexes:* If the identified table (or base table of the identified view) has any unique indexes or unique constraints, each row that is updated in the table must conform to the limitations that are imposed by those indexes and constraints.

All uniqueness checks are effectively made at the end of the statement. In the case of a multi-row update, this validation occurs after all the rows are updated.

- *Check constraints:* If the identified table (or base table of the identified view) has any check constraints, each check constraint must be true or unknown for each row that is updated in the table.

All checks constraints are effectively validated at the end of the statement. In the case of a multi-row update, this validation occurs after all the rows are updated.

- *Views and the WITH CHECK OPTION.* For views defined with WITH CHECK OPTION, an updated row must conform to the definition of the view. If the view you name is dependent on other views whose definitions include WITH CHECK OPTION, the updated rows must also conform to the definitions of those views. For an explanation of the rules governing this situation, see “CREATE VIEW” on page 1184.

For views that are not defined with WITH CHECK OPTION, you can change the rows so that they no longer conform to the definition of the view. Such rows are updated in the base table of the view and no longer appear in the view.

- *Field and validation procedures.* The updated rows must conform to any constraints imposed by any field or validation procedures on the identified table (or on the base table of the identified view).
- *Referential constraints.* The value of the parent key in a parent row must not be changed. If the update value produces a foreign key that is nonnull, the foreign key must be equal to some value of the parent key of the parent table of the relationship.

All referential constraints are effectively checked at the end of the statement. In the case of a multi-row update, this validation occurs after all the rows are updated.

- *Indexes with VARBINARY columns.* If the identified table has an index on a VARBINARY column or a column that is a distinct type that is based on VARBINARY data type, that index column cannot specify the DESC attribute. To use the SQL data change operation on the identified table, either drop the index or alter the data type of the column to BINARY and then rebuild the index.
- *Triggers.* An UPDATE statement might cause triggers to activate. A trigger might cause other statements to be executed or raise error conditions based on the update values. If an UPDATE statement for a view causes an instead of trigger to activate, validity, referential integrity, and check constraints are checked against the data changes that are performed in the trigger and not against the view that causes the trigger to activate or its underlying base tables.

Number of rows updated: Normally, after an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows updated. (For a complete description of the SQLCA, including exceptions to the preceding sentence, see “SQL communication area (SQLCA)” on page 1646.)

Nesting user-defined functions or stored procedures: An UPDATE statement can implicitly or explicitly refer to user-defined functions or stored procedures. This is known as *nesting* of SQL statements. A user-defined function or stored procedure that is nested within the UPDATE must not access the table being updated.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful update operation. Until a commit or rollback operation releases the locks, only the application process that performed the insert can access the updated row. If LOBs are not updated, application processes that are running with uncommitted read can also access the updated row. The locks can also prevent other application processes from performing operations on the table. However, application processes that are running with uncommitted read can access locked pages and rows.

Locks are not acquired on declared temporary tables.

Datetime representation when using datetime registers: As explained under Datetime special registers, when two or more datetime registers are implicitly or explicitly specified in a single SQL statement, they represent the same point in time. This is also true when multiple rows are updated.

Rules for positioned UPDATE with a SENSITIVE STATIC scrollable cursor: When a SENSITIVE STATIC scrollable cursor has been declared, the following rules apply:

- *Update attempt of delete holes.* If, with a positioned update against a SENSITIVE STATIC scrollable cursor, an attempt is made to update a row that has been identified as a delete hole, an error occurs.
- *Update operations.* Positioned update operations with SENSITIVE STATIC scrollable cursors perform as follows:
 1. The SELECT list items in the target row of the base table of the cursor are compared with the values in the corresponding row of the result table (that is, the result table must still agree with the base table). If the values are not identical, then the update operation is rejected, and an error occurs. The operation may be attempted again after a successful FETCH SENSITIVE has occurred for the target row.
 2. The WHERE clause of the SELECT statement is re-evaluated to determine whether the current values in the base table still satisfy the search criteria. The values in the SELECT list are compared to determine that these values have not changed. If the WHERE clause evaluates as true, and the values in the SELECT have not changed, the update operation is allowed to proceed. Otherwise, the update operation is rejected, an error occurs, and an *update hole* appears in the cursor.
- *Update of update holes.* Update holes are not permanent. It is possible for another process, or a searched update in the same process, to update an update hole row so that it is no longer an update hole. Update holes become visible with a FETCH SENSITIVE for positioned updates and positioned deletes.
- *Result table.* After the base table is updated, the row is re-evaluated and updated in the temporary result table. At this time, it is possible that the positioned update changed the data such that the row does not qualify the search condition, in which case the row is marked as an update hole for subsequent FETCH operations.

Updating rows in a table with multilevel security: When you update rows in a table with multilevel security, DB2 compares the security label of the user (the primary authorization ID) to the security label of the row. The update proceeds according to the following rules:

- If the security label of the user and the security label of the row are equivalent, the row is updated and the value of the security label is determined by whether the user has write-down privilege:
 - If the user has write-down privilege or write-down control is not enabled, the user can set the security label of the row to any valid security label. The value that is specified for the security label column must be assignable to a column that is defined as CHAR(8) FOR SBCS DATA NOT NULL.
 - If the user does not have write-down privilege and write-down control is enabled, the security label of the row is set to the value of the security label of the user.
- If the security label of the user dominates the security label of the row, the result of the UPDATE statement is determined by whether the user has write-down privilege:
 - If the user has write-down privilege or write-down control is not enabled, the row is updated and the user can set the security label of the row to any valid security label.
 - If the user does not have write-down privilege and write-down control is enabled, the row is not updated.
- If the security label of the row dominates the security label of the user, the row is not updated.

Other SQL statements in the same unit of work: The following statements cannot follow an UPDATE statement in the same unit of work:

- An ALTER TABLE statement that changes the data type of a column (ALTER COLUMN SET DATA TYPE)
- An ALTER INDEX statement that changes the padding attribute of an index with varying-length columns (PADDED to NOT PADDED or vice versa)

Examples

Example 1: Change employee 000190's telephone number to 3565 in DSN8910.EMP.

```
UPDATE DSN8910.EMP
SET PHONENO='3565'
WHERE EMPNO='000190';
```

Example 2: Give each member of department D11 a 100-dollar raise.

```
UPDATE DSN8910.EMP
SET SALARY = SALARY + 100
WHERE WORKDEPT = 'D11';
```

Example 3: Employee 000250 is going on a leave of absence. Set the employee's pay values (SALARY, BONUS, and COMMISSION) to null.

```
UPDATE DSN8910.EMP
SET SALARY = NULL, BONUS = NULL, COMM = NULL
WHERE EMPNO='000250';
```

Alternatively, the statement could also be written as follows:

```
UPDATE DSN8910.EMP
SET (SALARY, BONUS, COMM) = (NULL, NULL, NULL)
WHERE EMPNO='000250';
```

Example 4: Assume that a column named PROJSIZE has been added to DSN8910.EMP. The column records the number of projects for which the employee's department has responsibility. For each employee in department E21, update PROJSIZE with the number of projects for which the department is responsible.

```
UPDATE DSN8910.EMP
SET PROJSIZE = (SELECT COUNT(*)
                FROM DSN8910.PROJ
                WHERE DEPTNO = 'E21')
WHERE WORKDEPT = 'E21';
```

Example 5: Double the salary of the employee represented by the row on which the cursor C1 is positioned.

```
EXEC SQL UPDATE DSN8910.EMP
SET SALARY = 2 * SALARY
WHERE CURRENT OF C1;
```

Example 6: Assume that employee table EMP1 was created with the following statement:

```
CREATE TABLE EMP1
(EMP_ROWID ROWID GENERATED ALWAYS,
 EMPNO     CHAR(6),
 NAME      CHAR(30),
 SALARY    DECIMAL(9,2),
 PICTURE   BLOB(250K),
 RESUME    CLOB(32K));
```

Assume that host variable *HV_EMP_ROWID* contains the value of the ROWID column for employee with employee number '350000'. Using that ROWID value to identify the employee and user-defined function *UPDATE_RESUME*, increase the employee's salary by \$1000 and update that employee's resume.

```
EXEC SQL UPDATE EMP1
  SET SALARY = SALARY + 1000,
      RESUME = UPDATE_RESUME(:HV_RESUME)
  WHERE EMP_ROWID = :HV_EMP_ROWID;
```

Example 7: In employee table X, give each employee whose salary is below average a salary increase of 10%.

```
EXEC SQL UPDATE EMP X
  SET SALARY = 1.10 * SALARY
  WHERE SALARY < (SELECT AVG(SALARY) FROM EMP Y
  WHERE X.JOBCODE = Y.JOBCODE);
```

Example 8: Raise the salary of the employees in department 'E11' whose salary is below average to the average salary.

```
EXEC SQL UPDATE EMP T1
  SET SALARY = (SELECT AVG(T2.SALARY) FROM EMP T2)
  WHERE WORKDEPT = 'E11' AND
      SALARY < (SELECT AVG(T3.SALARY) FROM EMP T3);
```

Example 9: Give the employees in department 'E11' a bonus equal to 10% of their salary.

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT BONUS
    FROM DSN8710.EMP
    WHERE WORKDEPT = 'E12'
    FOR UPDATE OF BONUS;
EXEC SQL
  UPDATE DSN8710.EMP
    SET BONUS = ( SELECT .10 * SALARY FROM DSN8710.EMP Y
                  WHERE EMPNO = Y.EMPNO )
  WHERE CURRENT OF C1;
```

Example 10: Assuming that cursor CS1 is positioned on a rowset consisting of 10 rows in table T1, update all 10 rows in the rowset.

```
EXEC SQL UPDATE T1 SET C1 = 5 WHERE CURRENT OF CS1;
```

Example 11: Assuming that cursor CS1 is positioned on a rowset consisting of 10 rows in table T1, update the fourth row of the rowset.

```
short ind1, ind2;

int n, updt_value;

stmt = 'UPDATE T1 SET C1 = ? WHERE CURRENT OF CS1 FOR ROW ? OF ROWSET'

ind1 = 0;

ind2 = 0;

n = 4;

updt_value = 5;

...

strcpy(my_sqlda.sqldaid,"SQLDA");
```



```

my_sqlda.sqln = 2;

my_sqlda.sqld = 2;

my_sqlda.sqlvar[0].sqltype = 497;
my_sqlda.sqlvar[0].sqllen = 4;
my_sqlda.sqlvar[0].sqldata = (int *) &updt_value;
my_sqlda.sqlvar[0].sqlind = (short *) &ind1;

my_sqlda.sqlvar[1].sqltype = 497;
my_sqlda.sqlvar[1].sqllen = 4;
my_sqlda.sqlvar[1].sqldata = (int *) &n;
my_sqlda.sqlvar[1].sqlind = (short *) &ind2;

EXEC SQL PREPARE S1 FROM :stmt;

EXEC SQL EXECUTE S1 USING DESCRIPTOR :my_sqlda;

```

VALUES

The VALUES statement provides a method for invoking a user-defined function from a trigger. Transition variables and transition tables can be passed to the user-defined function.

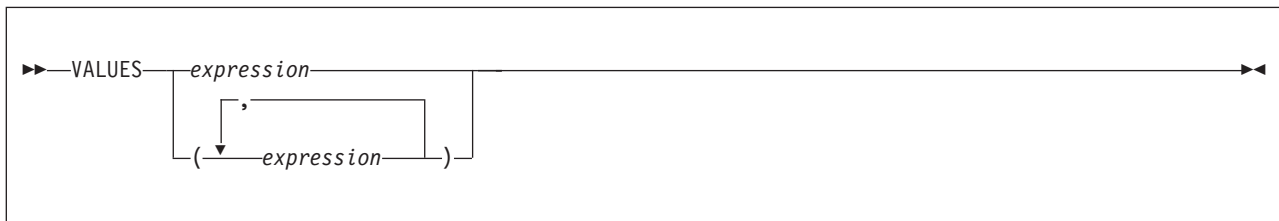
Invocation

This statement can only be used in the triggered action of a trigger.

Authorization

Authorization is required for any expressions that are used in the statement. For more information, see “Expressions” on page 180.

Syntax



Description

VALUES

Specifies one or more expressions. If more than one expression is specified, the expressions must be enclosed within parentheses.

expression

Any expression of the type described in “Expressions” on page 180. The expression must not contain a host variable.

The expressions are evaluated, but the resulting values are discarded and are not assigned to any output variables.

If a user-defined function is specified as part of an expression, the user-defined function is invoked. If a negative SQLCODE is returned when the function is invoked, DB2 stops executing the trigger and rolls back any triggered actions that were performed.

Example

Example: Create an after trigger EMPISRT1 that invokes user-defined function NEWEMP when the trigger is activated. An insert operation on table EMP activates the trigger. Pass transition variables for the new employee number, last name, and first name to the user-defined function.

```
CREATE TRIGGER EMPISRT1
  AFTER INSERT ON EMP
  REFERENCING NEW AS N
  FOR EACH ROW
  MODE DB2SQL
  BEGIN ATOMIC
    VALUES(NEWEMP(N.EMPNO, N.LASTNAME, N.FIRSTNAME));
  END
```

VALUES INTO

The VALUES INTO statement assigns one or more values to host variables.

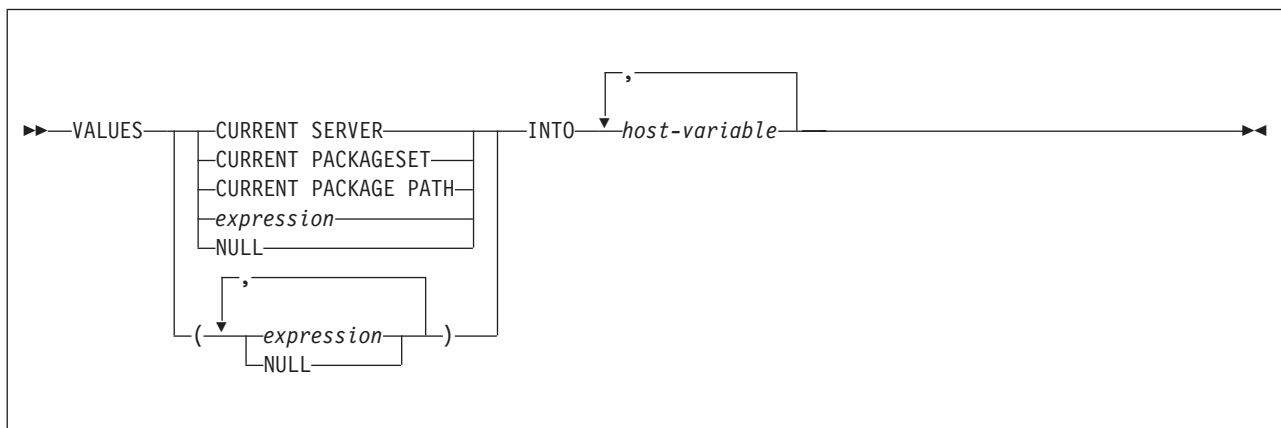
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

Authorization is required for any expressions that are used in the statement. For more information, see “Expressions” on page 180.

Syntax



Description

VALUES

Introduces one or more values. If more than one value is specified, the list of values must be enclosed within parentheses.

expression

The expression is any expression of the type described in “Expressions” on page 180 except it cannot contain a reference to local special registers (CURRENT SERVER, CURRENT PACKAGESET, or CURRENT PACKAGE PATH). The expression must not include a column name.

NULL

The null value. **NULL** can only be specified for host variables that have an associated indicator variable.

INTO

Introduces one or more host variables. The values that are specified in the **VALUES** clause are assigned to these host variables. The first value specified is assigned to the first host variable, the second value to the second host variable, and so on. Each assignment is made according to the rules described in “Assignment and comparison” on page 102. Assignments are made in sequence through the list. If there are fewer host variables than values, the value 'W' is assigned to the SQLWARN3 field of the SQLCA. (See “SQL communication area (SQLCA)” on page 1646.)

host-variable

Identifies a variable that is described in the program according to the rules for declaring host variables.

Notes

The default encoding scheme for the data is the value in the bind option ENCODING, which is the option for application encoding. If this statement is used with functions such as LENGTH or SUBSTRING that are operating on LOB locators, and the LOB data that is specified by the locator is in a different encoding scheme from the ENCODING bind option, LOB materialization and character conversion occur. To avoid LOB materialization and character conversion, select the LOB data from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table.

If an error occurs, no value is assigned to any host variable. However, if LOB values are involved, there is a possibility that the corresponding host variable was modified, but the variable's contents are unpredictable.

Normally, you use LOB locators to assign and retrieve data from LOB columns. However, because of compatibility rules, you can also use LOB locators to assign data to host variables with other data types. For more information on using locators, see *DB2 Application Programming and SQL Guide*.

Examples

Example 1: Assign the value of the CURRENT PATH special register to host variable *HV1*.

```
EXEC SQL VALUES(CURRENT PATH)
      INTO :HV1;
```

Example 2: Assign the value of the CURRENT MEMBER special register to host variable *MEM*.

```
EXEC SQL VALUES(CURRENT MEMBER)
      INTO :MEM;
```

Example 3: Assume that LOB locator *LOB1* is associated with a CLOB value. Assign a portion of the CLOB value to host variable *DETAILS* using the LOB locator.

```
EXEC SQL VALUES (SUBSTR(:LOB1,1,35))
      INTO :DETAILS;
```

If the LOB data that is specified by the LOB locator *LOB1* is in a different encoding scheme from the value of the ENCODING bind option, and you want to avoid LOB materialization and character conversion, use the following statement instead of the VALUES INTO statement:

```
EXEC SQL SELECT SUBSTR(:LOB1,1,35)
      INTO :DETAILS
      FROM SYSIBM.SYSDUMMYU;
```

WHENEVER

The **WHENEVER** statement specifies the host language statement to be executed when a specified exception condition occurs.

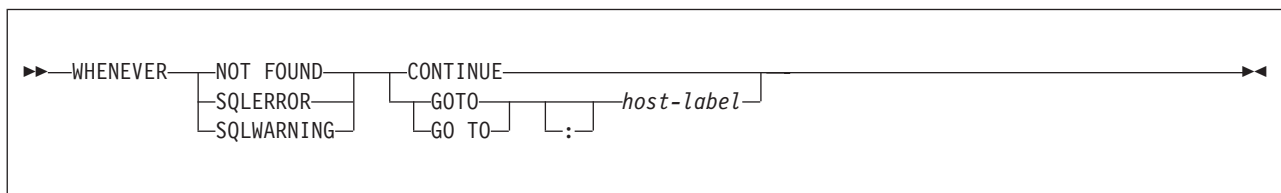
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

The **NOT FOUND**, **SQLERROR**, or **SQLWARNING** clause is used to identify the type of exception condition.

NOT FOUND

Identifies any condition that results in an SQLCODE of +100 (equivalently, an SQLSTATE code of '02000').

SQLERROR

Identifies any condition that results in a negative SQLCODE.

SQLWARNING

Identifies any condition that results in a warning condition (SQLWARN0 is W), or that results in a positive SQLCODE other than +100.

The **CONTINUE** or **GO TO** clause specifies the next statement to be executed when the identified type of exception condition exists.

CONTINUE

Specifies the next sequential statement of the source program.

GOTO or GO TO *host-label*

Specifies the statement identified by *host-label*. For *host-label*, substitute a single token, optionally preceded by a colon. The form of the token depends on the host language. In COBOL, for example, it can be *section-name* or an unqualified *paragraph-name*.

Notes

There are three types of **WHENEVER** statements:

- **WHENEVER NOT FOUND**
- **WHENEVER SQLERROR**
- **WHENEVER SQLWARNING**

Every executable SQL statement in an application program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a WHENEVER statement is related to the listing sequence of the statements in the application program, not their execution sequence.

An SQL statement is within the scope of the last WHENEVER statement of each type that is specified before that SQL statement in the source program. If a WHENEVER statement of some type is not specified before an SQL statement, that SQL statement is within the scope of an implicit WHENEVER statement of that type in which **CONTINUE** is specified. If a WHENEVER statement is specified in a Fortran subprogram, its scope is that subprogram, not the source program.

The GET DIAGNOSTICS statement can be used to provide additional information.

Examples

The following statements can be embedded in a COBOL program.

Example 1: Go to the label HANDLER for any statement that produces an error.

```
EXEC SQL WHENEVER SQLERROR GOTO HANDLER END-EXEC.
```

Example 2: Continue processing for any statement that produces a warning.

```
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
```

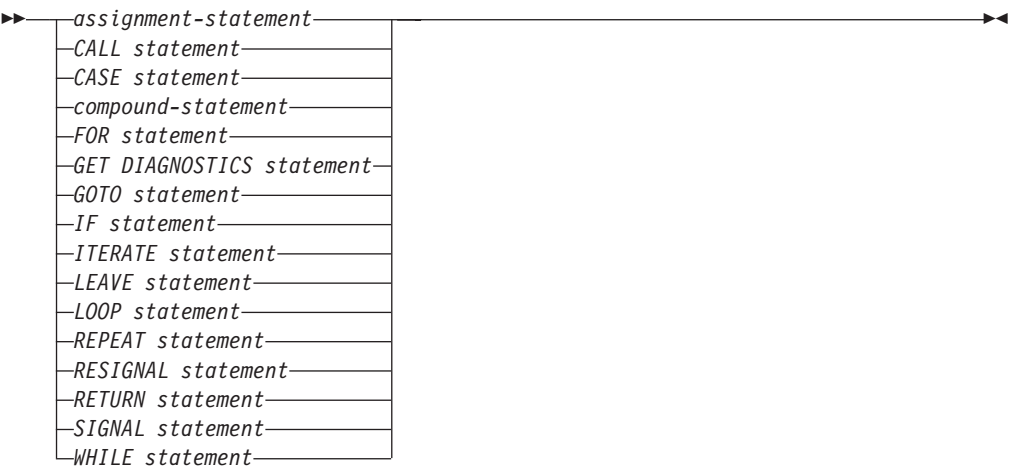
Example 3: Go to the label ENDDATA for any statement that does not return.

```
EXEC SQL WHENEVER NOT FOUND GO TO ENDDATA END-EXEC.
```

Chapter 6. SQL control statements for native SQL procedures

SQL control statements for SQL routines can be used in SQL functions and native SQL procedures. *SQL control statements* provide the capability to control the logic flow, declare and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements.

SQL-control-statement:



Control statements are supported in SQL procedures. SQL procedures are created by specifying an SQL routine body on the “CREATE PROCEDURE (SQL - native)” on page 1046 statement. The SQL routine body must be a single SQL statement which might be an SQL control statement.

The remainder of this section contains a description of the control statements that are supported by native SQL procedures, and includes syntax diagrams, semantic descriptions, usage notes, and examples of the use of the statements that constitute the SQL routine body. In addition, you can find information about referencing SQL parameters and variables in “References to SQL parameters and SQL variables” on page 1542.

Refer to “SQL control statements for external SQL procedures” on page 1610 for information about the control statements that are supported by external SQL procedures.

The two common elements that are used in describing specific SQL control statements are:

- SQL control statements as described above
- “SQL-procedure-statement” on page 1546

References to SQL parameters and SQL variables

SQL parameters and SQL variables can be referenced anywhere in the statement where an expression or a variable can be specified. Host variables cannot be specified in SQL routines. SQL parameters can be referenced anywhere in the routine and can be qualified with the routine name. SQL variables can be referenced anywhere in the compound statement in which they are declared, including any statement that is directly or indirectly nested within that compound statement.

If the compound statement where the variable is declared has a label, references to the variable name can be qualified with that label.

All SQL parameters and SQL variables are considered nullable. The name of an SQL parameter or an SQL variable in an SQL routine can be the same as the name of a column in a table or view that is referenced in the routine. The name of an SQL variable can also be the same as the name of another SQL variable that is declared in the same routine. This can occur when the two SQL variables are declared in different *compound-statements*. The *compound-statement* that contains the declaration of an SQL variable determines the scope of that variable. See “compound-statement” on page 1554 for additional information.

Names that are the same should be explicitly qualified. Qualifying a name clearly indicates whether the name refers to a column, an SQL variable, or an SQL parameter. If the name is not qualified or is qualified but is still ambiguous, the following rules describe whether the name refers to a column or to an SQL variable or an SQL parameter in an SQL routine:

- The name is checked to see if it is the name of a column of any existing table or a view that is specified in the SQL routine body at the current server. If the name is found as a column name, but the privilege set that is used to issue the CREATE PROCEDURE or ALTER PROCEDURE statement does not have the proper authority to access the table or view, the VALIDATE option that is in effect for the procedure determines what happens:
 - If VALIDATE BIND is in effect, an error is returned.
 - If VALIDATE RUN is in effect, the name is assumed to be a column name. If the privilege set that is used to issue the CREATE statement does not have the proper authority to access the table or view at run time, an error is returned.
- If the referenced tables or views do not exist at the current server, the name will be checked first as an SQL variable name in the compound statement and then as an SQL parameter name. The variable can be declared within the *compound-statement* that contains the reference, or within a compound statement in which that compound statement is nested. If two SQL variables are within the same scope and have the same name, the SQL variable that is declared in the innermost compound statement is used.

If the name is not found as an SQL parameter or SQL variable, the VALIDATE option that is in effect for the procedure determines what happens:

- If VALIDATE BIND is in effect, an error is returned.
- If VALIDATE RUN is in effect, the name is assumed to be a column name. If a column does not exist with that name at run time, an error is returned.

The name of an SQL variable or an SQL parameter in an SQL routine can be the name of an identifier that is used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to an identifier, an SQL variable, or an SQL parameter:

- In the SET CURRENT PACKAGE PATH, SET PATH and SET SCHEMA statements, the name is checked as an SQL variable name or an SQL parameter name. If an SQL variable or SQL parameter with that name is not found, the name is assumed to be an identifier.
- In the ASSOCIATE LOCATORS, CONNECT, RELEASE (connection), and SET CONNECTION statements, the name is used as an identifier.

References to SQL condition names

A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a condition name, the condition that is declared in the innermost compound statement is the condition that is used.

The name of an SQL condition can be the same as the name of another SQL condition that is declared in the same routine. This can occur when the two SQL conditions are declared in different *compound-statements*. The *compound-statement* that contains the declaration of an SQL condition name determines the scope of that condition name. A condition name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement. A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a condition name, the condition that is declared in the innermost compound statement is the condition that is used. See “compound-statement” on page 1554 for additional information.

References to SQL cursor names

A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement.

The name of an SQL cursor can not be the same as the name of another SQL cursor that is declared on the same routine. The *compound-statement* that contains the declaration of an SQL cursor name determines the scope of that cursor name. A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. See “compound-statement” on page 1554 for additional information.

References to labels

Specifying a label for an SQL procedure statement defines that label and determines the scope of that label. A label name can only be referenced within the compound statement in which it is defined, including a reference from any statement that is directly or indirectly nested within that compound statement. The FOR statement is considered the same as a compound statement with respect to defining and referencing labels. A label can be specified as the target of a GOTO, LEAVE, or ITERATE statement, subject to the rules for the statement that references the label as a target.

Labels can be specified on most SQL procedure statements. If a label is specified on an SQL procedure statement, it must be unique from other labels within the same scope. A label must not be the same as any other label within the same compound statement, must not be the same as a label specified on the compound statement itself, and if the compound statement is nested within another compound statement, the label must not be the same as the label specified on any higher level compound statement. The label must not be the same as the name of the SQL procedure.

Nested compound statements and scope of names

Nested compound statements can be used within an SQL routine to define the scope of SQL variable declarations, cursors, condition names, and condition handlers.

In addition, labels have a defined scope in the context of nested compound statements. However, the rules for name spaces and how non-unique names can be referenced, differs depending on the type of name. The following table summarizes these differences:

Table 130. Scope and qualification of names within nested compound statements

Type of name	Name can be qualified	Name must be unique within	Name can be referenced within
SQL variable	Yes. The name can be qualified with the label of the compound statement in which the variable is declared	the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement	<p>The compound statement in which it is declared, including any compound statements that are nested within that compound statement.</p> <p>When multiple SQL variables are defined with the same name, a label can be used to explicitly refer to a specific variable that is not the most local in scope</p>
condition name	No	the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement	<p>The compound statement in which it is declared, including any compound statements that are nested within that compound statement.</p> <p>Condition names can be used in the declaration of a condition handler, or in a SIGNAL or RESIGNAL statement.</p> <p>If multiple conditions are defined with the same name, there is no way to explicitly refer to the condition that is not the most local in scope.</p>

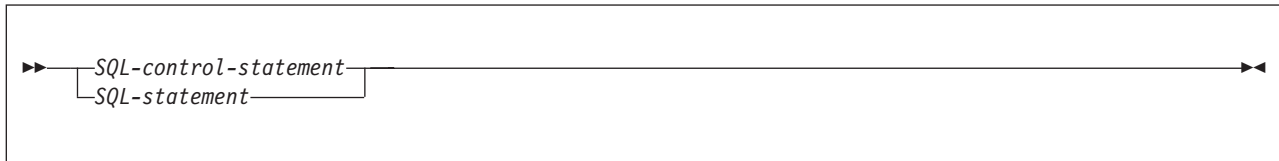
Table 130. Scope and qualification of names within nested compound statements (continued)

Type of name	Name can be qualified	Name must be unique within	Name can be referenced within
cursor name	No	the routine	<p>The compound statement in which it is declared, including any compound statements that are nested within that compound statement.</p> <p>If the cursor is defined as a result set cursor, the invoking application can access the result set.</p>
label	No	the compound statement that defined the label, including any definitions in compound statements that are nested within that compound statement	<p>The compound statement in which it is defined, including any compound statements that are nested within that compound statement.</p> <p>Use a label to qualify the name of an SQL variable or as the target of a GOTO, LEAVE, or ITERATE statement, subject to the rules for these statements.</p>

SQL-procedure-statement

An SQL control statement can allow multiple SQL statements to be specified within the SQL control statement. These statements are defined as SQL procedure statements.

Syntax



Description

SQL-control-statement

Specifies an SQL statement that provides the capability to control logic flow, declare and set variables, and handle warnings and exceptions, as defined in this section. Control statements are supported in SQL routines.

SQL-statement

Specifies an SQL statement as listed in Table 139 on page 1608. These statements are described in Chapter 5, “Statements,” on page 683.

Notes

Comments: Comments can be included within the body of an SQL routine. In addition to the double-dash form of comments (--), a comment can begin with /* and end with */. The following rules apply to this form of comment:

- The beginning characters /* must be adjacent and on the same line.
- The ending characters */ must be adjacent and on the same line.
- Comments can be started wherever a space is valid.
- Comments can be continued to the next line.

Detecting and processing error and warning conditions: As an SQL statement is executed, DB2 stores information about the processing of the statement in a diagnostics area (including the SQLSTATE and SQLCODE), unless otherwise noted in the description of the SQL statement. A completion condition can indicate that the SQL statement completed successfully, completed with a warning condition, or completed with a not found condition. An exception condition indicates that the SQL statement was not successful.

A condition handler can be defined to execute when an exception condition, a warning condition, or a not found condition occurs in a compound statement. The declaration of a condition handler includes the code that is executed when the condition handler is activated. When a condition other than a successful completion occurs in the processing of *SQL-procedure-statement* and a condition handler that can handle the condition is within scope, one such condition handler will be activated to process the condition. See “compound-statement” on page 1554 for information about defining condition handlers. The code in the condition handler can check for a warning condition, a not found condition, or an exception condition and can take the appropriate action. Use one of the following methods at the beginning of the body of a condition handler to check the condition in the diagnostics area that caused the handler to be activated.

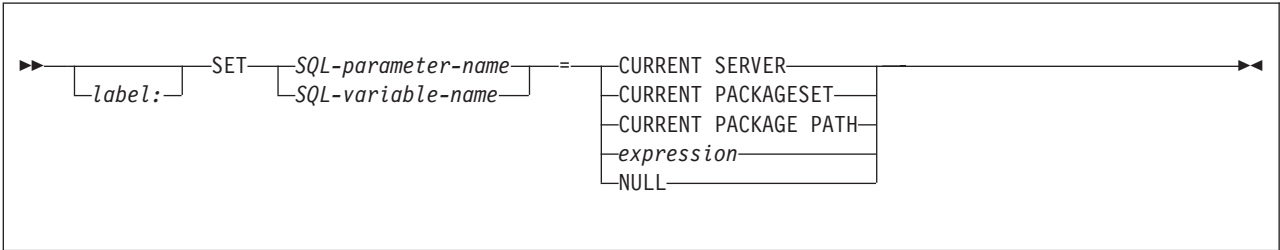
- Issue a GET DIAGNOSTICS statement to request the desired information from the diagnostics area. See “GET DIAGNOSTICS” on page 1317.
- Test the SQLSTATE and SQLCODE SQL variables.

If the condition is a warning and no handler exists for the condition, the previous two methods can be used outside of the body of a condition handler if they are used immediately following the statement for which the condition is desired. If the condition is an error and no handler exists for the condition, the routine terminates with the error condition.

assignment-statement

The assignment statement assigns a value to a SQL parameter or to an SQL variable.

Syntax



Description

label

Specifies the label for *assignment-statement*. The label name cannot be the same as the name of the SQL routine, or another label within the same scope. For additional information, see “References to labels” on page 1543.

SQL-parameter-name

Identifies the parameter that is the assignment target. The parameter must be specified in *parameter-declaration* in the CREATE statement.

SQL-variable-name

Identifies the SQL variable that is the assignment target. SQL variables can only be declared in a *compound-statement* and must be declared before they are used. For information on declaring SQL variables, see “compound-statement” on page 1554.

expression or NULL

Specifies the expression or value that is the source for the assignment. The expression can be any expression of the type described in “Expressions” on page 180 except it cannot contain a reference to local special registers (CURRENT SERVER, CURRENT PACKAGESET, or CURRENT PACKAGE PATH).

Notes

Assignment rules: Assignment statements in SQL routines must conform to the SQL assignment rules. For example, the data type of the target and source must be compatible. See “Assignment and comparison” on page 102 for assignment rules.

When a string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte or double-byte blanks. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the value is truncated and a warning is returned.

If truncation of the whole part of a number occurs on assignment to a numeric variable, the value is truncated and a warning is returned.

Assignments involving SQL parameters for SQL procedures: An IN parameter can appear on the left or right side in an assignment statement. When control returns

| to the caller, the original value of the IN parameter is retained. An OUT parameter
| can also appear on the left or right side in an assignment statement. If used
| without first being assigned a value, the value is undefined. When control returns
| to the caller, the last value that is assigned to an OUT parameter is returned to the
| caller. For an INOUT parameter, the first value of the parameter is determined by
| the caller, and the last value that is assigned to the parameter is returned to the
| caller.

| **Considerations for SQLSTATE and SQLCODE SQL variables:** Assignment to these
| variables is not prohibited. However, it is not recommended as assignment does
| not affect the diagnostic area or result in the activation of condition handlers.
| Furthermore, processing an assignment to these SQL variables causes the specified
| values for the assignment to be overlayed with the SQL return codes returned from
| executing the statement that does the assignment.

Examples

Example 1: Increase the SQL variable *p_salary* by 10 percent.

```
SET p_salary = p_salary + (p_salary * .10)
```

Example 2: Set SQL variable *p_salary* to the null value.

```
SET p_salary = NULL
```

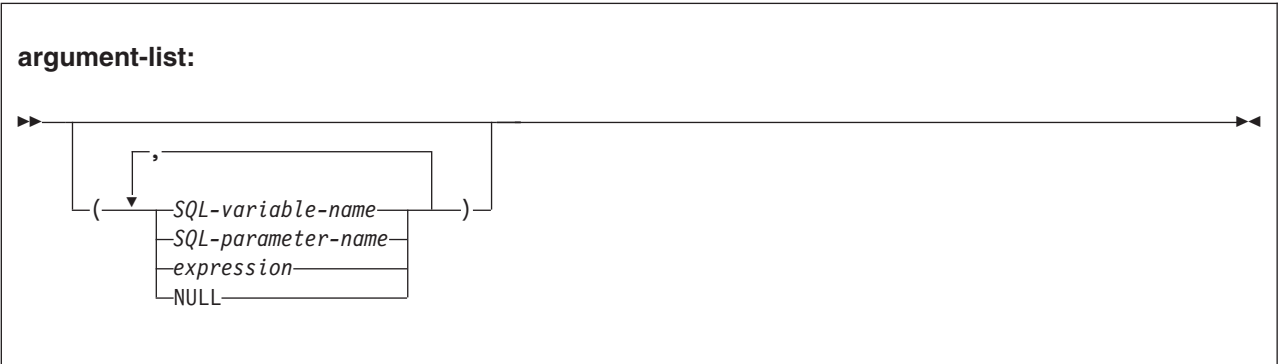
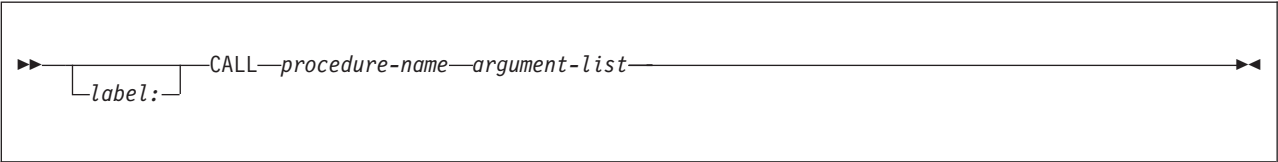
Example 3: Set SQL variable *midinit* to the first character of SQL variable *midname*.

```
SET midinit = SUBSTR(midname,1,1)
```

CALL statement

The CALL statement invokes a stored procedure.

Syntax



Description

label
Specifies the label for the CALL statement. The label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see “References to labels” on page 1543.

procedure-name
Identifies the stored procedure to call. The procedure name must identify a stored procedure that exists at the current server.

argument-list
Identifies a list of values to be passed as parameters to the stored procedure. The *n*th value corresponds to the *n*th parameter in the procedure. The number of parameters must be the same as the number of parameters defined for the stored procedure. See “CALL” on page 874 for more information.

Control is passed to the stored procedure according to the calling conventions for SQL routines. When execution of the stored procedure is complete, the value of each parameter of the stored procedure is assigned to the corresponding parameter of the CALL statement defined as OUT or INOUT.

SQL-variable-name
Specifies an SQL variable as an argument to the stored procedure. For an explanation of references to SQL variables, see “References to SQL parameters and SQL variables” on page 1542.

SQL-parameter-name
Specifies an SQL parameter as an argument to the stored procedure. For an explanation of references to SQL parameters, see “References to SQL parameters and SQL variables” on page 1542.

expression
The parameter is the result of the specified *expression*, which is evaluated

before the stored procedure is invoked. If *expression* is a single *SQL-parameter-name* or *SQL-variable-name*, the corresponding parameter of the procedure can be defined as IN, INOUT, or OUT. Otherwise, the corresponding parameter of the procedure must be defined as IN. If the result of the *expression* can be the null value, either the description of the procedure must allow for null parameters or the corresponding parameter of the stored procedure must be defined as OUT.

The following additional rules apply depending on how the corresponding parameter was defined in the CREATE PROCEDURE statement for the procedure:

- IN *expression* can contain references to multiple SQL parameters or variables. In addition to the rules stated in “Expressions” on page 180 for *expression*, *expression* cannot include a column name, an aggregate function, or a user-defined function that is sourced on an aggregate function.
- INOUT or OUT *expression* can only be a single SQL parameter or variable.

NULL

The parameter is a null value. The corresponding parameter of the procedure must be defined as IN and the description of the procedure must allow for null parameters.

Notes

See “CALL” on page 874 for more information on the SQL CALL statement.

Examples

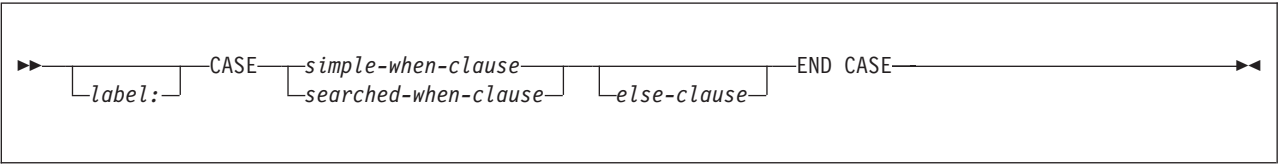
Call stored procedure proc1 and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

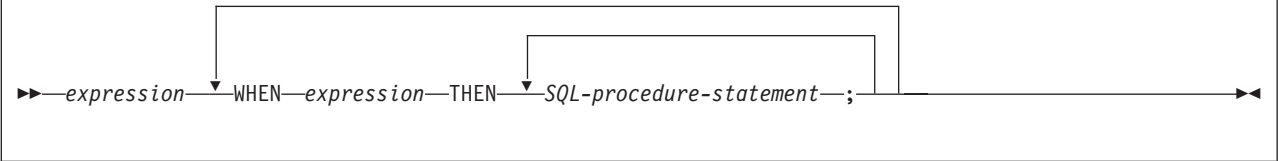
CASE statement

The CASE statement selects an execution path based on multiple conditions. A CASE statement operates in the same way as a CASE expression.

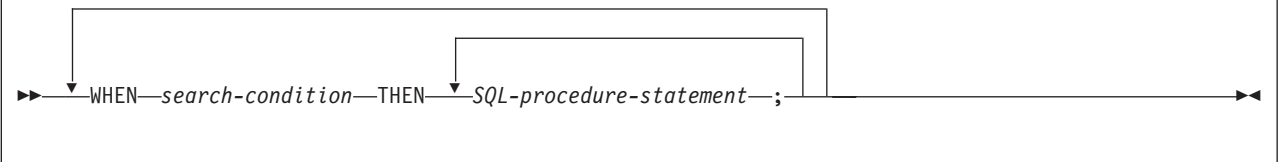
Syntax



simple-when-clause:



searched-when-clause:



else-clause:



Description

label
Specifies the label for the CASE statement. The label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see “References to labels” on page 1543.

CASE
Begins a *case-expression*.

simple-when-clause
The value of the *expression* prior to the first WHEN keyword is tested for equality with the value of the *expression* that follows each WHEN keyword. If the comparison is true, the statements in the associated THEN clause are

executed and processing of the CASE statement ends. If the result is unknown or false, processing continues to the next comparison. If the result does not match any of the comparisons, and an ELSE clause is present, the statements in the ELSE clause are executed.

searched-when-clause

The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are executed and processing of the CASE statement ends. If it evaluates to false, or unknown, the next *search-condition* is evaluated. If no *search-condition* evaluates to true and an ELSE clause is present, the statements in the ELSE clause are executed.

When *searched-when-clause* is used, *search-condition* cannot contain a *fullselect*.

SQL-procedure-statement

Specifies a statement to execute. See “SQL-procedure-statement” on page 1546.

search-condition

Specifies a condition that is true, false, or unknown about a row or group of table data.

ELSE *SQL-procedure-statement*

If none of the conditions specified in the *simple-when-clause* or *searched-when-clause* are true, the statements specified in *SQL-procedure-statement* are executed.

If none of the conditions specified in the WHEN clauses are true and an ELSE is not specified, an error is issued when the statement executes, and the execution of the CASE statement is terminated.

END CASE

Ends a *case-statement*.

Examples

Example 1: Use a simple case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

```
CASE v_workdept
  WHEN 'A00'
    THEN UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 1';
  WHEN 'B01'
    THEN UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 2';
  ELSE UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 3';
END CASE
```

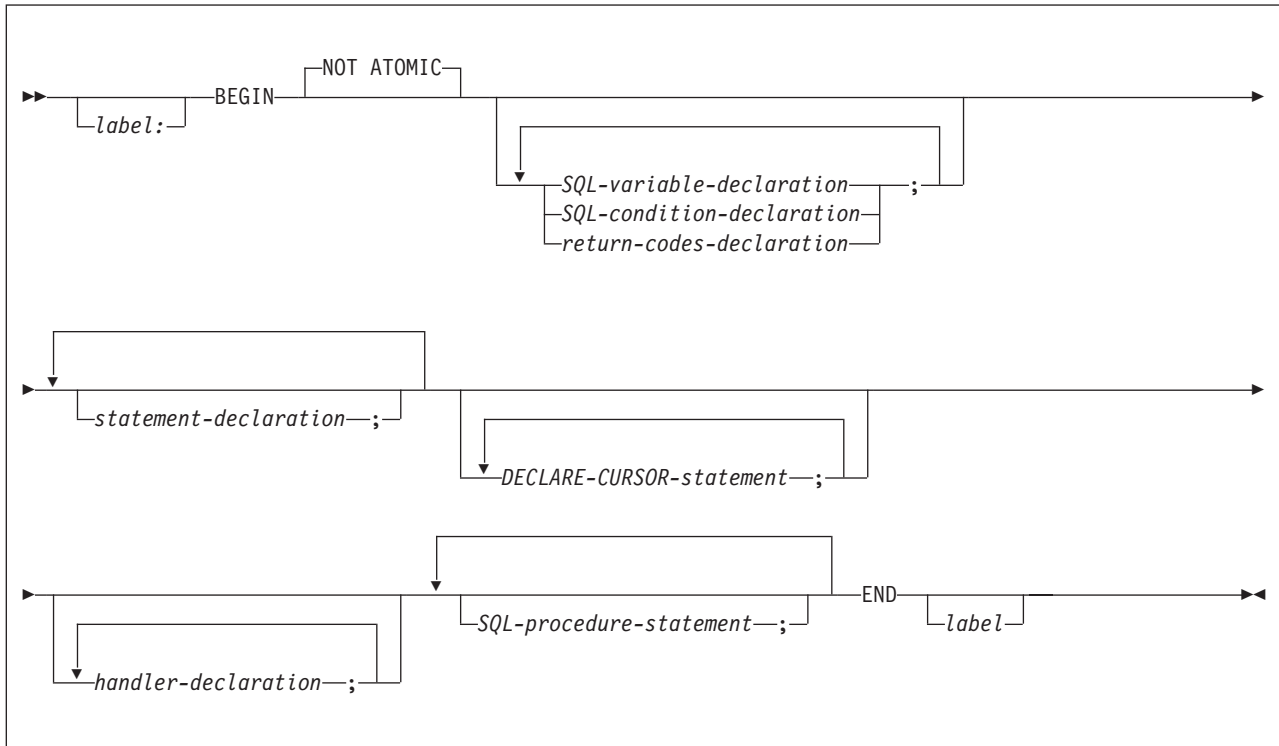
Example 2: Use a searched case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

```
CASE
  WHEN v_workdept < 'B01'
    THEN UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 1';
  WHEN v_workdept < 'C01'
    THEN UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 2';
  ELSE UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 3';
END CASE
```

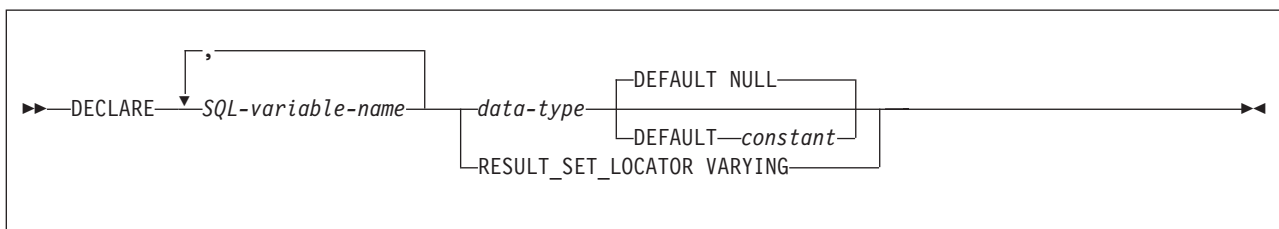
compound-statement

A compound statement groups other statements together in an SQL routine. A compound statement allows the declaration of SQL variables, cursors, and condition handlers.

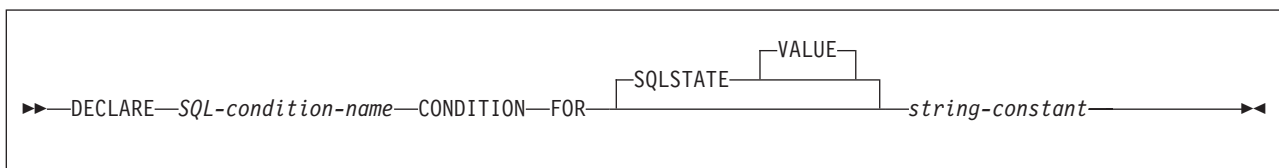
Syntax



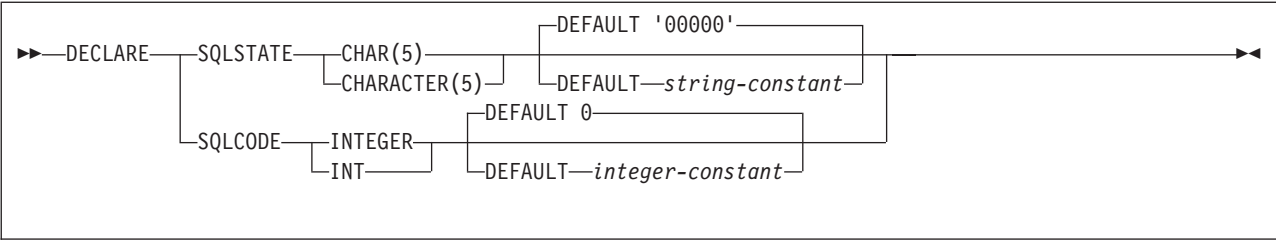
SQL-variable-declaration:



SQL-condition-declaration:



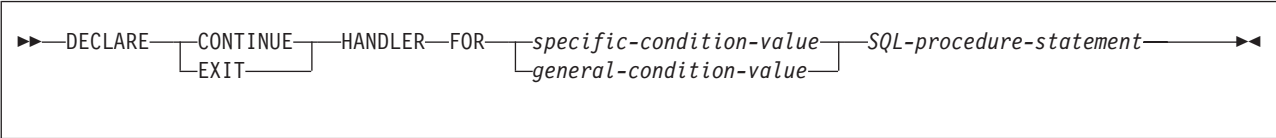
return-codes-declaration:



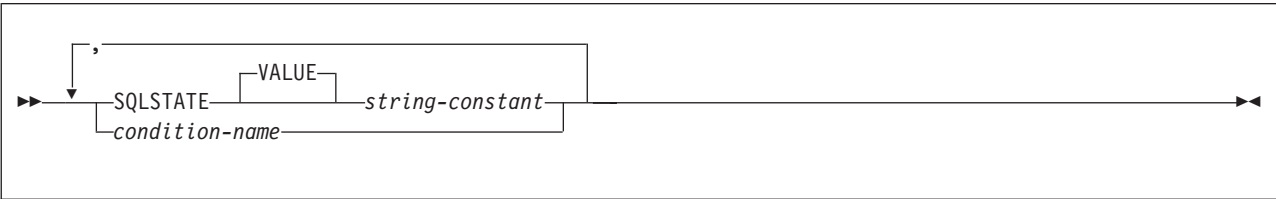
statement-declaration:



handler-declaration:



specific-condition-value:



general-condition-value:



Description

label

Specifies the label for the *compound-statement*. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound statement and can also be specified as the target on a LEAVE statement. If the ending label is specified, it must be the same as beginning label. The label name cannot be the same as the routine name or another label within the same scope.

NOT ATOMIC

NOT ATOMIC indicates that an unhandled exception condition within the *compound-statement* does not cause the *compound-statement* to be rolled back.

SQL-variable-declaration

Declares a variable that is local to the compound statement.

SQL-variable-name

Defines the name of a variable. DB2 converts all SBCS SQL variable names that are not delimited to uppercase. *SQL-variable-name* must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement. *SQL-variable-name* must not be the same as a parameter name. See “References to SQL parameters and SQL variables” on page 1542 for information about how SQL variable names are resolved when there are columns with the same name as an SQL variable involved in a statement, or when multiple SQL variables exist with the same name in the routine body.

SQL-variable-name can only be referenced within the compound statement in which it is declared, including any compound statement that is nested within that compound statement. If the compound statement where the variable is declared has a label, references to the variable name can be qualified with that label. For example, an SQL variable V that is declared in a compound statement that is labeled C can be referenced as C.V.

data-type

Specifies the data type and length of the variable. SQL variables follow the same rules for default lengths and maximum lengths as SQL routine parameters. *data-type* must not be XML. See “CREATE PROCEDURE (SQL - native)” on page 1046 for a description of SQL data types and lengths.

DEFAULT constant or NULL

Defines the default for the SQL variable. The specified constant must represent a value that could be assigned to the variable in accordance with the rules of assignment as described in “Assignment and comparison” on page 102. The variable is initialized when the compound statement begins processing. If a default value is not specified, the SQL variable is initialized to NULL.

RESULT_SET_LOCATOR VARYING

Specifies the data type for a result set locator variable.

SQL-condition-declaration

Declares a condition name and corresponding SQLSTATE value.

SQL-condition-name

Specifies the name of the condition. The condition name must be unique within the compound statement in which it is declared, excluding any declarations that are in compound statements that are nested within that compound statement. A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement.

FOR SQLSTATE string-constant

Specifies the SQLSTATE that is associated with the condition. The string must be specified as five characters enclosed in single quotes, and the SQLSTATE class (the first two characters) must not be '00'.

return-codes-declaration

Declares special variables named SQLSTATE and SQLCODE. These variables

are automatically set to the SQLSTATE and SQLCODE values for the first condition in the diagnostics area after executing an SQL statement other than GET DIAGNOSTICS or an empty compound statement.

The SQLSTATE and SQLCODE SQL variables are only intended to be used as a means of obtaining the SQL return codes that resulted from processing the previous SQL statement other than GET DIAGNOSTICS. If there is any intention to use the SQLSTATE and SQLCODE values, save the values immediately to other SQL variables to avoid having the values replaced by the SQL return codes returned after executing the next SQL statement. If a handler is defined that handles an SQLSTATE, you can use an assignment statement to save that SQLSTATE (or the associated SQLCODE) value in another SQL variable, if the assignment is the first statement in the handler.

Assignment to these variables is not prohibited; however, it is not recommended. Assignment to these variables is ignored by condition handlers, and processing an assignment to these special variables causes the specified values for the assignment to be overlayed with the SQL return codes returned from executing the statement that does the assignment. The SQLSTATE and SQLCODE SQL variables cannot be set to NULL.

statement-declaration

Declares a list of one or more names that are local to the compound statement. A statement name cannot be the same as another statement name within the same compound statement.

DECLARE-CURSOR-statement

Declares a cursor in the procedure body. Each cursor must have a unique name within the routine. The cursor can only be referenced from within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. Use an OPEN statement to open the cursor, a FETCH statement to read a row using the cursor, and a CLOSE statement to close the cursor. If the cursor is intended for use as a result set cursor:

- Specify WITH RETURN when the cursor is declared
- Create the procedure using the DYNAMIC RESULT SETS clause with a non-zero value
- Do not specify a CLOSE statement for the cursor in the compound statement

For additional information about declaring a cursor, see “DECLARE CURSOR” on page 1191.

handler-declaration

Specifies a condition handler, an *SQL-procedure-statement* to execute when an exception or completion condition occurs in the *compound-statement*. The *SQL-procedure-statement* executes when a condition handler receives control.

A condition handler declaration cannot reference the same condition value or SQLSTATE value more than one time. It cannot reference an SQLSTATE value and a condition name that represent the same SQLSTATE value.

When two or more condition handlers are declared in a compound statement, no two condition handler declarations can specify the same:

- general condition category
- specific condition, either as an SQLSTATE value or as a condition name that represents the same value

A condition handler is active for the set of *SQL-procedure-statements* that follow the condition handler declarations within the compound statement in which the condition handler is declared, including any nested compound statements.

CONTINUE

Specifies that after the condition handler is activated and completes successfully, control is returned to the SQL statement that follows the statement that raised the condition. However, if the condition is an error condition and it was encountered while evaluating a search condition, as in a CASE, FOR, IF, REPEAT or WHILE statement, control returns to the statement that follows the corresponding END CASE, END FOR, END IF, END REPEAT, or END WHILE.

EXIT

Specifies that after the condition handler is activated and completes successfully, control is returned to the end of the compound statement that declared the condition handler.

The conditions that can cause the handler to gain control are:

SQLSTATE *string-constant*

Specifies that the handler is invoked when the specific SQLSTATE occurs. The first two characters of the SQLSTATE value must not be '00'.

SQL-condition-name

Specifies that the handler is invoked when the specific SQLSTATE that is associated with the condition name occurs. The *SQL-condition-name* must be declared within the compound statement that contains the handler declarations, or within a compound statement in which that compound statement is nested.

SQLEXCEPTION

Specifies that the handler is invoked when an SQLEXCEPTION occurs. An SQLEXCEPTION is an SQLSTATE in which the class code is a value other than '00', '01', or '02'. For more information on SQLSTATE values, see *DB2 Codes*.

SQLWARNING

Specifies that the handler is invoked when an SQLWARNING occurs. An SQLWARNING is an SQLSTATE value with a class code of '01'.

NOT FOUND

Specifies that the handler is invoked when a NOT FOUND condition occurs. NOT FOUND corresponds to an SQLSTATE value with a class code of '02'.

Notes

Unlike host variables, SQL variables are not preceded by colons when they are used in SQL statements.

Nesting compound statements: Compound statements can be nested. Nested compound statements can be used to scope variable definitions, condition names, condition handlers, and cursors to a subset of the statements in a routine. This can simplify the processing that is done for each SQL routine statement. Nested compound statements enable the use of a compound statement within the declaration of a condition handler.

The scope of a cursor: The scope of a cursor name is the compound statement in which it is declared, including any compound statements that are nested within that compound statement. A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement.

Considerations for statement-name: The scope of a *statement-name* that is declared in a compound statement is the compound statement and any nested compound statements (unless the same *statement-name* is declared in a nested compound statement). If a *statement-name* is used in a DECLARE CURSOR statement or a PREPARE statement and has not been declared in the compound statement where it is used or any outer compound statements in which it is nested, the *statement-name* is assumed to be declared globally for the routine.

Condition handlers: Condition handlers in SQL routines are similar to WHENEVER statements that are used in external SQL application programs. A condition handler can be defined to automatically get control when an exception, warning, or not found condition occurs. The body of a condition handler contains code that is executed when the condition handler is activated. A condition handler can be activated as the result of an exception, a warning, or a not found condition that is returned by DB2 for the processing of an SQL statement. Or the condition that activates the handler can be the result of a SIGNAL or RESIGNAL statement that is issued within the SQL routine body.

A condition handler is declared within a compound statement, and it is active for the set of *SQL-procedure-statements* that follow all of the condition handler declarations within the compound statement in which the condition handler is declared. For example, the scope of a condition handler declaration H is the list of *SQL-procedure-statements* that follow the condition handler declarations that are contained within the compound statement in which H appears. This means that the scope of H does not include the statements that are contained in the body of the condition handler H, implying that a condition handler cannot handle conditions that arise inside its own body. Similarly, for any two condition handlers H1 and H2 that are declared in the same compound statement, H1 will not handle conditions that arise in the body of H2, and H2 will not handle conditions that arise in the body of H1.

The declaration of a condition handler specifies the condition that activates it, the type of condition handler (CONTINUE or EXIT), and the handler action. The type of condition handler determines to where control is returned after the handler action successfully completes.

Condition handler activation: When a condition other than a successful completion occurs in the processing of *SQL-procedure-statement*, if a condition handler that could handle the condition is within scope, one such condition handler will be activated to process the condition.

In a routine with nested compound statements, condition handlers that could handle a specific condition might exist at several levels of the nested compound statements. The condition handler that is activated is a condition handler that is declared innermost to the scope in which the condition was encountered. If more than one condition handler at the nesting level could handle the condition, the condition handler that is activated is the most appropriate handler that is declared in that compound statement.

The most appropriate handler is the condition handler that most closely matches the SQLSTATE or the exception or completion condition. For a given compound statement, when both a specific handler for a condition and a general handler are declared that address the same condition, the specific handler takes precedence over the general handler.

For example, if the innermost compound statement declares a specific handler for SQLSTATE '22001', as well as a general handler for SQLEXCEPTION, the specific handler for SQLSTATE '22001' is the most appropriate handler when SQLSTATE '22001' is encountered. In this case, the specific handler is activated.

When a condition handler is activated, the condition handler action is executed. If the handler action completes successfully or with an unhandled warning, the diagnostics area is cleared, and the type of the condition handler (CONTINUE or EXIT handler) determines to where control is returned. Additionally, the SQLSTATE and SQLCODE SQL variables are cleared when a handler completes successfully or with an unhandled warning.

If the handler action does not complete successfully and an appropriate handler exists for the condition that is encountered in the handler action, that condition handler is activated. Otherwise, the condition that is encountered within the condition handler is unhandled.

Unhandled conditions: If a condition is encountered and an appropriate handler does not exist for that condition, the condition is unhandled.

- If the unhandled condition is an exception, the SQL routine that contains the failing statement is terminated with an unhandled exception condition.
- If the unhandled condition is a warning or is a not found condition, processing continues with the next statement. Note that the processing of the next SQL statement will cause information about the unhandled condition in the diagnostics area to be overwritten, and evidence of the unhandled condition will no longer exist.

Considerations for using SIGNAL and RESIGNAL statements with nested compound statements: If an *SQL-procedure-statement* that is specified in the condition handler is either a SIGNAL or RESIGNAL statement with an exception SQLSTATE, the compound statement terminates with the specified exception. This happens even when this condition handler or another condition handler in the same compound statement specifies CONTINUE, since these condition handlers are not in the scope of this exception. If a compound statement is nested in another compound statement, condition handlers in the higher level compound statement can handle the exception because those condition handlers are within the scope of the exception.

SQLSTATE and SQLCODE variables in SQL routines: To help debug your SQL routines, you might find it useful to check the SQLSTATE and SQLCODE value after executing a statement. An SQLCODE or SQLSTATE variable can be declared and subsequently referenced in an SQL routine. You could insert the value of the SQLCODE and SQLSTATE into a table at various points in the SQL routine, or return the SQLCODE and SQLSTATE values in a diagnostics string as an OUT parameter. To use the SQLCODE and SQLSTATE values, you must declare the following SQL variables in the SQL routine body:

```
DECLARE SQLCODE INTEGER DEFAULT 0;  
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

When you reference the SQLCODE or SQLSTATE variables in an SQL routine, DB2 sets the value of SQLCODE to 0 and SQLSTATE to '00000' for the subsequent statement. You can also use CONTINUE condition handlers to assign the value of the SQLSTATE and SQLCODE variables to variables in your SQL routine body. You can then use these SQL variables to control your routine logic, or pass the value back as an output parameter. In the following example, the SQL routine returns control to the statement following each SQL statement with the SQLCODE set in a SQL variable called RETCODE:

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE retcode INTEGER DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR SQLWARNING SET retcode = SQLCODE;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET retcode = SQLCODE;
```

The compound statement itself does not affect the SQLSTATE and SQLCODE SQL variables. However, SQL statements contained within the compound statement can affect the SQLSTATE and SQLCODE SQL variables. At the end of the compound statement, the SQLSTATE and SQLCODE SQL variables reflect the result of the last SQL statement executed within the compound statement that caused a change to the SQLSTATE and SQLCODE SQL variables. If the SQLSTATE and SQLCODE SQL variables were not changed within the compound statement, they contain the same values as when the compound statement was entered.

Null values in SQL parameters and SQL variables: If the value of an SQL parameter or SQL variable is null and it is used in an SQL statement that does not allow an indicator variable, an error is returned.

Effect on open cursors: At the end of the compound statement, all open cursors that are declared in that compound statement, except cursors that are used to return result sets, are closed.

Atomic processing of a compound statement: Atomic processing is not supported for a compound statement. If atomic behavior is needed for a block of code in a compound statement, set a savepoint before the nested compound statement is entered. This will allow changes to be undone with a ROLLBACK TO SAVEPOINT statement.

Examples

Example 1: Create a procedure body with a compound statement that performs the following actions:

- Declares SQL variables.
- Declares a cursor to return the salary of employees in a department determined by an IN parameter.
- Declares an EXIT handler for the condition NOT FOUND (end of file) which assigns the value 6666 to the OUT parameter *medianSalary*.
- Select the number of employees in the given department into the SQL variable *v_numRecords*.
- Fetch rows from the cursor in a WHILE loop until 50% + 1 of the employees have been retrieved.
- Return the median salary.

```
CREATE PROCEDURE DEPT_MEDIAN
  (IN deptNumber SMALLINT,
   OUT medianSalary DOUBLE)
LANGUAGE SQL
```

```

| BEGIN
|   DECLARE v_numRecords INTEGER DEFAULT 1;
|   DECLARE v_counter INTEGER DEFAULT 0;
|   DECLARE c1 CURSOR FOR
|     SELECT salary FROM staff
|       WHERE DEPT = deptNumber
|       ORDER BY salary;
|   DECLARE EXIT HANDLER FOR NOT FOUND
|     SET medianSalary = 6666;
|   /* initialize OUT parameter */
|   SET medianSalary = 0;
|   SELECT COUNT(*) INTO v_numRecords FROM staff
|     WHERE DEPT = deptNumber;
|   OPEN c1;
|   WHILE v_counter < (v_numRecords / 2 + 1) DO
|     FETCH c1 INTO medianSalary;
|     SET v_counter = v_counter + 1;
|   END WHILE;
|   CLOSE c1;
| END

```

Example 2: Define an exit handler for any error, warning, or case of end of data. When this procedure is invoked and it returns to the caller, the value 45000 is returned for the output parameter:

```

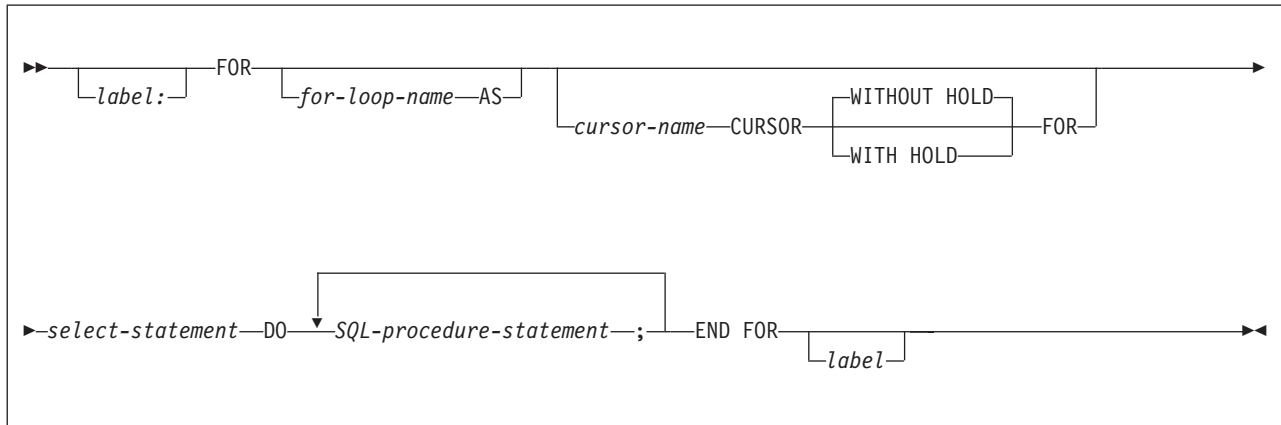
| CREATE PROCEDURE JMBLIB.PROCL(OUT MEDIANSALARY INT)
|   LANGUAGE SQL
|   BEGIN
|     DECLARE CHAR1 CHAR;
|     DECLARE C1 CURSOR FOR SELECT *
|       FROM SYSIBM.SYSDUMMY1;
|     DECLARE EXIT HANDLER FOR NOT FOUND,
|       SQLEXCEPTION,
|       SQLWARNING
|       OPEN C1;
|     FETCH C1 INTO CHAR1;
|     SET MEDIANSALARY = 45000;
|     FETCH C1 INTO CHAR1;
|     SET MEDIANSALARY = 50000;
|   END

```

FOR statement

The FOR statement executes a statement for each row of a table. An implicit compound statement is generated to implement the FOR statement.

Syntax



Description

label

Specifies the label for the FOR statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as the routine name or another label within the same scope. For more information, see “References to labels” on page 1543.

for-loop-name

Specifies the label for the implicit compound statement that is generated to implement the FOR statement. *for-loop-name* follows the rules for the label of a compound statement except that it cannot be used with an ITERATE, GOTO, or LEAVE statement within the FOR statement. *for-loop-name* must not be the same as any label within the same scope.

for-loop-name can be used to qualify generated SQL variables that correspond to the columns that are returned by *select-statement*.

cursor-name

Names a cursor that is generated to select rows from the result table of *select-statement*. If *cursor-name* is not specified, a unique cursor name is generated.

cursor-name cannot be referenced outside of the FOR statement and cannot be specified on an OPEN, FETCH, or CLOSE statement.

WITH HOLD or WITHOUT HOLD

Specifies whether the cursor should be prevented from being closed as a consequence of a commit operation.

WITHOUT HOLD

Specifies that the cursor is not prevented from being closed as a consequence of a commit operation. **WITHOUT HOLD** is the default.

WITH HOLD

Specifies that the cursor should not be closed as a consequence of a commit operation. A cursor that is declared using the **WITH HOLD** clause is implicitly closed at commit time only if the connection that is associated

with the cursor is ended during the commit operation. For more information, see “DECLARE CURSOR” on page 1191.

select-statement

Specifies the select statement of the cursor.

Each expression in the SELECT list must have a name. If an expression is not a simple column name, the AS clause must be used to name the expression. If the AS clause is specified, that name is used for the generated SQL variable that corresponds to the column returned by *select-statement*. The names for all of the generated SQL variables must be unique.

SQL-procedure-statement

Specifies the SQL statements to be executed for each row of the table. The SQL statements must not include an OPEN, FETCH, or CLOSE statement that specifies the cursor name of the FOR statement.

Notes

FOR statement rules: The FOR statement executes one or multiple statements for each row in a table. The cursor is defined by specifying a SELECT list that describes the columns and rows selected. The statements within the FOR statement are executed for each row selected.

The SELECT list must consist of unique column names and the table that is specified in the FROM clause of *select-statement* must exist when the routine is created.

Handler warning: Handlers can be used to handle errors that might occur on the open of the cursor or fetch of a row using the cursor in the FOR statement. Handlers defined to handle these open or fetch conditions should not be CONTINUE handlers as they might cause the FOR statement to loop indefinitely.

Examples

In the following example, the FOR statement is used to specify a cursor that selects three columns from the employee table. For every row selected, SQL variable *fullname* is set to the last name followed by a comma, the first name, a blank, and the middle initial. Each value for *fullname* is inserted into table TNAMES.

```
BEGIN
  DECLARE fullname CHAR(40);
  FOR v1 AS
    c1 CURSOR FOR
      SELECT firstname, midinit, lastname FROM employee
  DO
    SET fullname =
      lastname CONCAT ', '
      CONCAT firstname
      CONCAT ' '
      CONCAT midinit;
    INSERT INTO TNAMES VALUES ( fullname );
  END FOR;
END;
```

GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

See “GET DIAGNOSTICS” on page 1317.

When you need to specify a variable in a GET DIAGNOSTICS statement that is used within an SQL routine, you would use either *SQL-variable-name* or *SQL-parameter-name*. In an embedded GET DIAGNOSTICS statement, you would use a *host-variable*. You can replace the instances of *host-variable* in the description of “GET DIAGNOSTICS” on page 1317 with *SQL-variable-name* or *SQL-parameter-name*.

Effects of the statement: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area except for DB2_GET_DIAGNOSTICS_DIAGNOSTICS.

Considerations for the SQLSTATE and SQLCODE SQL variables: The GET DIAGNOSTICS statement does not change the value of the SQLSTATE and SQLCODE SQL variables.

GOTO statement

The GOTO statement is used to branch to a user-defined label within an SQL routine. The GOTO statement is used to branch to a user-defined label within an SQL procedure.

Syntax



```
label: GOTO target-label
```

Description

label

Specifies the label for the GOTO statement. The label name cannot be the same as the name of the SQL routine in which the label is used or another label in the same scope.

target-label

Specifies a label of the statement where processing is to continue. *target-label* must be defined as a label for an SQL procedure statement. The target label must be accessible to the GOTO statement as defined in “References to labels” on page 1543, subject to the following restrictions:

- If the GOTO statement is in a condition handler, *target-label* must be defined in that condition handler.
- If the GOTO statement is not defined in a condition handler, *target-label* must not be defined in a condition handler.

Notes

Using a GOTO statement: It is recommended that the GOTO statement be used sparingly. This statement interferes with the normal sequence of processing SQL statements, thus making a routine more difficult to read and maintain. Before using a GOTO statement, determine whether another statement, such as IF or LEAVE, can be used in place, to eliminate the need for a GOTO statement.

Effect on open cursors: When a GOTO statement transfers control out of a compound statement, all open cursors that are declared in the compound statement that contains the GOTO statement are closed, unless those statements are declared to return result sets. If the GOTO statement transfers control out of any directly or indirectly nested compound statements, all open cursors that are declared in those nested compound statements are also closed, unless those cursors are declared to return result sets.

Examples

Example 1: In the following procedure, the GOTO statement branches outside of the current compound statement to a higher level:

```
CREATE PROCEDURE TESTGOTO5 ( )
P1: BEGIN
    DECLARE I ,A INTEGER;
    SET I = 1;
    LAB1: SET A = 1;
```



```
BEGIN
    LAB2: SET A = 2;
    BEGIN
        SET I = I+1;
        IF I<3 THEN GOTO LAB1;
        END IF;
    END;
END;
END P1
```

Example 2: In the following example, cursors are declared at multiple levels. The GOTO statement that specified TargLabel as the target label, results in the closing of cursors C1, C2, and C3. This is because cursors C1, C2, and C3 are all declared directly or indirectly in the compound statement with the label L1. The GOTO statement causes control to transfer out of the compound statement with label L1, so the cursors that are defined within that compound statement (at any level) are closed.

```

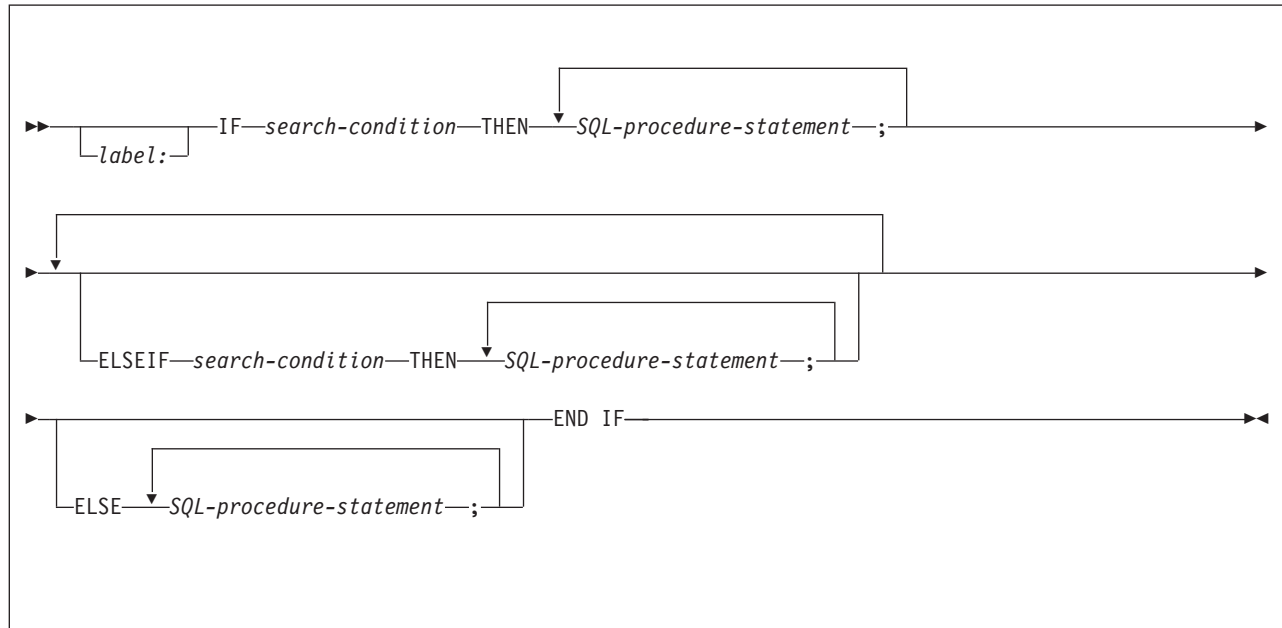
L0: BEGIN
    DECLARE CURSOR C0 ...
    ...
    TARGLABEL: ...
    ...
    L1: BEGIN
        DECLARE CURSOR C1 ...
        ...
        L2: BEGIN
            DECLARE CURSOR C2 ...
            ...
            GOTO TARGLABEL;
            ...
            L3: BEGIN
                DECALUE CURSOR C3 ...
                ...
            END L3;
        END L2;
    END L1;
END L0

```

IF statement

The IF statement executes different sets of SQL statements based on the result of search conditions.

Syntax



Description

label

Specifies the label for the IF statement. The label name cannot be the same as the name of the SQL routine or another label name within the same scope. For additional information, see “References to labels” on page 1543.

search-condition

Specifies the *search-condition* for which an SQL statement should be executed. If the condition is unknown or false, processing continues to the next search condition, until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement

Specifies an SQL statement to be executed if the preceding *search-condition* is true. See “SQL-procedure-statement” on page 1546.

Examples

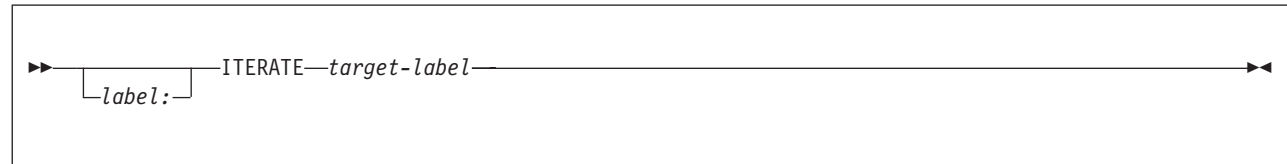
Assign a value to the SQL variable *new_salary* based on the value of SQL variable *rating*.

```
IF rating = 1
  THEN SET new_salary =
    new_salary + (new_salary * .10);
ELSEIF rating = 2
  THEN SET new_salary =
    new_salary + (new_salary * .05);
ELSE SET new_salary =
    new_salary + (new_salary * .02);
END IF
```

ITERATE statement

The ITERATE statement causes the flow of control to return to the beginning of a labeled loop.

Syntax



Description

label

Specifies the label for the ITERATE statement. The label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see “References to labels” on page 1543.

target-label

Specifies the label of the FOR, LOOP, REPEAT, or WHILE statement to which the flow of control is passed. *target-label* must be defined as a label for a FOR, LOOP, REPEAT, or WHILE statement. The ITERATE statement must be in that FOR, LOOP, REPEAT, or WHILE statement, or in the block of code that is directly or indirectly nested within that statement, subject to the following restrictions:

- If the ITERATE statement is in a condition handler, *target-label* must be defined in that condition handler.
- If the ITERATE statement is not in a condition handler, *target-label* must not be defined in a condition handler.

Examples

Example 1: This example uses a cursor to return information for a new department. If the not_found condition handler is invoked, the flow of control passes out of the loop. If the value of *v_dept* is 'D11', an ITERATE statement causes the flow of control to be passed back to the top of the LOOP statement. Otherwise, a new row is inserted into the table.

```
CREATE PROCEDURE ITERATOR ()
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
  DECLARE v_dept CHAR(3);
  DECLARE v_deptname VARCHAR(29);
  DECLARE v_admdept CHAR(3);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE c1 CURSOR FOR
    SELECT deptno,deptname,admdept
    FROM department
    ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found
    SET at_end = 1;
  OPEN c1;
  ins_loop:
  LOOP
    FETCH c1 INTO v_dept, v_deptname, v_admdept;
    IF at_end = 1 THEN
```

```

        LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
    END IF;
    INSERT INTO department (deptno,deptname,admrdept)
        VALUES('NEW', v_deptname, v_admdept);
END LOOP;
CLOSE c1;
END

```

Example 2: An ITERATE statement can be issued from a nested block to cause that flow of control to return to the beginning of a loop at a higher level. In the following example, the ITERATE statement within the LAB2 compound statement causes the flow of control to return to the beginning of the LAB1 LOOP statement:

```

LAB1: LOOP
    SET A = 0;
    LAB2: BEGIN
        ...
        LAB3: BEGIN
            ...
            ITERATE LAB1; -- Multilevel ITERATE
            ...
        END LAB3;
        ...
        ITERATE LAB1; -- Multilevel ITERATE
        ...
    END LAB2;
END LOOP;S

```

LEAVE statement

The LEAVE statement transfers program control out of a FOR, LOOP, REPEAT, WHILE, or compound statement.

Syntax

```
┌──┐ ─── LEAVE—target-label ───┐
|  |                             |
|  |                             |
└──┘
```

Description

label

Specifies the label for the LEAVE statement. The label name cannot be the same as the name of the SQL routine or the same as another label that is within the same scope. For additional information, see “References to labels” on page 1543.

target-label

Specifies the label of the compound, FOR, LOOP, REPEAT, or WHILE statement to exit. *target-label* must be defined as a label for a compound, FOR, LOOP, REPEAT, or WHILE statement, or in a block of code that is directly or indirectly nested within that statement, subject to the following rules:

- If the LEAVE statement is in a condition handler, *target-label* must be defined in that condition handler.
- If the LEAVE statement is not in a condition handler, *target-label* must not be defined in a condition handler.

Notes

Effect on open cursors: When a LEAVE statement transfers control out of a compound statement, all open cursors that are declared in the compound statement that contain the LEAVE statement are closed, except cursors that are used to return result sets.

Examples

Example 1: The example contains a loop that fetches data for cursor c1. If the value of SQL variable *at_end* is not zero, the LEAVE statement transfers control out of the loop.

```
CREATE PROCEDURE LEAVE_LOOP (OUT COUNTER INTEGER)
LANGUAGE SQL
BEGIN
    DECLARE v_counter INTEGER;
    DECLARE v_firstname VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE at_end SMALLINT DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
        SELECT firstname, midinit, lastname
        FROM employee;
    DECLARE CONTINUE HANDLER FOR not_found
        SET at_end = 1;
    SET v_counter = 0;
```

```

OPEN c1;
fetch_loop:
LOOP
    FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
    IF at_end <> 0 THEN
        LEAVE fetch_loop;
    END IF;
    SET v_counter = v_counter + 1;
END LOOP fetch_loop;
SET counter = v_counter;
CLOSE c1;
END

```

Example 2: A LEAVE statement can be issued from a nested block to leave a statement at a higher level. In the following example, the LEAVE statement within the LAB2 compound statement causes the LAB1 LOOP statement to terminate:

```

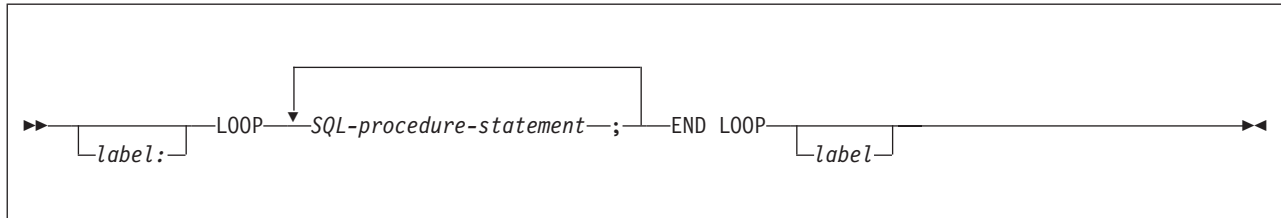
LAB1: LOOP
    ...
    LAB2: BEGIN
        SET A = 0;
        ...
        LAB3: BEGIN
            ...
            LEAVE LAB1; -- Multilevel LEAVE
            ...
        END LAB3;
        ...
        LEAVE LAB1; -- Multilevel LEAVE
        ...
    END LAB2;
END LOOP;S

```

LOOP statement

The LOOP statement executes a statement or group of statements multiple times.

Syntax



Description

label

| Specifies the label for the LOOP statement. If the ending label is specified, a
| matching beginning label must be specified. A label name cannot be the same
| as the name of the SQL routine or another label within the same scope. For
| additional information, see “References to labels” on page 1543.

SQL-procedure-statement

| Specifies an SQL statement to be executed in the loop. The statement must be
| one of the statements listed under “SQL-procedure-statement” on page 1546.

Notes

| **Considerations for the diagnostics area:** At the beginning of the first iteration of
| the LOOP statement, and with every subsequent iteration, the diagnostics area is
| cleared.

| **Considerations for the SQLSTATE and SQLCODE SQL variables:** Prior to
| executing the first SQL-procedure-statement within that LOOP statement, the
| SQLSTATE and SQLCODE values reflect the last values that were set prior to the
| LOOP statement. If the loop is terminated with a GOTO or a LEAVE statement, the
| SQLSTATE and SQLCODE values reflect successful completion of that statement.
| When the LOOP statement iterates, the SQLSTATE and SQLCODE values reflect
| the result of the last SQL statement that is executed within the LOOP statement.

Examples

| This procedure uses a LOOP statement to fetch values from the employee table.
| Each time the loop iterates, the OUT parameter counter is incremented and the
| value of *v_midinit* is checked to ensure that the value is not a single space (' '). If
| *v_midinit* is a single space, the LEAVE statement passes the flow of control outside
| of the loop.

```
| CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
|   LANGUAGE SQL
|   BEGIN
|       DECLARE v_counter INTEGER DEFAULT 0;
|       DECLARE v_firstname VARCHAR(12);
|       DECLARE v_midinit CHAR(1);
|       DECLARE v_lastname VARCHAR(15);
|       DECLARE c1 CURSOR FOR
|           SELECT firstname, midinit, lastname
|             FROM employee;
|       DECLARE EXIT HANDLER FOR NOT FOUND
```

```

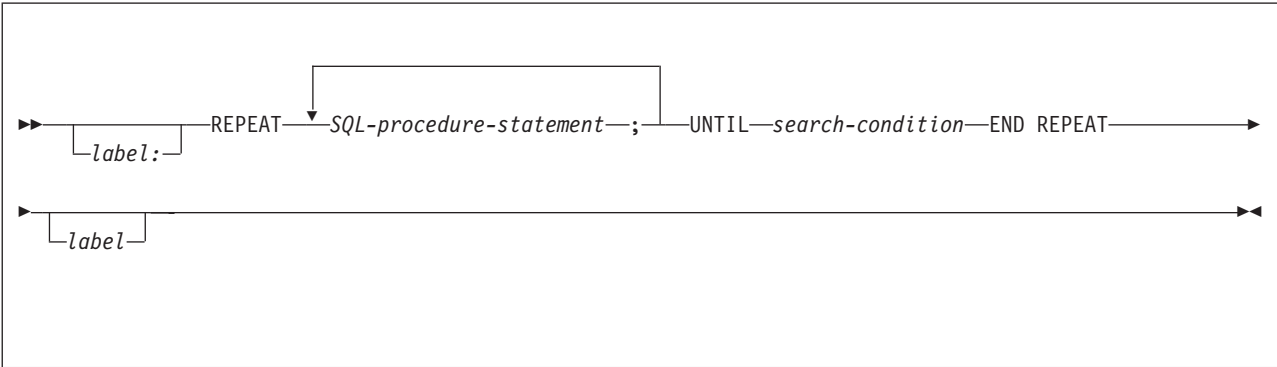
|         SET counter = -1;
| OPEN c1;
| fetch_loop:
| LOOP
|     FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
|     IF v_midinit = ' ' THEN
|         LEAVE fetch_loop;
|     END IF;
|     SET v_counter = v_counter + 1;
| END LOOP fetch_loop;
| SET counter = v_counter;
| CLOSE c1;
| END

```


REPEAT statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Description

label

| Specifies the label for the REPEAT statement. If an ending label is specified, a
| matching beginning label must be specified. A label name cannot be the same
| as the name of the SQL routine or another label within the same scope. For
| additional information, see “References to labels” on page 1543.

SQL-procedure-statement

| Specifies an SQL statement to be executed within the REPEAT loop. The
| statement must be one of the statements listed under
| “SQL-procedure-statement” on page 1546.

search-condition

| Specifies a condition that is evaluated after each execution of the REPEAT loop.
| If the condition is true, the REPEAT loop will exit. If the condition is unknown
| or false, the looping continues.

Notes

| **Considerations for the diagnostics area:** At the beginning of the first iteration of
| the REPEAT statement, and with every subsequent iteration, the diagnostics area is
| cleared.

| **Considerations for the SQLSTATE and SQLCODE SQL variables:** At the beginning
| of the first iteration of the REPEAT statement, the SQLSTATE and SQLCODE
| values reflect the values that were set prior to the REPEAT statement. At the
| beginning of iterations 2 through *n* of the REPEAT statement, the SQLSTATE and
| SQLCODE SQL values reflect the result of evaluating the search condition in the
| UNTIL clause of that REPEAT. If the loop is terminated with a GOTO, ITERATE,
| or LEAVE statement, the SQLSTATE and SQLCODE values reflect the successful
| completion of that statement. Otherwise, after the END REPEAT of the REPEAT
| statement completes, the SQLSTATE and SQLCODE reflect the result of evaluating
| the search condition in the UNTIL clause of that REPEAT statement.

Examples

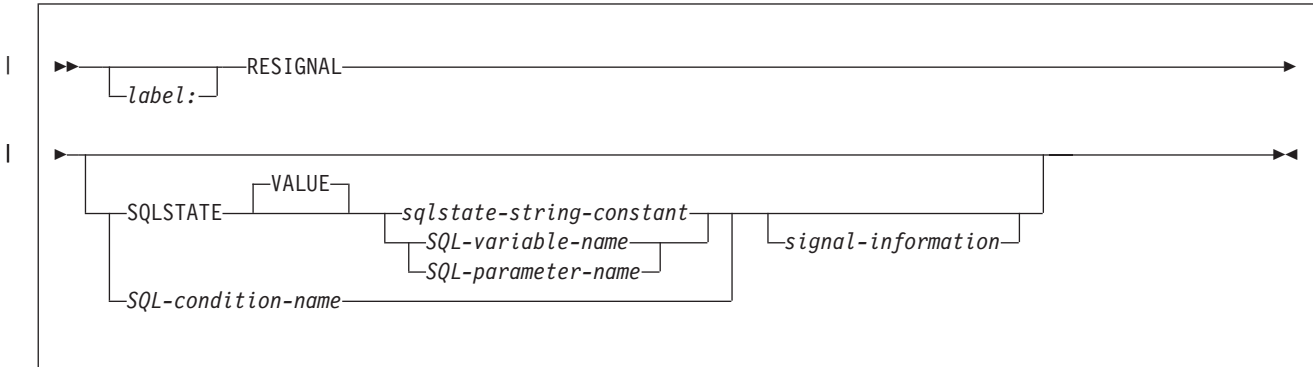
Use a REPEAT statement to fetch rows from a table.

```
fetch_loop:
REPEAT
  FETCH c1 INTO
    v_firstname, v_midinit, v_lastname;
UNTIL
  SQLCODE <> 0
END REPEAT fetch_loop
```

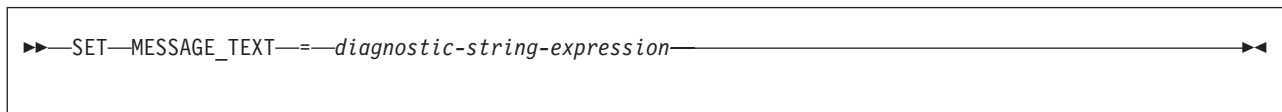
RESIGNAL statement

The RESIGNAL statement is used within a condition handler to resignal the condition that activated the handler, or to raise an alternate condition so that it can be processed at a higher level. It causes an exception, warning, or not found condition to be returned along with optional message text.

Syntax



signal-information:



Description

label

Specifies the label for the RESIGNAL statement. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see “References to labels” on page 1543.

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATE values:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').
- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant

A character string constant with an actual length of five bytes that is a valid SOLSTATE value.

SQL-variable-name or *SQL-parameter-name*

Specifies an SQL variable or SQL parameter that is defined for the routine.

SQL-variable-name

Specifies an SQL variable that is declared within the *compound-statement* that contains the RESIGNAL statement or within a compound statement in which that compound statement is nested.

SQL-variable-name must be defined as CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL_parameter-name

Specifies an SQL parameter that is defined for the routine that contains the SQLSTATE value. The SQL parameter must be defined as CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-condition-name

Specifies the name of the condition that will be returned. *SQL-condition-name* must be declared within the *compound-statement* that contains the RESIGNAL statement, or within a compound statement in which that compound statement is nested.

SET MESSAGE_TEXT

Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression

An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see “GET DIAGNOSTICS” on page 1317.

Notes

While any valid SQLSTATE value can be used in the RESIGNAL statement, programmers should define new SQLSTATES based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If the SQLSTATE or condition indicates that an exception is signaled (SQLSTATE class other than '01' or '02'):

- If a condition handler exists in the same compound statement as the RESIGNAL statement, and the compound statement contains a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.
- If the compound statement is nested and an outer level compound statement has a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If an SQLSTATE or a condition indicates that a warning or a not found condition is signaled:

- If a condition handler exists in the same compound statement as the RESIGNAL statement, and the compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified SQLSTATE or condition, the warning or not found condition is handled and control is transferred to that condition handler.
- If the compound statement is nested and an outer level compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified

SQLSTATE or condition, the warning or not found condition is handled and control is returned to that condition handler.

- Otherwise, the warning is not handled and processing continues with the next statement.

Considerations for the diagnostics area: The RESIGNAL statement might modify the contents of the current diagnostics area. If an SQLSTATE or *condition-name* is specified as part of the RESIGNAL statement, the RESIGNAL statement starts with a clear diagnostics area and sets the RETURNED_SQLSTATE to reflect the specified SQLSTATE or *condition-name*. If message text is specified, the MESSAGE_TEXT item of the condition area is assigned the specified value. DB2_RETURNED_SQLCODE is set to +438 or -438 corresponding to the specified SQLSTATE or *condition-name*.

Examples

The following example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the overflow condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE to the client application.

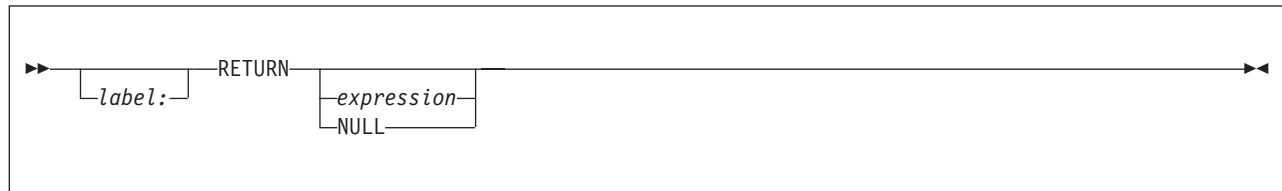
```
CREATE PROCEDURE divide (IN numerator INTEGER,
                        IN denominator INTEGER,
                        OUT divide_result INTEGER)
LANGUAGE SQL
CONTAINS SQL
BEGIN
  DECLARE overflow CONDITION for SQLSTATE '22003';
  DECLARE CONTINUE HANDLER FOR overflow
    RESIGNAL SQLSTATE '22375';
  IF denominator = 0 THEN
    SIGNAL overflow;
  ELSE
    SET divide_result = numerator / denominator;
  END IF;
END
```

RETURN statement

The RETURN statement is used to return from the routine.

For SQL functions, it returns the result of the function. For an SQL procedure, it optionally returns an integer status value.

Syntax



Description

label

Specifies the label for the RETURN statement. *label* must not be specified within an SQL function. A label name cannot be the same as the name of the SQL procedure or another label within the same scope. For additional information, see “References to labels” on page 1543.

expression

Specifies a value that is returned from the routine.

- If the routine is a function, *expression* must be specified and the value of *expression* must conform to the SQL assignment rules as described in “Assignment and comparison” on page 102. If the value is being assigned to a string variable, storage assignment rules apply.
- If the routine is a procedure, the data type of *expression* must be INTEGER. If *expression* evaluates to the null value, a value of 0 is returned.

The *expression* cannot include a column name. See “Expressions” on page 180 for information on expressions. For an SQL function, the *expression* cannot contain a scalar fullselect.

NULL

The null value is returned from the SQL routine. **NULL** is not allowed in SQL procedures.

Notes

Considerations for SQL procedures:

- **When a RETURN statement is used within an SQL procedure:** If a RETURN statement with a specified return value was used to return from a procedure, the SQLCODE, SQLSTATE, and message length in the SQLCA are initialized to zeros and the message text is set to blanks. An error is not returned to the caller.
- **When a RETURN statement is not used within an SQL procedure or when no value is specified:** If a RETURN statement was not used to return from a procedure or if a value is not specified on the RETURN statement, one of the following values is set:
 - If the procedure returns with an SQLCODE that is greater or equal to zero, the specified target for DB2_RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of zero.

- If the procedure returns with an SQLCODE that is less than zero, the specified target for DB2_RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of '-1'.
- **When the value is returned from an SQL procedure:** When a value is returned from a procedure, the caller may access the value using one of the following methods:
 - The GET DIAGNOSTICS statement to retrieve the RETURN_STATUS when the SQL procedure was called from another SQL procedure.
 - The parameter bound for the return value parameter marker in the escape clause CALL syntax (=?CALL...) in a CLI application.
 - Directly from the SQLCA returned from processing the CALL of an SQL procedure by retrieving the value of sqlerrd[0]. When the SQLCODE is less than zero, the sqlerrd[0] value is not set. The application should assume a return status value of '-1'.

Examples

Example 1: Use a RETURN statement to return from an SQL procedure with a status value of zero if successful or '-200' if not successful.

```
BEGIN
    . . .
    GOTO FAIL;
    . . .
SUCCESS: RETURN 0;
FAIL: RETURN -200;
END
```

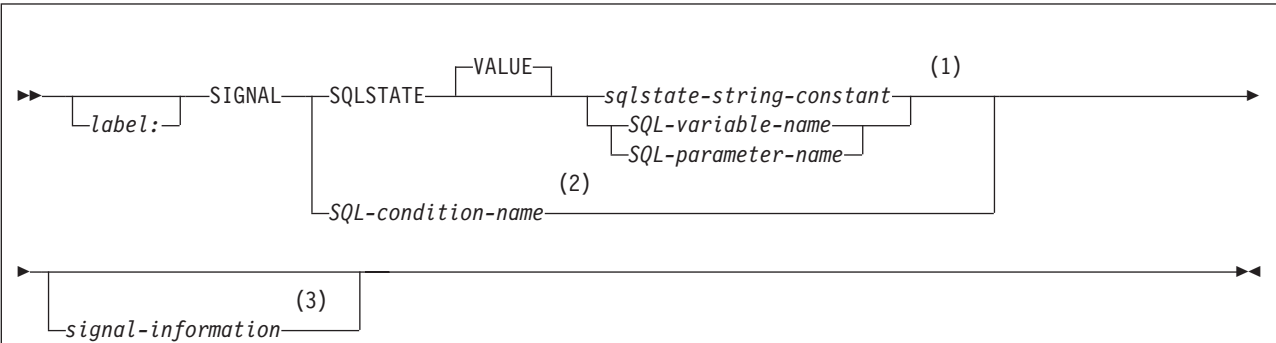
Example 2: Define a scalar function that returns the tangent of a value using the existing sine and cosine functions:

```
CREATE FUNCTION TAN (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL CONTAINS SQL NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X)
```

SIGNAL statement

The SIGNAL statement is used to return an exception or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text. The SIGNAL statement places the specified condition information in the cleared diagnostics area.

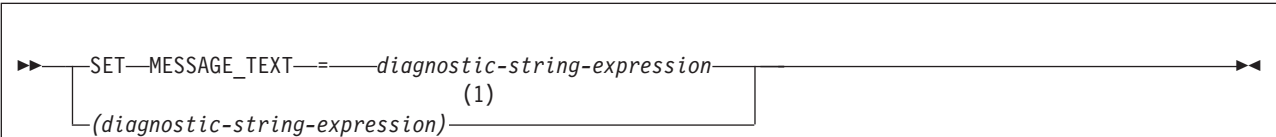
Syntax



Notes:

- 1 The **SQLSTATE** variation must be used within a trigger body.
- 2 *SQL-condition-name* must not be specified within a trigger body.
- 3 *signal-information* must be specified within a trigger body

signal-information:



Notes:

- 1 (*diagnostic-string-expression*) must only be specified within a trigger body.

Description

label

Specifies the label for the SIGNAL statement. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see “References to labels” on page 1543.

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATES:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').

- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant

A character string constant with an actual length of five bytes that is a valid SQLSTATE value.

SQL-variable-name or SQL-parameter-name

Specifies an SQL variable or SQL parameter that contains a valid SQLSTATE value.

SQL-variable-name

Specifies an SQL variable that is declared within the *compound-statement* that contains the SIGNAL statement, or within a compound statement in which that compound statement is nested. *SQL-variable-name* must be defined as a CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-parameter-name

Specifies an SQL parameter that is defined for the routine and contains the SQLSTATE value. The SQL parameter must be defined as a CHAR or VARCHAR data type with an actual length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-condition-name

Specifies the name of the condition that will be returned. The *SQL-condition-name* must be declared within the compound statement that contains the SIGNAL statement, or within a compound statement in which that compound statement is nested.

SET MESSAGE_TEXT

Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression

An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1317.

(diagnostic-string-expression)

An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1317.

This syntax variation is only provided within the scope of a CREATE TRIGGER statement for compatibility with previous versions of DB2. To conform with the ANS and ISO standards, this form should not be used.

Notes

While any valid SQLSTATE value can be used in the SIGNAL statement, programmers should define new SQLSTATES based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If the SQLSTATE or condition indicates that an exception is signaled:

- If a condition handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.
- If the compound statement is nested and the outer level compound statement has a condition handler for SQLEXCEPTION or the specified SQLSTATE or condition, the exception is handled and control is transferred to that condition handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If the SQLSTATE or condition indicates that a warning or not found condition is signaled:

- If a condition handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified SQLSTATE or condition, the warning or not found condition is handled and control is transferred to that condition handler.
- If the compound statement is nested and an outer level compound statement contains a condition handler for SQLWARNING, NOT FOUND, or the specified SQLSTATE or condition, the warning or not found condition is handled and control is transferred to that condition handler.
- Otherwise, the warning or not found condition is not handled and processing continues with the next statement.

Considerations for the diagnostics area: The SIGNAL statement starts with a clear diagnostics area and sets the RETURNED_SQLSTATE to reflect the specified SQLSTATE or *condition-name*. If message text is specified, the MESSAGE_TEXT item of the condition area is assigned the specified value. DB2_RETURNED_SQLCODE is set to +438 or -438 corresponding to the specified SQLSTATE or *condition-name*.

Using a SIGNAL statement in the body of a trigger: Within the triggered action of a CREATE TRIGGER statement, the message text can be specified using only these variations:

```
SIGNAL SQLSTATE sqlstate-string-constant
      SET MESSAGE_TEXT = diagnostic-string-expression
SIGNAL SQLSTATE sqlstate-string-constant
      (diagnostic-string-expression)
```

Examples

Example 1: The following example shows an SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
      (IN ONUM INTEGER, IN CNUM INTEGER,
      IN PNUM INTEGER, IN QNUM INTEGER)
LANGUAGE SQL
SPECIFIC SUBMIT_ORDER
MODIFIES SQL DATA
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
    SIGNAL SQLSTATE '75002'
```

```

        SET MESSAGE TEXT = 'Customer number is not known';
    INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
        VALUES (ONUM, CNUM, PNUM, QNUM);
END

```

Example 2: The following example shows a trigger for an order system that allows orders to be recorded in an ORDERS table (ORDERNO, CUSTNO, PARTNO, QUANTITY) only if there is sufficient stock in the PARTS tables. When there is insufficient stock for an order, SQLSTATE '75001' is returned along with an appropriate error description.

```

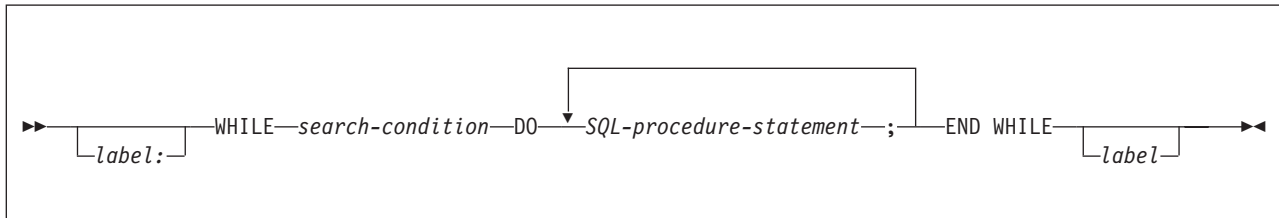
CREATE TRIGGER CK_AVAIL
    NO CASCADE BEFORE INSERT ON ORDERS
    REFERENCING NEW AS NEW_ORDER
    FOR EACH ROW MODE DB2SQL
    WHEN (NEW_ORDER.QUANTITY > (SELECT ON_HAND FROM PARTS
                                WHERE NEW_ORDER.PARTNO = PARTS.PARTNO))
    BEGIN ATOMIC
        SIGNAL SQLSTATE '75001' ('Insufficient stock for order');
    END

```

WHILE statement

The WHILE statement repeats the execution of a statement or group of statements while a specified condition is true.

Syntax



Description

label

Specifies the label for the WHILE statement. If the ending label is specified, it must be the same as the beginning label. A label name cannot be the same as the name of the SQL routine or another label within the same scope. For additional information, see “References to labels” on page 1543.

search-condition

Specifies a condition that is evaluated before each execution of the loop. If the condition is true, the SQL procedure statements in the loop are executed.

SQL-procedure-statement

Specifies a statement to be run within the WHILE loop. The statement must be one of the statements listed under “SQL-procedure-statement” on page 1546.

Notes

Considerations for the diagnostics area: At the beginning of the first iteration of the WHILE statement, and with every subsequent iteration, the diagnostics area is cleared.

Considerations for the SQLSTATE and SQLCODE SQL variables: With each iteration of the WHILE statement, when the first SQL-procedure-statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search condition of that WHILE statement. If the loop is terminated with a GOTO, ITERATE, or LEAVE statement, the SQLSTATE and SQLCODE values reflect the successful completion of that statement. Otherwise, after the END WHILE of the WHILE statement completes, the SQLSTATE and SQLCODE reflect the result of evaluating that search condition of that WHILE statement.

Examples

Use a WHILE statement to fetch rows from a table while SQL variable `at_end`, which indicates whether the end of the table has been reached, is 0.

```
WHILE at_end = 0 DO
  FETCH c1 INTO
    v_firstname, v_midinit,
    v_lastname, v_edlevel, v_salary;
  IF SQLCODE=100 THEN SET at_end=1;
  END IF;
END WHILE
```

Appendix. Additional information for DB2 SQL

These topics contain additional information for DB2 SQL.

Limits in DB2 for z/OS

DB2 for z/OS has system limits, object and SQL limits, length limits for identifiers and strings, and limits for certain data type values.

System storage limits might preclude the limits specified in this section. The limit for items not that are not specified below is limited by system storage.

The following table shows the length limits for identifiers.

Table 131. Identifier length limits. The term *byte(s)* in this table means the number of bytes for the UTF-8 representation unless noted otherwise.

Item	Limit
External-java-routine-name	1305 bytes
Name of an alias , auxiliary table, collection, clone table, constraint, correlation, cursor (except for DECLARE CURSOR WITH RETURN or the EXEC SQL utility), distinct type (both parts of two-part name), function (both parts of two-part name), host identifier, index, JARs, parameter, procedure, role, schema, sequence, specific, statement, storage group, savepoint, SQL condition, SQL label, SQL parameter, SQL variable, synonym, table, trigger, view, XML attribute name, XML element name	128 bytes
Name of an authorization ID or name of a security label.	8 bytes
Routine version identifier	64 EBCDIC bytes, and the UTF-8 representation of the name must not exceed 122 bytes.
Name of a column	30 bytes
Name of cursor that is created with DECLARE CURSOR WITH RETURN	30 bytes
Name of cursor that is created with the EXEC SQL utility	8 bytes
Name of a location	16 bytes
Name of buffer pool name, catalog, database, plan, program, table space	8 bytes
Name of package	8 bytes (Only 8 EBCDIC characters are used for packages that are created with the BIND PACKAGE command. 128 bytes can be used for packages that are created as a result of the CREATE TRIGGER statement.)
Name of a profile that is created with CREATE TRUSTED CONTEXT or ALTER TRUSTED CONTEXT	127 bytes

Table 132 shows the minimum and maximum limits for numeric values.

Table 132. Numeric limits

Item	Limit
Smallest SMALLINT value	-32768
Largest SMALLINT value	32767
Smallest INTEGER value	-2147483648
Largest INTEGER value	2147483647
Smallest BIGINT value	-9223372036854775808

Table 132. Numeric limits (continued)

[illegible]

The following table shows the length limits for strings.

Table 133. String length limits

Item	Limit
Maximum length of CHAR	255 bytes
Maximum length of GRAPHIC	127 double-byte characters
Maximum length of BINARY	255 bytes
Maximum length ¹ of VARCHAR	4046 bytes for 4 KB pages 8128 bytes for 8 KB pages 16320 bytes for 16 KB pages 32704 bytes for 32 KB pages
Maximum length of VARCHAR that can be indexed by an XML index	1000 bytes after conversion to UTF-8

Table 133. String length limits (continued)

Item	Limit
Maximum length ¹ of VARCHAR	2023 double-byte characters for 4 KB pages 4064 double-byte characters for 8 KB pages 8160 double-byte characters for 16 KB pages 16352 double-byte characters for 32 KB pages
Maximum length of VARBINARY	32704 bytes
Maximum length of CLOB	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of DBCLOB	1 073 741 823 double-byte characters
Maximum length of BLOB	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of a character constant	32704 UTF-8 bytes
Maximum length of a hexadecimal character constant	32704 hexadecimal digits
Maximum length of a graphic string constant	16352 double-byte characters (32704 bytes when expressed in UTF-8)
Maximum length of a hexadecimal graphic string constant	32704 hexadecimal digits
Maximum length of a text string used for a scalar expression	4000 UTF-8 bytes
Maximum length of a concatenated character string	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of a concatenated graphic string	1 073 741 824 double-byte characters
Maximum length of a concatenated binary string	2 147 483 647 bytes (2 GB - 1 byte)
Maximum length of XML pattern text	4000 bytes after conversion to UTF-8
Maximum length of an XML element or attribute name in an XML document	1000 bytes
Maximum length of a namespace uri	1000 bytes
maximum length of a namespace prefix	998 bytes
largest depth of an internal XML tree	128 levels

Note:

1. The maximum length can be achieved only if the column is the only column in the table. Otherwise, the maximum length depends on the amount of space remaining on a page.

The following table shows the minimum and maximum limits for datetime values.

Table 134. Datetime limits

Item	Limit
Smallest DATE value (shown in ISO format)	0001-01-01
Largest DATE value (shown in ISO format)	9999-12-31
Smallest TIME value (shown in ISO format)	00.00.00
Largest TIME value (shown in ISO format)	24.00.00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000

The following table shows the DB2 limits on SQL statements.

Table 135. DB2 limits on SQL statements

Item	Limit
Maximum number of columns that are in a table or view (the value depends on the complexity of the CREATE VIEW statement) or columns returned by a table function.	750 or fewer (including hidden columns) 749 if the table is a dependent
Maximum number of base tables in a view, SELECT, UPDATE, INSERT, MERGE, or DELETE	225
Maximum number of rows that can be inserted with a single INSERT or MERGE statement	32767
Maximum row and record sizes for a table	See Maximum record size under CREATE TABLE.
Maximum number of volume IDs in a storage group	133
Maximum number of partitions in a partitioned table space or partitioned index	64 for table spaces that are not defined with LARGE or a DSSIZE greater than 2 GB. 4096, depending on what is specified for DSSIZE or LARGE and the page size.
Maximum sum of the lengths of limit key values of a partition boundary	765 UTF-8 bytes
Maximum size of a partition (table space or index)	For table spaces that are not defined with LARGE or a DSSIZE greater than 2 GB: 4 GB, for 1 to 16 partitions 2 GB, for 17 to 32 partitions 1 GB, for 33 to 64 partitions For table spaces that are defined with LARGE: 4 GB, for 1 to 4096 partitions For table spaces that are defined with a DSSIZE greater than 2 GB: 64 GB, depending on the page size (for 1 to 256 partitions for 4 KB pages, for 1 to 512 partitions for 8 KB pages, for 1 to 1024 partitions for 16 KB pages, and 1 to 2048 partitions for 32 KB pages)
Maximum length of an index key	Partitioning index: $255-n$ Nonpartitioning index that is padded: $2000-n$ Nonpartitioning index that is not padded: $2000-n-2m$ Where n is the number of columns in the key that allow nulls and m is the number of varying-length columns in the key
Maximum number of bytes used in the partitioning of a partitioned index	255 (This maximum limit is subject to additional limitations, depending on the number of partitions in the table space. The number of partitions * (106 + limit key size) must be less than 65394.)
Maximum number of columns in an index key	64
Maximum number of expressions in an index key	64
Maximum number of tables in a FROM clause	225 or fewer, depending on the complexity of the statement
Maximum number of subqueries in a statement	224
Maximum total length of host and indicator variables pointed to in an SQLDA	32767 bytes 2 147 483 647 bytes (2 GB - 1 byte) for a LOB, subject to the limitations that are imposed by the application environment and host language

Table 135. DB2 limits on SQL statements (continued)

Item	Limit
Longest host variable used for insert or update operation	32704 bytes for a non-LOB 2 147 483 647 bytes (2 GB - 1 byte) for a LOB, subject to the limitations that are imposed by the application environment and host language
Longest SQL statement	2 097 152 bytes
Maximum number of elements in a select list	750 or fewer, depending on whether the select list is for the result table of static scrollable cursor ¹
Maximum number of predicates in a WHERE or HAVING clause	Limited by storage
Maximum total length of columns of a query operation requiring a sort key (SELECT DISTINCT, ORDER BY, GROUP BY, UNION, EXCEPT, and INTERSECT, without the ALL keyword, and the DISTINCT keyword for aggregate functions)	4000 bytes
Maximum total length of columns of a query operation requiring sort and evaluating column functions (MULTIPLE DISTINCT and GROUP BY)	32600 bytes
Maximum length of a sort key	16000 bytes
Maximum length of a check constraint	3800 bytes
Maximum number of bytes that can be passed in a single parameter of an SQL CALL statement	32765 bytes for a non-LOB 2 147 483 647 bytes (2 GB - 1 byte) for a LOB, subject to the limitations imposed by the application environment and host language
Maximum number of stored procedures, triggers, and user-defined functions that an SQL statement can implicitly or explicitly reference	16 nesting levels
Maximum length of the SQL path	2048 bytes
Maximum length of a WLM environment name in a CREATE PROCEDURE, CREATE FUNCTION, ALTER PROCEDURE, or ALTER FUNCTION statement.	32 bytes
Maximum number of XPath level in the XMLPATTERN clause of the CREATE INDEX statement.	50 nesting levels

Note:

1. If the scrollable cursor is read-only, the maximum number is 749 less the number of columns in the ORDER BY that are not in the select list. If the scrollable cursor is not read-only, the maximum number is 747.

The following table shows the DB2 system limits.

Table 136. DB2 system limits

Item	Limit
Maximum number of concurrent DB2 or application agents	Limited by the EDM pool size, buffer pool size, and the amount of storage that is used by each DB2 or application agent
Largest non-LOB table or table space	128 terabytes (TB)
Largest simple or segmented table space	64 GB
Largest log space	2 ⁴⁸ bytes

Table 136. DB2 system limits (continued)

Item	Limit
Largest active log data set	4 GB -1 byte
Largest archive log data set	4 GB -1 byte
Maximum number of active log copies	2
Maximum number of archive log copies	2
Maximum number of active log data sets (each copy)	93
Maximum number of archive log volumes (each copy)	10000
Maximum number of databases accessible to an application or end user	Limited by system storage and EDM pool size
Maximum number of databases	65271
Maximum number of implicitly created databases	10000
Maximum number of indexes on declared global temporary tables	10000
Largest EDM pool	The installation parameter maximum depends on available space
Maximum number of rows per page	255 for all table spaces except catalog and directory tables spaces, which have a maximum of 127
Maximum simple or segmented data set size	2 GB
Maximum partitioned data set size	See item “maximum size of a partition” in Table 135 on page 1591
Maximum LOB data set size	64 GB
Maximum number of table spaces that can be defined in a work file database	500
Maximum number of tables and triggers that can be defined in a work file database	11767

Reserved schema names and reserved words

Restrictions exist on the use of certain names that are used by DB2. In some cases, names are reserved and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs though not prevented by the database manager.

Reserved schema names

In general, for distinct types, user-defined functions, stored procedures, sequences, and triggers, schema names that begin with the prefix SYS are reserved. The schema name for these objects cannot begin with SYS except for certain exceptions.

The schema name for distinct types, user-defined functions, stored procedures, sequences, and triggers cannot begin with SYS with these exceptions:

- SYSADM. The schema name can be SYSADM for all these objects.
- SYSIBM. The schema name can be SYSIBM for procedures.
- SYSPROC. The schema name can be SYSPROC for procedures.
- SYSTOOLS. The schema name can be SYSTOOLS for distinct types, user-defined functions, procedures, and triggers if the user who executes the CREATE statement has the SYSADM or SYSCTRL privilege.
- SYSFUN. The schema name can be SYSFUN for external user-defined scalar functions or external user-defined table functions if the user who executes the CREATE statement has the SYSADM or SYSCTRL privilege.

Recommendation: Do not to use SESSION name as a schema name.

Reserved words

Certain words cannot be used as ordinary identifiers in some contexts because those words might be interpreted as SQL keywords. For example, ALL cannot be a column name in a SELECT statement. Each word, however, can be used as a delimited identifier in contexts where it otherwise cannot be used as an ordinary identifier. For example, if the quotation mark (") is the escape character that begins and ends delimited identifiers, "ALL" can appear as a column name in a SELECT statement.

New reserved words for this version of DB2 for z/OS are identified with notes in this topic. In addition, some topics in this information might indicate words that cannot be used in the specific context that is being described.

IBM SQL has additional reserved words that DB2 for z/OS does not enforce. Therefore, you should not use these additional reserved words as ordinary identifiers in names that have a continuing use. See *IBM DB2 SQL Reference for Cross-Platform Development* for a list of the words.

GUPI

ADD	ALTER	ASSOCIATE
AFTER	AND	ASUTIME
ALL	ANY	AT ¹
ALLOCATE	AS	AUDIT
ALLOW	ASENSITIVE	AUX
		AUXILIARY

BEFORE
BEGIN
BETWEEN
BUFFERPOOL
BY

CALL	CLUSTER	CONTAINS
CAPTURE	COLLECTION	CONTENT
CASCADED	COLLID	CONTINUE
CASE	COLUMN	CREATE
CAST	COMMENT	CURRENT
CCSID	COMMIT	CURRENT_DATE
CHAR	CONCAT	CURRENT_LC_CTYPE
CHARACTER	CONDITION	CURRENT_PATH
CHECK	CONNECT	CURRENT_SCHEMA ¹
CLONE ¹	CONNECTION	CURRENT_TIME
CLOSE	CONSTRAINT	CURRENT_TIMESTAMP
		CURSOR

DATA	DELETE	DO
DATABASE	DESCRIPTOR	DOCUMENT ¹
DAY	DETERMINISTIC	DOUBLE
DAYS	DISABLE	DROP
DBINFO	DISALLOW	DSSIZE
DECLARE	DISTINCT	DYNAMIC
DEFAULT		

EDITPROC ELSE ELSEIF ENCODING ENCRYPTION END	ENDING END-EXEC ² ERASE ESCAPE EXCEPT EXCEPTION	EXECUTE EXISTS EXIT EXPLAIN EXTERNAL
FENCED FETCH FIELDPROC FINAL FOR	FREE FROM FULL FUNCTION	
GENERATED GET GLOBAL GO GOTO	GRANT GROUP	
HANDLER HAVING HOLD HOUR HOURS		
IF IMMEDIATE IN INCLUSIVE INDEX	INF ¹ INFINITY ¹ INHERIT INNER INOUT INSENSITIVE	INSERT INTERSECT ¹ INTO IS ISOBID ITERATE
JAR JOIN		
KEEP ¹ KEY		
LABEL LANGUAGE LC_CTYPE LEAVE LEFT	LIKE LOCAL LOCALE LOCATOR LOCATORS	LOCK LOCKMAX LOCKSIZE LONG LOOP
MAINTAINED MATERIALIZED MICROSECOND MICROSECONDS MINUTE	MINUTES MODIFIES MONTH MONTHS	

	NAN ¹ NEXTVAL NO NONE NOT	NULL NULLS NUMPARTS	
	OBID OF ON OPEN OPTIMIZATION	OPTIMIZE OR ORDER OUT OUTER	
	PACKAGE PARAMETER PART PADDED PARTITION PARTITIONED PARTITIONING	PATH PIECESIZE PLAN PRECISION PREPARE PREVVAL	PRIQTY PRIVILEGES PROCEDURE PROGRAM PSID PUBLIC ¹
	QUERY QUERYNO		
	READS REFERENCES REFRESH RESIGNAL RELEASE RENAME REPEAT RESTRICT RESULT	RESULT_SET_LOCATOR RETURN RETURNS REVOKE RIGHT ROLE ¹ ROLLBACK ROUND_CEILING ¹	ROUND_DOWN ¹ ROUND_FLOOR ¹ ROUND_HALF_DOWN ¹ ROUND_HALF_EVEN ¹ ROUND_HALF_UP ¹ ROUND_UP ¹ ROW ¹ ROWSET RUN
	SAVEPOINT SCHEMA SCRATCHPAD SECOND SECONDS SECQTY SECURITY SEQUENCE SELECT SENSITIVE SESSION_USER ¹	SET SIGNAL SIMPLE SNAN ¹ SOME SOURCE SPECIFIC STANDARD STATIC STATEMENT ¹ STAY	STOGROUP STORES STYLE SUMMARY SYNONYM SYSFUN SYSIBM SYSPROC SYSTEM
	TABLE TABLESPACE THEN TO TRIGGER	TRUNCATE ¹ TYPE ¹	

|
|
|

UNDO UNION UNIQUE UNTIL UPDATE	USER USING
VALIDPROC VALUE VALUES VARIABLE VARIANT	VCAT VIEW VOLATILE VOLUMES
WHEN WHENEVER WHERE WHILE WITH WLM	
XMLELEMENT XMLEXISTS ¹ XMLNAMESPACES ¹ XMLCAST ¹	
YEAR YEARS	
Note: 1. New reserved word for Version 9.1. 2. COBOL only	



Characteristics of SQL statements in DB2 for z/OS

DB2 allows specific actions on each SQL statement, and only certain SQL statements are allowed in external routines and SQL procedures.

Actions allowed on SQL statements

Specific DB2 statements can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler.

The following table shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler. The letter **Y** means *yes*.

Table 137. Actions allowed on SQL statements in DB2 for z/OS

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
ALLOCATE CURSOR ¹	Y	Y	Y		
ALTER ²	Y	Y		Y	
ASSOCIATE LOCATORS ¹	Y	Y	Y		
BEGIN DECLARE SECTION					Y
CALL ¹	Y			Y	
CLOSE	Y			Y	
COMMENT	Y	Y		Y	
COMMIT ⁸	Y	Y		Y	
CONNECT	Y		Y		
CREATE ²	Y	Y		Y	
DECLARE CURSOR					Y
	Y	Y		Y	
DECLARE GLOBAL TEMPORARY TABLE					
DECLARE STATEMENT					Y
DECLARE TABLE					Y
DECLARE VARIABLE					Y
DELETE	Y	Y		Y	
DESCRIBE prepared statement or table	Y			Y	
DESCRIBE CURSOR	Y		Y		
DESCRIBE INPUT	Y			Y	
DESCRIBE PROCEDURE	Y		Y		
DROP ²	Y	Y		Y	
END DECLARE SECTION					Y
EXECUTE	Y			Y	
EXECUTE IMMEDIATE	Y			Y	
EXPLAIN	Y	Y		Y	
FETCH	Y			Y	
FREE LOCATOR ¹	Y	Y		Y	
GET DIAGNOSTICS	Y			Y	
GRANT ²	Y	Y		Y	
HOLD LOCATOR ¹	Y	Y		Y	

Table 137. Actions allowed on SQL statements in DB2 for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
INCLUDE					Y
INSERT	Y	Y		Y	
LABEL	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
MERGE	Y	Y		Y	
OPEN	Y			Y	
PREPARE	Y			Y ⁴	
REFRESH TABLE	Y	Y		Y	
RELEASE connection	Y		Y		
RELEASE SAVEPOINT	Y	Y		Y	
RENAME ²	Y	Y		Y	
REVOKE ²	Y	Y		Y	
ROLLBACK ⁸	Y	Y		Y	
SAVEPOINT	Y	Y		Y	
SELECT INTO	Y			Y	
SET CONNECTION	Y		Y		
SET CURRENT APPLICATION ENCODING SCHEME	Y		Y		
SET CURRENT DEBUG MODE	Y	Y		Y	
SET CURRENT DECFLOAT ROUNDING MODE	Y	Y		Y	
SET CURRENT DEGREE	Y	Y		Y	
SET CURRENT LC_CTYPE	Y	Y		Y	
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Y	Y		Y	
SET CURRENT OPTIMIZATION HINT	Y	Y		Y	
SET CURRENT PACKAGE PATH	Y		Y		
SET CURRENT PACKAGESET	Y			Y	
SET CURRENT PRECISION	Y	Y		Y	
SET CURRENT REFRESH AGE	Y	Y		Y	
SET CURRENT ROUTINE VERSION	Y	Y		Y	
SET CURRENT RULES	Y	Y		Y	
SET CURRENT SQLID ⁵	Y	Y		Y	
SET <i>host-variable</i> = CURRENT APPLICATION ENCODING SCHEME	Y	Y	Y		

Table 137. Actions allowed on SQL statements in DB2 for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
SET <i>host-variable</i> = CURRENT DATE	Y			Y	
SET <i>host-variable</i> = CURRENT DEGREE	Y			Y	
SET <i>host-variable</i> = CURRENT MEMBER	Y			Y	
SET <i>host-variable</i> = CURRENT PACKAGESET	Y		Y		
SET <i>host-variable</i> = CURRENT PATH	Y			Y	
SET <i>host-variable</i> = CURRENT QUERY OPTIMIZATION LEVEL	Y			Y	
SET <i>host-variable</i> = CURRENT SERVER	Y		Y		
SET <i>host-variable</i> = CURRENT SQLID	Y			Y	
SET <i>host-variable</i> = CURRENT TIME	Y			Y	
SET <i>host-variable</i> = CURRENT TIMESTAMP	Y			Y	
SET <i>host-variable</i> = CURRENT TIMEZONE	Y			Y	
SET PATH	Y	Y		Y	
SET SCHEMA	Y	Y		Y	
SET <i>transition-variable</i> = CURRENT DATE	Y			Y	
SET <i>transition-variable</i> = CURRENT DEGREE	Y			Y	
SET <i>transition-variable</i> = CURRENT PATH	Y			Y	
SET <i>transition-variable</i> = CURRENT QUERY OPTIMIZATION LEVEL	Y			Y	
SET <i>transition-variable</i> = CURRENT SQLID	Y			Y	
SET <i>transition-variable</i> = CURRENT TIME	Y			Y	
SET <i>transition-variable</i> = CURRENT TIMESTAMP	Y			Y	
SET <i>transition-variable</i> = CURRENT TIMEZONE	Y			Y	
SIGNAL ⁶	Y			Y	
I TRUNCATE	Y	Y		Y	
UPDATE	Y	Y		Y	

Table 137. Actions allowed on SQL statements in DB2 for z/OS (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
VALUES ⁶	Y			Y	
VALUES INTO ⁷	Y			Y	
WHENEVER					Y

Note:

1. The statement can be dynamically prepared. It cannot be issued dynamically.
2. The statement can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.
3. The statement can be dynamically prepared, but only from an ODBC or CLI driver that supports dynamic CALL statements.
4. The requesting system processes the PREPARE statement when the statement being prepared is ALLOCATE CURSOR or ASSOCIATE LOCATORS.
5. The value to which special register CURRENT SQLID is set is used as the SQL authorization ID for dynamic SQL statements only when DYNAMICRULES run behavior is in effect. The CURRENT SQLID value is ignored for the other DYNAMICRULES behaviors.
6. This statement can be used only in the triggered action of a trigger.
7. Local special registers can be referenced in a VALUES INTO statement if it results in the assignment of a single host-variable, not if it results in setting more than one value.
8. Some processing also occurs at the requester.

SQL statements allowed in external functions and stored procedures

Certain SQL statements can be executed in an external stored procedure or in an external user-defined function. Whether the statements can be executed depends on the level of SQL data access with which the stored procedure or external function is defined.

The following table shows which SQL statements in an external stored procedure or in an external user-defined function can execute. The letter Y means *yes*.

In general, if an executable SQL statement is encountered in a stored procedure or function defined as NO SQL, SQLSTATE 38001 is returned. If the routine is defined to allow some level of SQL access, SQL statements that are not supported in any context return SQLSTATE 38003. SQL statements not allowed for routines defined as CONTAINS SQL return SQLSTATE 38004, and SQL statements not allowed for READS SQL DATA return SQL STATE 38002.

Table 138. SQL statements in external user-defined functions and stored procedures

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALLOCATE CURSOR			Y	Y
ALTER				Y
ASSOCIATE LOCATORS			Y	Y
BEGIN DECLARE SECTION	Y ¹	Y	Y	Y
CALL		Y ²	Y ²	Y ²
CLOSE			Y	Y
COMMENT				Y
COMMIT ³		Y	Y	Y
CONNECT		Y	Y	Y
CREATE				Y
DECLARE CURSOR	Y ¹	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE				Y
DECLARE STATEMENT	Y ¹	Y	Y	Y
DECLARE TABLE	Y ¹	Y	Y	Y
DECLARE VARIABLE	Y ¹	Y	Y	Y
DELETE				Y
DESCRIBE			Y	Y
DESCRIBE CURSOR			Y	Y
DESCRIBE INPUT			Y	Y
DESCRIBE PROCEDURE			Y	Y
DROP				Y
END DECLARE SECTION	Y ¹	Y	Y	Y
EXECUTE		Y ⁴	Y ⁴	Y
EXECUTE IMMEDIATE		Y ⁴	Y ⁴	Y

Table 138. SQL statements in external user-defined functions and stored procedures (continued)

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
EXPLAIN				Y
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GET DIAGNOSTICS		Y	Y	Y
GRANT				Y
HOLD LOCATOR		Y	Y	Y
INCLUDE	Y ¹	Y	Y	Y
INSERT				Y
LABEL				Y
LOCK TABLE		Y	Y	Y
MERGE				Y
OPEN			Y	Y
PREPARE		Y	Y	Y
REFRESH TABLE				Y
RELEASE connection		Y	Y	Y
RELEASE SAVEPOINT ⁶				Y
REVOKE				Y
ROLLBACK ^{6, 7, 8}		Y	Y	Y
ROLLBACK TO SAVEPOINT ^{6, 7, 8}				Y
SAVEPOINT ⁶				Y
SELECT INTO			Y	Y
SET CONNECTION		Y	Y	Y
SET CURRENT DEBUG MODE		Y	Y	Y
SET CURRENT ROUTINE VERSION		Y	Y	Y
SET host-variable Assignment		Y ⁵	Y	Y
SET special register		Y	Y	Y
SET transition-variable Assignment		Y ⁵	Y	Y
SIGNAL		Y	Y	Y
TRUNCATE				Y
UPDATE				Y
VALUES			Y	Y
VALUES INTO		Y ⁵	Y	Y
WHENEVER	Y ¹	Y	Y	Y

Table 138. SQL statements in external user-defined functions and stored procedures (continued)

SQL statement	Level of SQL access			
	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
Notes:				
1. Although the SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.				
2. The stored procedure that is called must have the same or more restrictive level of SQL data access than the current level in effect. For example, a routine defined as MODIFIES SQL DATA can call a stored procedure defined as MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL. A routine defined as CONTAINS SQL can call a procedure defined as CONTAINS SQL or NO SQL.				
3. The COMMIT statement cannot be executed in a user-defined function. The COMMIT statement cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.				
4. The statement specified for the EXECUTE statement must be a statement that is allowed for the particular level of SQL data access in effect. For example, if the level in effect is READS SQL DATA, the statement must not be an INSERT, UPDATE, MERGE, or DELETE statement.				
5. The statement is supported only if it does not contain a subquery or query-expression.				
6. RELEASE SAVEPOINT, SAVEPOINT, and ROLLBACK (with the TO SAVEPOINT clause) cannot be executed from a user-defined function.				
7. If the ROLLBACK statement (without the TO SAVEPOINT clause) is executed in a user-defined function, an error is returned to the calling program, and the application is placed in a <i>must rollback</i> state.				
8. The ROLLBACK statement (without the TO SAVEPOINT clause) cannot be executed in a stored procedure if the procedure is in the calling chain of a user-defined function or trigger.				

SQL statements allowed in SQL procedures

Certain SQL statements that are valid in an SQL procedure body in addition to SQL procedure statements. Some SQL statements can be used as the only statement in the SQL procedure. An SQL statement can be executed in an SQL procedure depending on whether MODIFIES SQL DATA, CONTAINS SQL, or READS SQL DATA is specified in the stored procedure definition.

The following table lists the statements that are valid in an SQL procedure body, in addition to SQL procedure statements. The table lists the statements that can be used as the only statement in the SQL procedure and as the statements that can be nested in a compound statement. An SQL statement can be executed in an SQL procedure depending on whether MODIFIES SQL DATA, CONTAINS SQL, or READS SQL DATA is specified in the stored procedure definition. See Table 138 on page 1605 for a list of SQL statements that can be executed for each of these parameter values.

Table 139. Valid SQL statements in an SQL procedure body

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
ALLOCATE CURSOR		Y
ALTER DATABASE	Y	Y
ALTER FUNCTION	Y	Y
ALTER INDEX	Y	Y
ALTER PROCEDURE	Y	Y
ALTER SEQUENCE	Y	Y
ALTER STOGROUP	Y	Y
ALTER TABLE	Y	Y
ALTER TABLESPACE	Y	Y
ALTER TRUSTED CONTEXT	Y	Y
ALTER VIEW	Y	Y
ASSOCIATE LOCATORS		Y
CALL		Y
CLOSE		Y
COMMENT	Y	Y
COMMIT ¹	Y	Y
CONNECT	Y	Y
CREATE ALIAS	Y	Y
CREATE DATABASE	Y	Y
CREATE TYPE	Y	Y
CREATE FUNCTION ²	Y	Y
CREATE GLOBAL TEMPORARY TABLE	Y	Y
CREATE INDEX	Y	Y
CREATE PROCEDURE ²	Y	Y
CREATE ROLE	Y	Y
CREATE SEQUENCE	Y	Y

Table 139. Valid SQL statements in an SQL procedure body (continued)

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
CREATE STOGROUP	Y	Y
CREATE SYNONYM	Y	Y
CREATE TABLE	Y	Y
CREATE TABLESPACE	Y	Y
CREATE TRUSTED CONTEXT	Y	Y
CREATE VIEW	Y	Y
DECLARE CURSOR		Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y
DELETE	Y	Y
DROP INDEX	Y	Y
DROP ROLE	Y	Y
DROP TABLE	Y	Y
DROP TRUSTED CONTEXT	Y	Y
DROP VIEW	Y	Y
EXCHANGE	Y	Y
EXECUTE		Y
EXECUTE IMMEDIATE	Y	Y
FETCH ³		Y
GET DIAGNOSTICS	Y	Y
GRANT	Y	Y
INSERT	Y	Y
LABEL	Y	Y
LOCK TABLE	Y	Y
MERGE	Y	Y
OPEN		Y
PREPARE		Y
REFRESH TABLE	Y	Y
RELEASE connection	Y	Y
RELEASE SAVEPOINT	Y	Y
RENAME	Y	Y
REVOKE	Y	Y
ROLLBACK ¹	Y	Y
SAVEPOINT	Y	Y
SELECT INTO	Y	Y
SET CONNECTION	Y	Y
SET CURRENT DEBUG MODE	Y	Y
SET CURRENT DECFLOAT ROUNDING MODE	Y	Y

Table 139. Valid SQL statements in an SQL procedure body (continued)

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
SET CURRENT ROUTINE VERSION	Y	Y
SET ENCRYPTION PASSWORD	Y	Y
SET PATH	Y	Y
SET SCHEMA	Y	Y
TRUNCATE	Y	Y
UPDATE	Y	Y
VALUES INTO	Y	Y

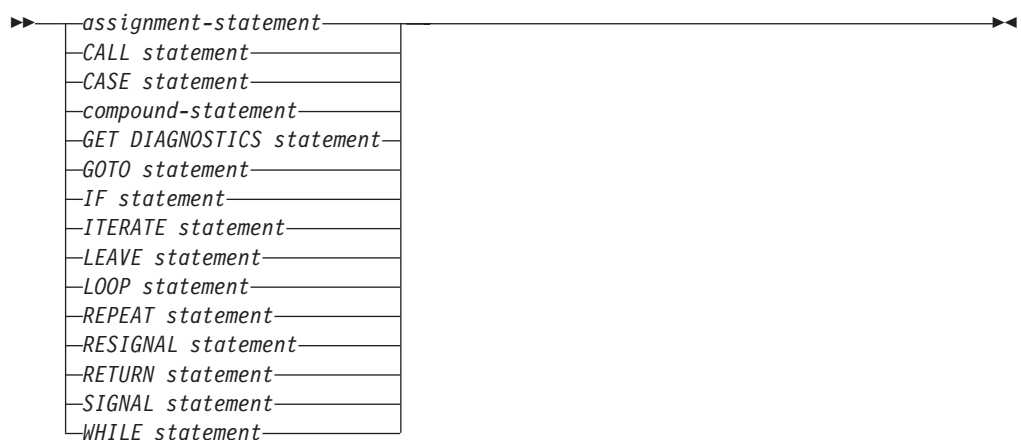
Notes:

1. The COMMIT statement and the ROLLBACK statement (without the TO SAVEPOINT clause) cannot be executed in a procedure if the procedure is in the calling chain of a user-defined function or trigger
2. CREATE FUNCTION with LANGUAGE SQL (specified either implicitly or explicitly) and CREATE PROCEDURE with LANGUAGE SQL are not allowed within the body of an SQL procedure.
3. The FETCH statement cannot specify the WITH CONTINUE or CURRENT CONTINUE clauses within an SQL procedure body.

SQL control statements for external SQL procedures

SQL control statements for external SQL procedures can be used only with SQL procedures that are created with the `FENCED` or `EXTERNAL` clause. *SQL control statements* provide the capability to control the logic flow, declare and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements.

SQL-control-statement:



Control statements are supported in SQL procedures. External SQL procedures are created by specifying either FENCED or EXTERNAL, LANGUAGE SQL, and an SQL routine body on the “CREATE PROCEDURE (SQL - external)” on page 1035 statement. The SQL routine body must be a single SQL statement which may be an SQL control statement.

The remainder of this chapter contains a description of the control statements that are supported for external SQL procedures, and includes syntax diagrams, semantic descriptions, usage notes, and examples of the use of the statements that constitute the SQL routine body. In addition, you can find information about referencing SQL parameters and variables in “References to SQL parameters and SQL variables.”

The two common elements that are used in describing specific SQL control statements are:

- SQL control statements as described above
- “SQL-procedure-statement” on page 1612

References to SQL parameters and SQL variables

SQL parameters and SQL variables can be referenced anywhere in the statement where an expression or a host variable can be specified. Host variables can be specified in SQL routines. SQL parameters and SQL variables can be referenced anywhere in the compound statement in which they are declared and can be qualified with the label name that is specified at the beginning of the compound statement.

All SQL parameters and SQL variables are considered nullable. The name of an SQL parameter or SQL variable in an SQL routine can be the same as the name of a column in a table or view that the SQL routine references. Names that are the same should be explicitly qualified. Qualifying a name clearly indicates whether the name refers to a column, SQL variable, or SQL parameter.

If the name is not qualified, the following rules describe whether the name refers to the column, the SQL variable, or the SQL parameter:

- The name is checked first as an SQL variable name and then as an SQL parameter name.
- If an SQL variable or SQL parameter by that name is not found, the name is assumed to be a column name.

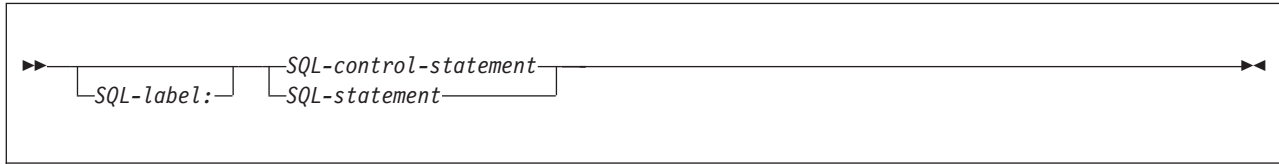
The name of an SQL variable or SQL parameter in an SQL routine can be the name of an identifier that is used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to the identifier, the SQL variable, or the SQL parameter:

- In the SET PATH and SET SCHEMA statements, the name is checked as an SQL variable name or an SQL parameter name. If an SQL variable or SQL parameter by that name is not found, the name is assumed to be an identifier.
- In the ASSOCIATE LOCATORS, CONNECT statement, the SET CONNECTION statement, and the RELEASE (connection) statement the name is used as an identifier.

SQL-procedure-statement

An SQL control statement may allow multiple SQL statements to be specified within the SQL control statement. These statements are defined as SQL procedure statements.

Syntax



Description

SQL-label

Specifies a label for the statement. *SQL-label* must not be a delimited identifier that includes lowercase letters or special characters. The label must be unique within the procedure.

SQL-control-statement

Specifies an SQL statement that provides the capability to control logic flow, declare and set variables, and handle warnings and exceptions, as defined in this section. Control statements are supported in SQL procedures.

SQL-statement

Specifies an SQL statement as listed in Table 139 on page 1608. These statements are described in Chapter 5, "Statements," on page 683.

Notes

Comments: Comments can be included within the body of an SQL procedure. In addition to the double-dash form of comments (--), a comment can begin with /* and end with */. The following rules apply to this form of comment:

- The beginning characters /* must be adjacent and on the same line.
- The ending characters */ must be adjacent and on the same line.
- Comments can be started wherever a space is valid.
- Comments can be continued to the next line.

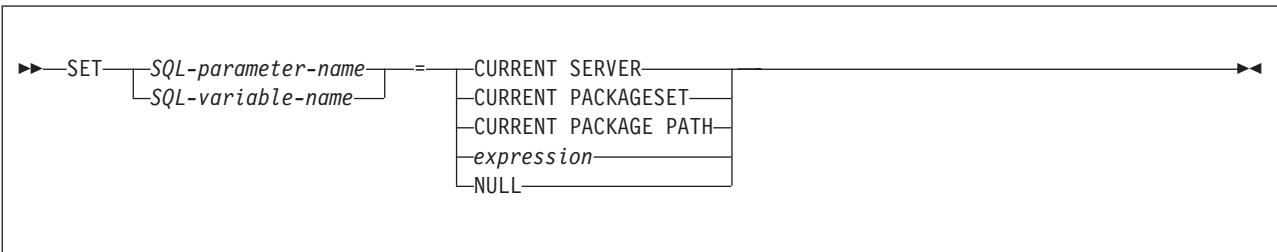
Handling errors and warnings: Conditions can be detected within an SQL procedure by using the following methods:

- Test the special SQL variables SQLSTATE and SQLCODE.
- Issue a GET DIAGNOSTICS statement to request the condition information. See "GET DIAGNOSTICS" on page 1317.
- Define condition handlers to detect and process conditions. See "compound-statement" on page 1620 for information about defining condition handlers.

assignment-statement (SQL control statements for external routines)

The assignment statement assigns a value to an SQL parameter or to an SQL variable.

Syntax



Description

SQL-parameter-name

Identifies the parameter that is the assignment target. The parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE statement and must be defined as OUT or INOUT.

SQL-variable-name

Identifies the SQL variable that is the assignment target. SQL variables can be declared in a compound-statement and must be declared before it is used. For information on declaring SQL variables, see “compound-statement” on page 1620.

expression or NULL

Specifies the expression or value that is the assignment source. The expression can be any expression of the type described in “Expressions” on page 180 except it cannot contain a reference to local special registers (CURRENT SERVER, CURRENT PACKAGESET, or CURRENT PACKAGE PATH).

Notes

Assignment rules: Assignment statements in SQL procedures must conform to the SQL assignment rules. For example, the data type of the target and source must be compatible. See “Assignment and comparison” on page 102 for assignment rules.

When a string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte or double-byte blanks. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the value is truncated and a warning is returned.

The ENCODING bind option is not used during processing of assignments to string variables. For example, assume that the system does not use mixed or DBCS, and the system EBCDIC SBCS CCSID is 37. Character conversion will not occur on assignment even if CCSID 500 is specified for the ENCODING bind parameter for the package for the procedure.

If truncation of the whole part of a number occurs on assignment to a numeric variable, the value is truncated and a warning is returned.

Assignments involving SQL parameters:

- An IN parameter can appear on the left side of an assignment statement. When control returns to the caller, the original value of an IN parameter is passed to the caller.
- An OUT parameter can appear on the left or right side of an assignment statement. When control returns to the caller, the last value that is assigned to an OUT parameter is returned to the caller.
- An INOUT parameter can appear on the left or right side of an assignment statement. The first value of the parameter is determined by the caller, and the last value that is assigned to the parameter is returned to the caller.
- A LOB parameter can not be used as an output value in an SQL statement in an SQL procedure when connected to a remote site. To circumvent the restriction, use a LOB SQL variable instead of a LOB parameter.

Considerations for SQLSTATE and SQLCODE SQL variables: Assignment to these variables is not prohibited. However, it is not recommended as assignment does not affect the diagnostic area or result in the activation of condition handlers. Furthermore, processing an assignment to these SQL variables causes the specified values for the assignment to be overlayed with the SQL return codes returned from executing the statement that does the assignment.

Examples

Increase the SQL variable *p_salary* by 10 percent.

```
SET p_salary = p_salary + (p_salary * .10)
```

Set SQL variable *p_salary* to the null value.

```
SET p_salary = NULL
```

Set SQL variable *midinit* to the first character of SQL variable *midname*.

```
SET midinit = SUBSTR(midname,1,1)
```


CALL statement

The CALL statement invokes a stored procedure.

Syntax

»»—CALL—*procedure-name*—*argument-list*—««

argument-list:



Description

procedure-name

Identifies the stored procedure to call. The procedure name must identify a stored procedure that exists at the current server.

argument-list

Identifies a list of values to be passed as parameters to the stored procedure. The number of parameters must be the same as the number of parameters defined for the stored procedure. See “CALL” on page 874 for more information.

Control is passed to the stored procedure according to the calling conventions for SQL procedures. When execution of the stored procedure is complete, the value of each parameter of the stored procedure is assigned to the corresponding parameter of the CALL statement defined as OUT or INOUT.

SQL-variable-name

Specifies an SQL variable as an argument to the stored procedure. For an explanation of references to SQL variables, see “References to SQL parameters and SQL variables” on page 1611.

SQL-parameter-name

Specifies an SQL parameter as an argument to the stored procedure. For an explanation of references to SQL parameters, see “References to SQL parameters and SQL variables” on page 1611.

expression

The parameter is the result of the specified *expression*, which is evaluated before the stored procedure is invoked. If *expression* is a single *SQL-parameter-name* or *SQL-variable-name*, the corresponding parameter of the procedure can be defined as IN, INOUT, or OUT. Otherwise, the corresponding parameter of the procedure must be defined as IN. If the result of the *expression* can be the null value, either the description of the

procedure must allow for null parameters or the corresponding parameter of the stored procedure must be defined as OUT.

The following additional rules apply depending on how the corresponding parameter was defined in the CREATE PROCEDURE statement for the procedure:

- IN *expression* can contain references to multiple SQL parameters or variables. In addition to the rules stated in “Expressions” on page 180 for *expression*, *expression* cannot include a column name, an aggregate function, or a user-defined function that is sourced on an aggregate function.
- INOUT or OUT *expression* can only be a single SQL parameter or variable.

NULL

The parameter is a null value. The corresponding parameter of the procedure must be defined as IN and the description of the procedure must allow for null parameters.

Notes

See “CALL” on page 874 for more information on the SQL CALL statement.

Examples

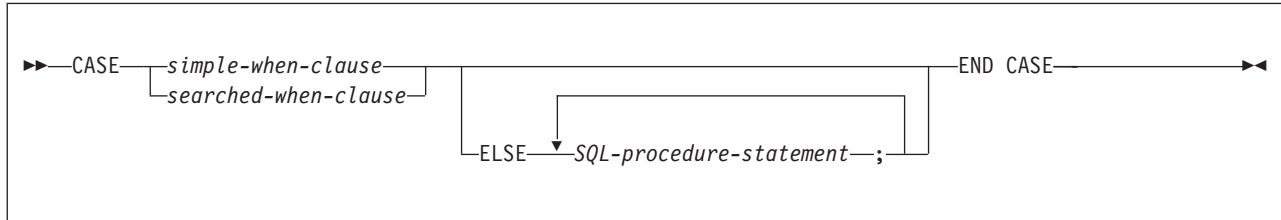
Call stored procedure proc1 and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

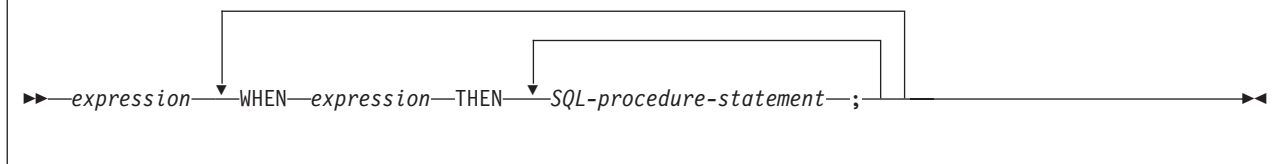
CASE statement

The CASE statement selects an execution path based on the evaluation of one or more conditions. A CASE statement operates in the same way as a CASE expression.

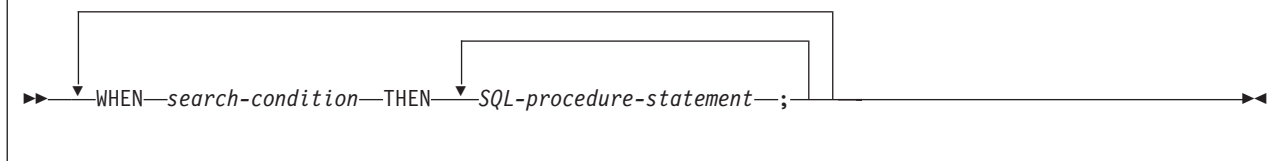
Syntax



simple-when-clause:



searched-when-clause:



Description

CASE

Begins a *case-expression*.

simple-when-clause

Specifies the *expression* prior to the first **WHEN** keyword that is tested for equality with the value of each *expression* that follows the **WHEN** keyword, and the result to be executed when those expressions are equal. If the comparison is true, the **THEN** statement is executed. If the result is unknown or false, processing continues to the next expression or the **ELSE** statement.

The data type of the *expression* prior to the first **WHEN** keyword must be comparable to the data types of each *expression* that follows the **WHEN** keywords.

searched-when-clause

Specifies the *search-condition* that is applied to each row or group of table data presented for evaluation, and the result when that condition is true.

search-condition cannot contain a *fullselect*. If the search condition is true, the

THEN statement is executed. If the condition is unknown or false, processing continues to the next search condition or the ELSE statement.

SQL-procedure-statement

Specifies a statement that follows the THEN and ELSE keyword. The statement specifies the result of a *searched-when-clause* or a *simple-when-clause* that is true, or the result if no case is true. The statement must be one of the statements listed under “SQL-procedure-statement” on page 1612.

search-condition

Specifies a condition that is true, false, or unknown about a row or group of table data.

ELSE *SQL-procedure-statement*

If none of the conditions specified in the *simple-when-clause* or *searched-when-clause* are true, the statements in the *else-clause* are executed.

If none of the conditions specified in the WHEN clause are true and an ELSE clause is not specified, an error is returned at run time, and the execution of the CASE statement is terminated.

END CASE

Ends a *case-statement*.

Notes

If none of the conditions specified in the WHEN clause are true and an ELSE clause is not specified, an error is returned at run time, and the execution of the CASE statement is terminated.

CASE statements that use a simple case statement WHEN clause can be nested up to three levels. CASE statements that use a searched statement WHEN clause have no limit to the number of nesting levels.

Considerations for the SQLSTATE and SQLCODE SQL variables: When the first SQL-procedure-statement in the CASE statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the expression or search conditions of that CASE statement. If a CASE statement does not include an ELSE clause and none of the search conditions evaluate to true, an error is returned.

Examples

Example 1: Use a simple case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

```
CASE v_workdept
  WHEN 'A00'
    THEN UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 1';
  WHEN 'B01'
    THEN UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 2';
  ELSE UPDATE DEPT SET
      DEPTNAME = 'DATA ACCESS 3';
END CASE
```

Example 2: Use a searched case statement WHEN clause to update column DEPTNAME in table DEPT, depending on the value of SQL variable *v_workdept*.

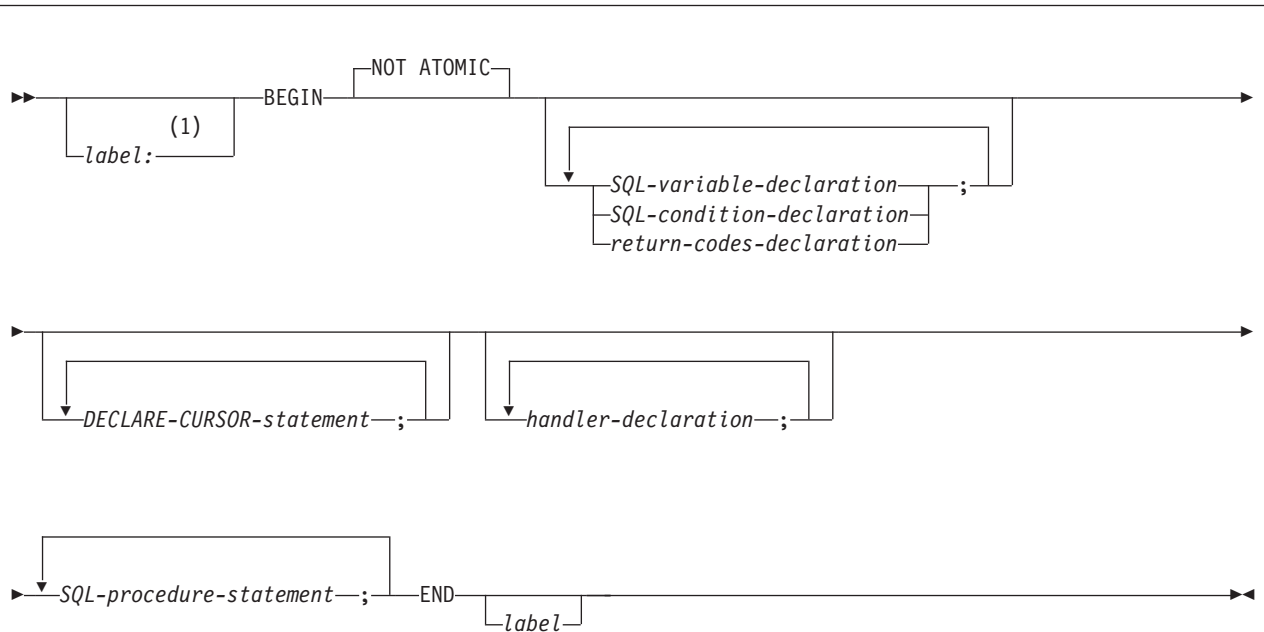
```
CASE
  WHEN v_workdept < 'B01'
    THEN UPDATE DEPT SET
```

```
|      DEPTNAME = 'DATA ACCESS 1';  
|      WHEN v_workdept < 'C01'  
|      THEN UPDATE DEPT SET  
|      DEPTNAME = 'DATA ACCESS 2';  
|      ELSE UPDATE DEPT SET  
|      DEPTNAME = 'DATA ACCESS 3';  
|      END CASE  
  
|
```

compound-statement

A compound statement contains a group of statements and declarations for SQL variables, cursors, and condition handlers.

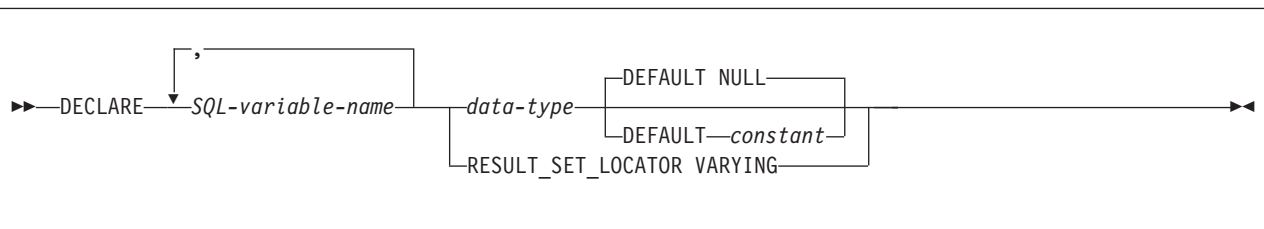
Syntax



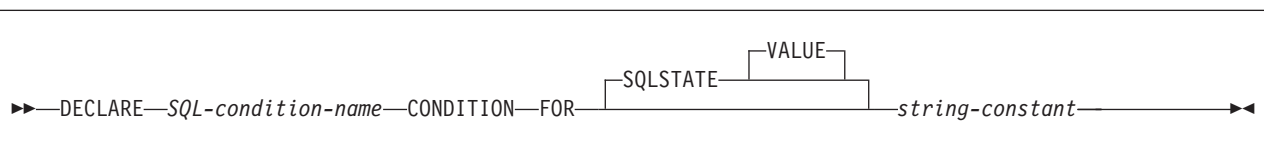
Notes:

- 1 Only one *label*: can be specified for each *SQL-procedure-statement*. If an ending label is specified for this beginning label, the labels must be the same.

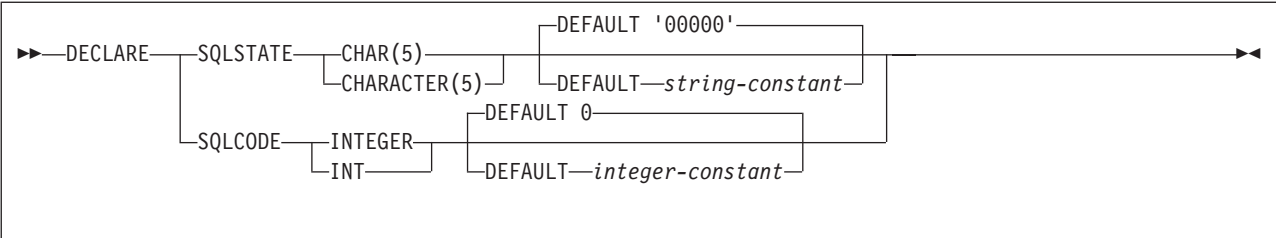
SQL-variable-declaration:



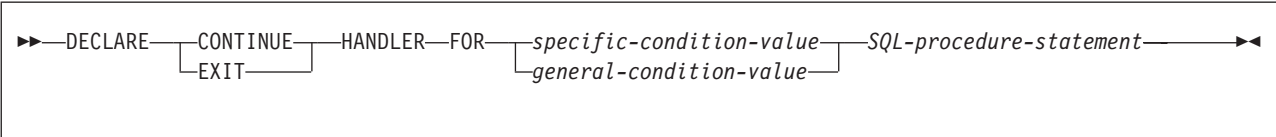
SQL-condition-declaration:



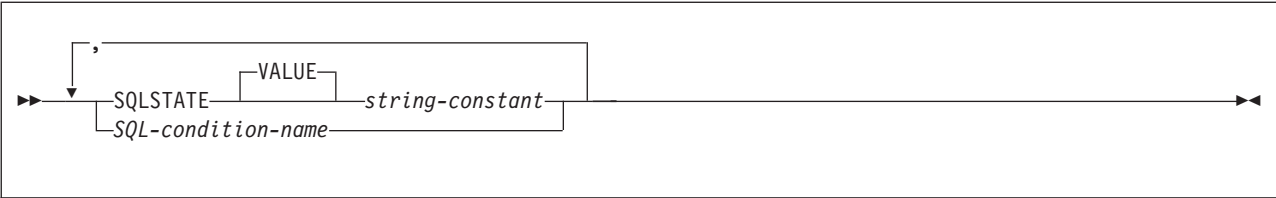
return-codes-declaration:



handler-declaration:



specific-condition-value:



general-condition-value:



Description

label

Defines the label for the code block. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound statement and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label.

NOT ATOMIC

NOT ATOMIC indicates that an error within the compound statement does not cause the compound statement to be rolled back.

SQL-variable-declaration

Declares a variable that is local to the compound statement.

SQL-variable-name

A qualified or unqualified name that designates a variable in an SQL procedure body. The unqualified form of *SQL-variable-name* is an SQL identifier and must not be a delimited identifier that contains lowercase letters or special characters. The qualified form is an SQL procedure statement label followed by a period (.) and an SQL identifier.

DB2 folds all SBCS SQL variable names to uppercase. SQL variable names should not be the same as column names. If an SQL statement contains an SQL variable or parameter and a column reference with the same name, DB2 interprets the name as an SQL variable or parameter name. To refer to the column, qualify the column name with the table name. Further, to avoid ambiguous variable references and to ensure compatibility with other DB2 platforms, qualify the SQL variable or parameter name with the label of the SQL procedure statement.

data-type

Specifies the data type and length of the variable. SQL variables follow the same rules for default lengths and maximum lengths as SQL procedure parameters. See “CREATE PROCEDURE (SQL - external)” on page 1035 for a description of SQL data types and lengths.

DEFAULT *constant* or NULL

Defines the default for the SQL variable. The variable is initialized when the SQL procedure is called. If a default value is not specified, the variable is initialized to NULL.

RESULT_SET_LOCATOR VARYING

Specifies the data type for a result set locator variable.

SQL-condition-declaration

Declares a condition name and corresponding SQLSTATE value.

SQL-condition-name

Specifies the name of the condition. The condition name is an SQL identifier and must not be a delimited identifier that includes lowercase letters or special characters. *SQL-condition-name* must be unique within the procedure body and can be referenced only within the compound statement in which it is declared.

FOR SQLSTATE *string-constant*

Specifies the SQLSTATE that is associated with the condition. The string must be specified as five characters enclosed in single quotes, and cannot be '00000'.

return-codes-declaration

Declares special variables called SQLSTATE and SQLCODE that are set automatically to the value returned after processing an SQL statement. Both the SQLSTATE and SQLCODE variables can be declared only in the outermost compound statement of the SQL procedure. Assignment to these variables is not prohibited; however, assignment is ignored by exception handlers, and processing the next SQL statement replaces the assigned value.

DECLARE-CURSOR-statement

Declares a cursor. Each cursor in the procedure body must have a unique name. An OPEN statement must be specified to open the cursor, and a FETCH statement can be specified to read rows. The cursor can be referenced only from within the compound statement. For more information on declaring a cursor, see “DECLARE CURSOR” on page 1191.

handler-declaration

Specifies a set of statements to execute when an exception or completion condition occurs in the compound statement. *SQL-procedure-statement* is the set of statements that execute when the handler receives control. See “SQL-procedure-statement” on page 1612 for information on *SQL-procedure-statement*.

A handler is active only within the compound statement in which it is declared.

The actions that a handler can perform are:

CONTINUE

Specifies that after the condition handler is activated and completes successfully, control is returned to the SQL statement that follows the statement that raised the condition. However, if the condition is an error condition and it was encountered while evaluating a search condition, as in a CASE, IF, REPEAT or WHILE statement, control returns to the statement that follows the corresponding END CASE, END IF, END REPEAT, or END WHILE.

EXIT

After the handler is invoked successfully, control is returned to the end of the compound statement.

The conditions that can cause the handler to gain control are:

SQLSTATE *string-constant*

Specifies an SQLSTATE for which the handler is invoked. The SQLSTATE cannot be '00000'.

SQL-condition-name

Specifies a condition name for which the handler is invoked. The condition name must be previously defined in a condition declaration.

SQLEXCEPTION

Specifies that the handler is invoked when an SQLEXCEPTION occurs. An SQLEXCEPTION is an SQLSTATE in which the class code is a value other than '00', '01', or '02'. For more information on SQLSTATE values, see *DB2 Codes*.

SQLWARNING

Specifies that the handler is invoked when an SQLWARNING occurs. An SQLWARNING is an SQLSTATE value with a class code of '01'.

NOT FOUND

Specifies that the handler is invoked when a NOT FOUND condition occurs. NOT FOUND corresponds to an SQLSTATE value with a class code of '02'.

Notes

The order of statements in a compound statement must be:

1. SQL variable, condition declarations, and return codes declarations
2. Cursor declarations
3. Handler declarations
4. SQL procedure statements

Compound statements cannot be nested.

Unlike host variables, SQL variables are not preceded by colons when they are used in SQL statements.

The following rules apply to handlers:

- A handler declaration that contains SQLEXCEPTION, SQLWARNING, or NOT FOUND cannot contain additional SQLSTATE or condition names.

- Handler declarations within the same compound statement cannot contain duplicate conditions.
- A handler declaration cannot contain the same condition code or SQLSTATE value more than once, and cannot contain an SQLSTATE value and a condition name that represent the same SQLSTATE value.
- A handler is activated when it is the most appropriate handler for an exception or completion condition.
- If there is no handler for an SQL error, the error is passed to the caller in the SQLCA.
- A handler cannot be activated by an assignment statement that assigns a value to SQLSTATE.

The following rules and recommendations apply to the SQLCODE and SQLSTATE SQL variables:

- A null value cannot be assigned to SQLSTATE or SQLCODE.
- The SQLSTATE and SQLCODE variable values should be saved immediately to temporary variables if there is any intention to use the values. If a handler exists for SQLSTATE, this assignment must be done as the first statement to be processed in the handler to avoid having the value replaced by the next SQL procedure statement. If the condition raised by the SQL statement is handled, the value is changed by the first SQL statement contained in the handler.

Considerations for the SQLSTATE and SQLCODE SQL variables: The compound statement itself does not affect the SQLSTATE and SQLCODE SQL variables. However, SQL statements contained within the compound statement can affect the SQLSTATE and SQLCODE SQL variables. At the end of the compound statement, the SQLSTATE and SQLCODE SQL variables reflect the result of the last SQL statement executed within the compound statement that caused a change to the SQLSTATE and SQLCODE SQL variables. If the SQLSTATE and SQLCODE SQL variables were not changed within the compound statement, they contain the same values as when the compound statement was entered.

Examples

Create a procedure body with a compound statement that performs the following actions:

- Declares SQL variables, a condition for SQLSTATE '02000', a handler for the condition, and a cursor
- Opens the cursor, fetches a row, and closes the cursor

```
CREATE PROCEDURE PROC1(OUT NOROWS INT) LANGUAGE SQL
BEGIN
  DECLARE v_firstnme VARCHAR(12);
  DECLARE v_midinit CHAR(1);
  DECLARE v_lastname VARCHAR(15);
  DECLARE v_edlevel SMALLINT;
  DECLARE v_salary DECIMAL(9,2);
  DECLARE at_end INT DEFAULT 0;
  DECLARE not_found
    CONDITION FOR '02000';
  DECLARE c1 CURSOR FOR
    SELECT FIRSTNME, MIDINIT, LASTNAME,
      EDLEVEL, SALARY
    FROM EMP;
  DECLARE CONTINUE HANDLER FOR not_found SET NOROWS=1;
```

```
|      OPEN c1;  
|      FETCH c1 INTO v_firstnme, v_midinit,  
|      v_lastname, v_edlevel, v_salary;  
|      END  
  
|
```

GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

See “GET DIAGNOSTICS” on page 1317.

When you need to specify a variable in a GET DIAGNOSTICS statement that is used within an SQL procedure, you would use either *SQL-variable-name* or *SQL-parameter-name*. In an embedded GET DIAGNOSTICS statement, you would use a *host-variable*. You can replace the instances of *host-variable* in the description of “GET DIAGNOSTICS” on page 1317 with *SQL-variable-name* or *SQL-parameter-name*.

GOTO statement

The GOTO statement is used to branch to a user-defined label within an SQL procedure.

Syntax

```
►► GOTO label ◄◄
```

Description

label

Specifies a labeled statement at which processing is to continue.

The labeled statement and the GOTO statement must be in the same scope. The following rules apply to the scope:

- If the GOTO statement is defined in a compound statement, *label* must be defined inside the same compound statement.
- If the GOTO statement is defined in a handler, *label* must be defined in the same handler and follow the other scope rules.
- If the GOTO statement is defined outside of a handler, *label* must not be defined within a handler.

If *label* is not defined within a scope that the GOTO statement can reach, an error is returned.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

Notes

Use the GOTO statement sparingly. Because the GOTO statement interferes with the normal sequence of processing, it makes an SQL procedure more difficult to read and maintain. Before using a GOTO statement, determine whether some other statement, such as an IF statement or LEAVE statement, can be used instead.

Examples

Use a GOTO statement to transfer control to the end of a compound statement if the value of an SQL variable is less than 600.

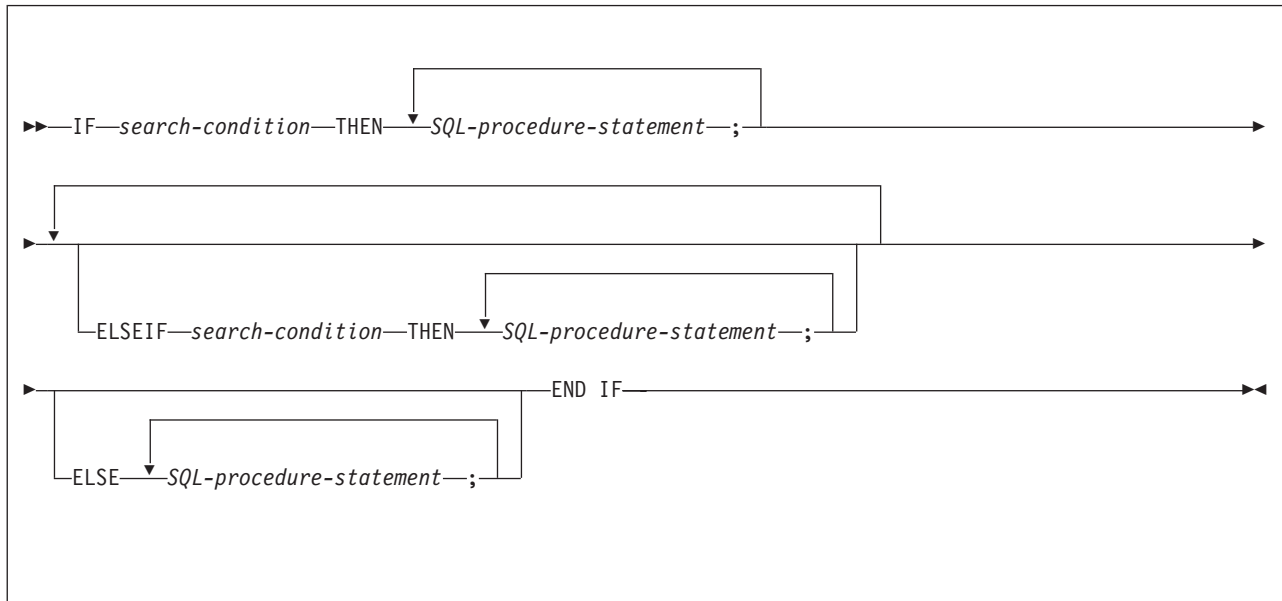
```
BEGIN
  DECLARE new_salary DECIMAL(9,2);
  DECLARE service DECIMAL(8,2);
  SELECT SALARY, CURRENT_DATE - HIREDATE
    INTO new_salary, service
  FROM EMP
 WHERE EMPNO = v_empno;
 IF service < 600
   THEN GOTO EXIT;
 END IF;
 IF rating = 1
   THEN SET new_salary =
    new_salary + (new_salary * .10);
 ELSEIF rating = 2
   THEN SET new_salary =
    new_salary + (new_salary * .05);
```

```
|      END IF;  
|      UPDATE EMP  
|      SET SALARY = new_salary  
|      WHERE EMPNO = v_empno;  
|      EXIT: SET return_parm = service;  
|      END  
  
|
```

IF statement

The IF statement selects an execution path based on the evaluation of a condition.

Syntax



Description

search-condition

Specifies the condition for which an SQL statement should be invoked. If the condition is unknown or false, processing continues to the next search condition until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement

Specifies the statement to be invoked if the preceding *search-condition* is true. If no *search-condition* evaluates to true, then the *SQL-procedure-statement* following the ELSE keyword is invoked. The statement must be one of the statements listed under “SQL-procedure-statement” on page 1612.

Notes

Considerations for the SQLSTATE and SQLCODE SQL variables: When the first SQL-procedure-statement in the IF statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search conditions of that IF statement. If an IF statement does not include an ELSE clause and none of the search conditions evaluate to true, then when the statement that follows that IF statement is executed, the SQLSTATE and SQLCODE SQL variables reflect the result of evaluating the search conditions of that IF statement.

Examples

Assign a value to the SQL variable *new_salary* based on the value of SQL variable *rating*.

```
IF rating = 1
  THEN SET new_salary =
    new_salary + (new_salary * .10);
ELSEIF rating = 2
```

```
|          THEN SET new_salary =  
|              new_salary + (new_salary * .05);  
|          ELSE SET new_salary =  
|              new_salary + (new_salary * .02);  
|      END IF
```


ITERATE statement

The ITERATE statement causes the flow of control to return to the beginning of a labeled loop.

Syntax

```
»—ITERATE—label—«
```

Description

label

Specifies the label of the LOOP, REPEAT, or WHILE statement to which the flow of control is passed.

Examples

This example uses a cursor to return information for a new department. If the `not_found` condition handler is invoked, the flow of control passes out of the loop. If the value of `v_dept` is 'D11', an ITERATE statement causes the flow of control to be passed back to the top of the LOOP statement. Otherwise, a new row is inserted into the table.

```
CREATE PROCEDURE ITERATOR ()
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
    DECLARE v_dept CHAR(3);
    DECLARE v_deptname VARCHAR(29);
    DECLARE v_admdept CHAR(3);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
        SELECT deptno,deptname,admrdept
        FROM department
        ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found
    SET at_end = 1;
    OPEN c1;
    ins_loop:
    LOOP
        FETCH c1 INTO v_dept, v_deptname, v_admdept;
        IF at_end = 1 THEN
            LEAVE ins_loop;
        ELSEIF v_dept = 'D11' THEN
            ITERATE ins_loop;
        END IF;
        INSERT INTO department (deptno,deptname,admrdept)
            VALUES('NEW', v_deptname, v_admdept);
    END LOOP;
    CLOSE c1;
END
```

LEAVE statement

The LEAVE statement transfers program control out of a loop or a compound statement.

Syntax

```
▶▶—LEAVE—label—▶▶
```

Description

label

Specifies the label of the compound statement or loop to exit.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

Notes

When a LEAVE statement transfers control out of a compound statement, all open cursors in the compound statement, except cursors that are used to return result sets, are closed.

Examples

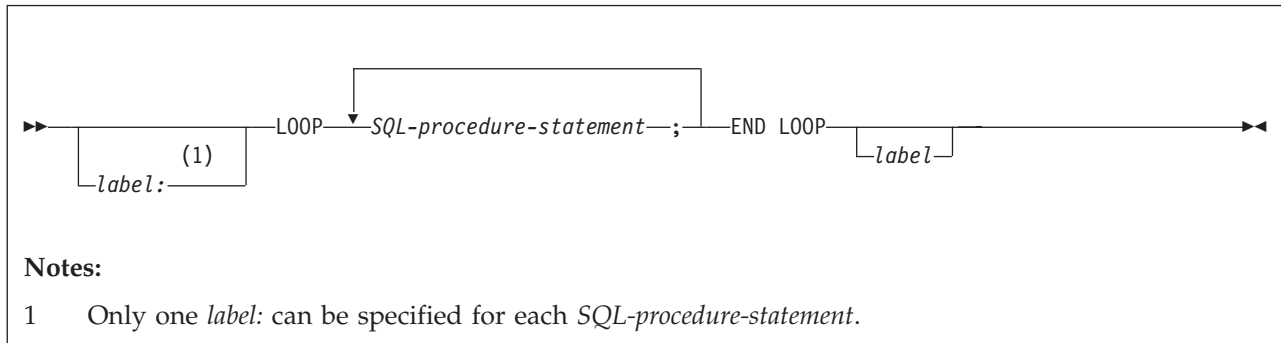
Use a LEAVE statement to transfer control out of a LOOP statement when a negative SQLCODE occurs.

```
ftch_loop: LOOP
  FETCH c1 INTO
    v_firstname, v_midinit,
    v_lastname, v_edlevel, v_salary;
  IF SQLCODE=100 THEN LEAVE ftch_loop;
END IF;
END LOOP
```

LOOP statement

The LOOP statement executes a statement or group of statements multiple times.

Syntax



Description

label

Specifies the label for the LOOP statement. If the ending label is specified, the beginning label must be specified, and the two must match.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

SQL-procedure-statement

Specifies the statements to be executed in the loop. The statement must be one of the statements listed under “SQL-procedure-statement” on page 1612.

Examples

This procedure uses a LOOP statement to fetch values from the employee table. Each time the loop iterates, the OUT parameter counter is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

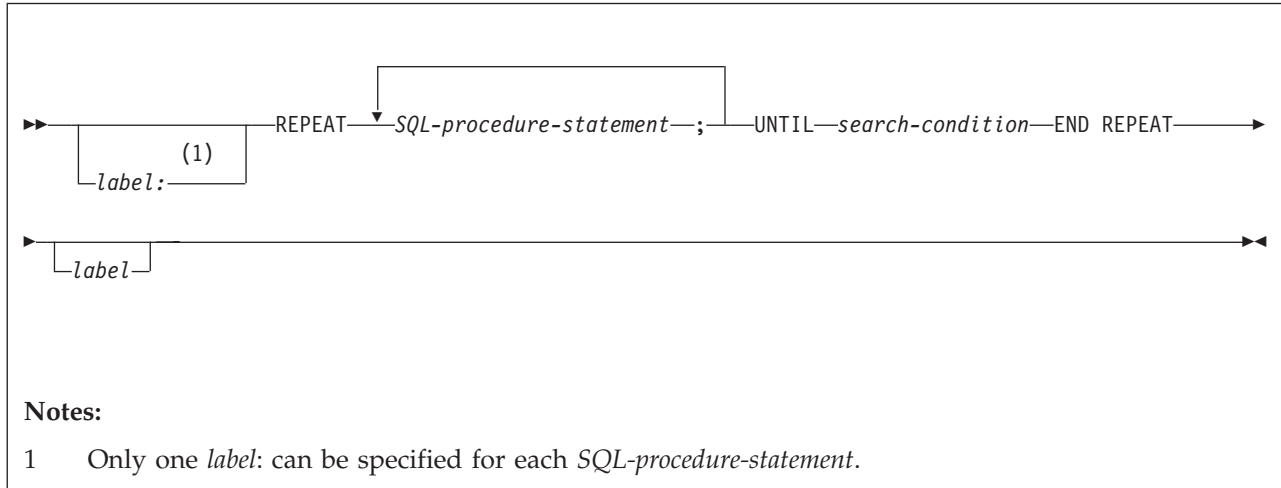
```
CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
LANGUAGE SQL
BEGIN
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE v_firstname VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE c1 CURSOR FOR
        SELECT firstname, midinit, lastname
        FROM employee;
    DECLARE EXIT HANDLER FOR NOT FOUND
        SET counter = -1;
    OPEN c1;
    fetch_loop:
    LOOP
        FETCH c1 INTO v_firstname, v_midinit, v_lastname;
        IF v_midinit = ' ' THEN
            LEAVE fetch_loop;
        END IF;
        SET v_counter = v_counter + 1;
```

```
|          END LOOP fetch_loop;  
|          SET counter = v_counter;  
|          CLOSE c1;  
|      END  
  
|
```

REPEAT statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Description

label

Specifies the label for the REPEAT statement. If the ending label is specified, the beginning label must be specified, and the two must match.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

SQL-procedure-statement

Specifies the statements to be executed. The statement must be one of the statements listed under “SQL-procedure-statement” on page 1612.

search-condition

Specifies a condition that is evaluated after each execution of the REPEAT statement. If the condition is true, the REPEAT loop will exit. If the condition is unknown or false, the REPEAT loop continues.

Examples

Use a REPEAT statement to fetch rows from a table.

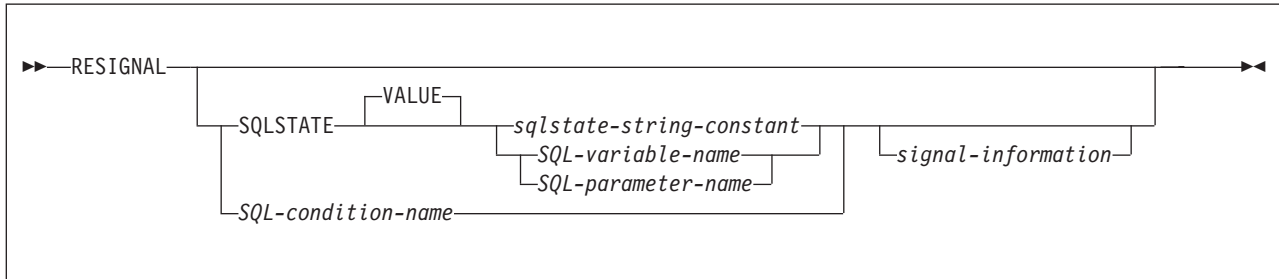
```
fetch_loop:
REPEAT
  FETCH c1 INTO
    v_firstname, v_midinit, v_lastname;
UNTIL
  SQLCODE <> 0
END REPEAT fetch_loop
```

RESIGNAL statement

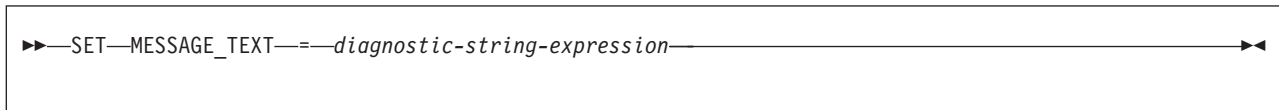
The RESIGNAL statement is used within a condition handler to re-raise the current condition, or to raise an alternate condition so that it can be processed at a higher level. It causes an exception, warning, or not found condition to be returned along with optional message text.

Issuing the RESIGNAL statement without an operand causes the current condition to be passed upwards.

Syntax



signal-information:



Description

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATE values:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').
- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant

A character string constant with a length of five bytes that is a valid SQLSTATE value.

SQL-variable-name or *SQL-parameter-name*

Specifies an SQL variable or SQL parameter that is defined for the procedure.

SQL-variable-name

Specifies an SQL variable that is declared within the *compound-statement* that contains the RESIGNAL statement. *SQL-variable-name* must be defined as CHAR or VARCHAR data type with a length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL_parameter-name

Specifies an SQL parameter that is defined for the procedure that contains the SQLSTATE value. The SQL parameter must be defined as

a CHAR or VARCHAR value and have a length of five bytes and must not be null. The SQL parameter must contain a valid SQLSTATE value.

SQL-condition-name

Specifies the name of the condition that will be returned. *condition-name* must be declared within the *compound-statement*.

SET MESSAGE_TEXT

Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression

An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see “GET DIAGNOSTICS” on page 1317.

Notes

While any valid SQLSTATE value can be used in the RESIGNAL statement, programmers should define new SQLSTATE values based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If the RESIGNAL statement is issued without an SQLSTATE clause or a *condition-name*, the RESIGNAL statement must be in a handler and the identical condition that activated the handler is returned. The SQLSTATE, SQLCODE, and the SQLCA associated with the condition are unchanged.

If an SQLSTATE clause or a *condition-name* was specified, the SQLCODE returned is based on the SQLSTATE value as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not-found message is returned, and the SQLCODE is set to +438.
- Otherwise, an exception is returned and the SQLCODE is set to -438.

The other fields of the SQLCA are set as follows:

- SQLERRDx fields are set to zero.
- SQLWARNx fields are set to blank.
- SQLERRMC is set to the first 70 bytes of MESSAGE_TEXT.
- SQLERRML is set to the length of SQLERRMC.
- SQLERRP is set to ROUTINE.

When the SQLSTATE or condition indicates that an exception is returned (an SQLSTATE class other than '01' or '02'), the exception is not handled, and control is immediately returned to the end of the compound statement.

When the SQLSTATE or condition indicates that a warning (SQLSTATE class '02') is returned, the warning is not handled, and processing continues with the next statement.

When the SQLSTATE or condition indicates that a not-found condition (SQLSTATE class '02') is returned, the not-found condition is not handled, and processing continues with the next statement.

Examples

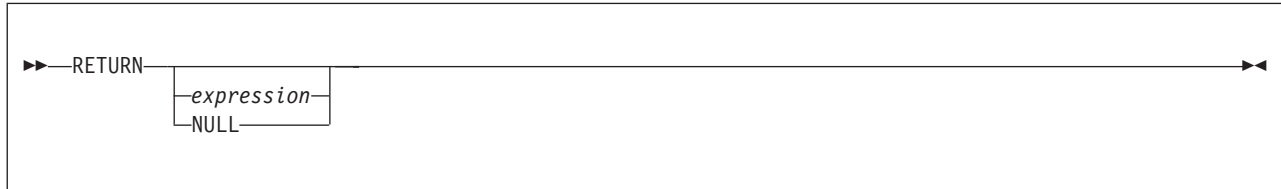
The following example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the overflow condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE to the client application.

```
CREATE PROCEDURE divide ( IN numerator INTEGER,
                        IN denominator INTEGER,
                        OUT divide_result INTEGER)
LANGUAGE SQL
CONTAINS SQL
BEGIN
    DECLARE overflow CONDITION for SQLSTATE '22003' ;
    DECLARE CONTINUE HANDLER FOR overflow
        RESIGNAL SQLSTATE '22375';
    IF denominator = 0 THEN
        SIGNAL overflow;
    ELSE
        SET divide_result = numerator / denominator;
    END IF;
END
```


RETURN statement

The RETURN statement is used to return from the routine. For SQL functions, it returns the result of the function. For an SQL procedure, it optionally returns an integer status value.

Syntax



Description

expression

Specifies a value that is returned from the routine.

- If the routine is a function, *expression* must be specified and the value of *expression* must conform to the SQL assignment rules as described in “Assignment and comparison” on page 102. If the value is being assigned to a string variable, storage assignment rules apply.
- If the routine is a procedure, the data type of *expression* must be INTEGER. If *expression* evaluates to the null value, a value of 0 is returned.

The *expression* cannot include a column name or a host variable. See “Expressions” on page 180 for information on expressions. The *expression* cannot contain a scalar fullselect.

NULL

The null value is returned from the SQL function. **NULL** is not allowed in SQL procedures.

Notes

When a RETURN statement is not used within an SQL procedure or when no value is specified: If a RETURN statement was not used to return from a procedure or if a value is not specified on the RETURN statement, one of the following values is set:

- If the procedure returns with an SQLCODE that is greater or equal to zero, the return status is set to a value of '0'.
- If the procedure returns with an SQLCODE that is less than zero, the return status is set to a value of '-1'.

When a RETURN statement is used within an SQL procedure: If a RETURN statement with a specified return value was used to return from a procedure, the SQLCODE, SQLSTATE, and message length in the SQLCA are initialized to zeros and the message text is set to blanks. An error is not returned to the caller.

When the value is returned: When a value is returned from a procedure, the caller may access the value using one of the following methods:

- The GET DIAGNOSTICS statement to retrieve the RETURN_STATUS when the SQL procedure was called from another SQL procedure.
- The parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI application.

- Directly from the SQLCA returned from processing the CALL of an SQL procedure by retrieving the value of sqlerrd[0]. When the SQLCODE is less than zero, the sqlerrd[0] value is not set. The application should assume a return status value of '-1'.

Examples

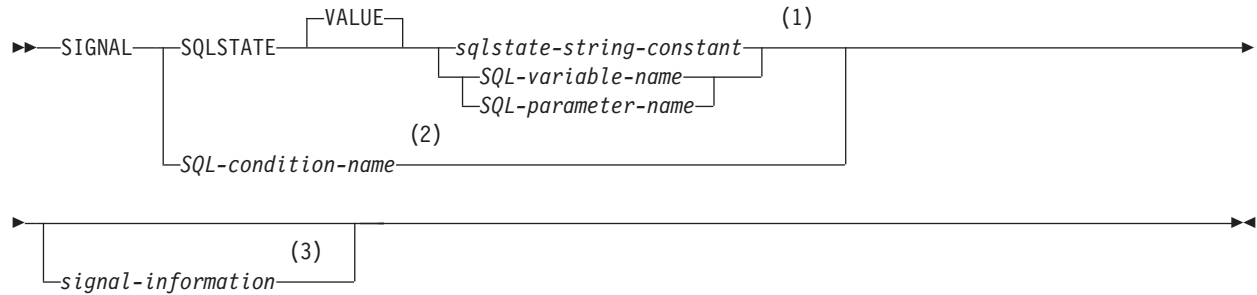
Use a RETURN statement to return from an SQL procedure with a status value of zero if successful or '-200' if not successful.

```
BEGIN
    . . .
    GOTO FAIL;
    . . .
SUCCESS: RETURN 0;
FAIL: RETURN -200;
END
```

SIGNAL statement

The SIGNAL statement is used to return an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

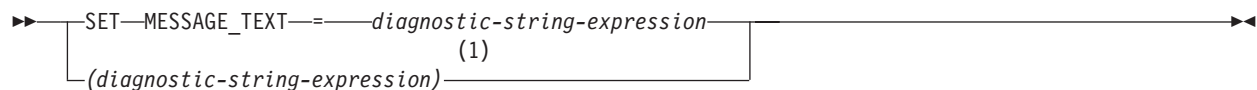
Syntax



Notes:

- 1 The **SQLSTATE** variation must be used within a trigger body.
- 2 **SQL-condition-name** must not be specified within a trigger body.
- 3 **signal-information** must be specified within a trigger body

signal-information:



Notes:

- 1 **(diagnostic-string-expression)** must only be specified within a trigger body.

Description

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. It must be a character string constant with exactly five characters that follow the rules for SQLSTATES:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letter ('A' through 'Z').
- The SQLSTATE class (the first two characters) cannot be '00' because it represents successful completion.

If the SQLSTATE does not conform to these rules, an error occurs.

sqlstate-string-constant

A character string constant with a length of five bytes that is a valid SQLSTATE value.

SQL-variable-name **or** *SQL-parameter-name*

Specifies an SQL variable or SQL parameter that contains a valid SQLSTATE value.

SQL-variable-name

Specifies an SQL variable that is declared within the *compound-statement*. *SQL-variable-name* must be defined as a CHAR or VARCHAR data type, have a length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-parameter-name

Specifies an SQL parameter that is defined for the procedure and contains the SQLSTATE value. The SQL parameter must be defined as a CHAR or VARCHAR value, have a length of five bytes, must not be null, and must contain a valid SQLSTATE value.

SQL-condition-name

Specifies the name of the condition that will be returned. *condition-name* must be declared within the *compound-statement*.

SET MESSAGE_TEXT

Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA or with the GET DIAGNOSTICS statement.

diagnostic-string-expression

An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1317.

(diagnostic-string-expression)

An expression with a data type of CHAR or VARCHAR that returns a character string of up to 1000 bytes that describes the error or warning condition. For information on how to obtain the complete message text, see "GET DIAGNOSTICS" on page 1317.

This syntax variation is only provided within the scope of a CREATE TRIGGER statement for compatibility with previous versions of DB2. To conform with the ANS and ISO standards, this form should not be used.

Notes

While any valid SQLSTATE value can be used in the SIGNAL statement, programmers should define new SQLSTATES based on ranges reserved for applications. This practice prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

If a SIGNAL statement is issued, the SQLCODE that is returned is based on the SQLSTATE as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not-found message is returned, and the SQLCODE is set to +438.
- Otherwise, an exception is returned and the SQLCODE is set to -438.

The other fields of the SQLCA are set as follows:

- SQLERRDx fields are set to zero.
- SQLWARNx fields are set to blank.
- SQLERRMC is set to the first 70 bytes of MESSAGE_TEXT.

- SQLERRML is set to the length of SQLERRMC.
- SQLERRP is set to ROUTINE.

When the SQLSTATE or condition indicates that an exception (an SQLSTATE class other than '01' or '02') is returned, one of the following actions occurs:

- If a handler exists for the specified SQLSTATE, condition, or SQLEXCEPTION, the exception is handled, and control is transferred to that handler.
- Otherwise, the exception is not handled, and control is immediately returned to the end of the compound statement.

When the SQLSTATE or condition indicates that a warning (SQLSTATE class '01') is returned, one of the following actions occurs:

- If an active handler exists for the specified SQLSTATE, condition, or SQLWARNING, the warning is handled, and control is transferred to that handler.
- Otherwise, the warning is not handled, and processing continues with the next statement.

When the SQLSTATE or condition indicates that a not-found condition (SQLSTATE class '02') is returned, one of the following actions occurs:

- If an active handler exists for the specified SQLSTATE, condition, or not-found condition, the not-found condition is handled, and control is transferred to that handler.
- Otherwise, the not-found condition is not handled, and processing continues with the next statement.

When the SIGNAL statement is issued in a handler, no active handler exists.

Using a SIGNAL statement in the body of a trigger: Within the triggered action of a CREATE TRIGGER statement, the message text can be specified using only these variations:

```
SIGNAL SQLSTATE sqlstate-string-constant
      SET MESSAGE_TEXT = diagnostic-string-expression
SIGNAL SQLSTATE sqlstate-string-constant
      (diagnostic-string-expression)
```

Examples

Example 1: The following example shows an SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
      (IN ONUM INTEGER, IN CNUM INTEGER,
       IN PNUM INTEGER, IN QNUM INTEGER)
LANGUAGE SQL
SPECIFIC SUBMIT_ORDER
MODIFIES SQL DATA
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
    SIGNAL SQLSTATE '75002'
      SET MESSAGE_TEXT = 'Customer number is not known';
  INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
    VALUES (ONUM, CNUM, PNUM, QNUM);
END
```

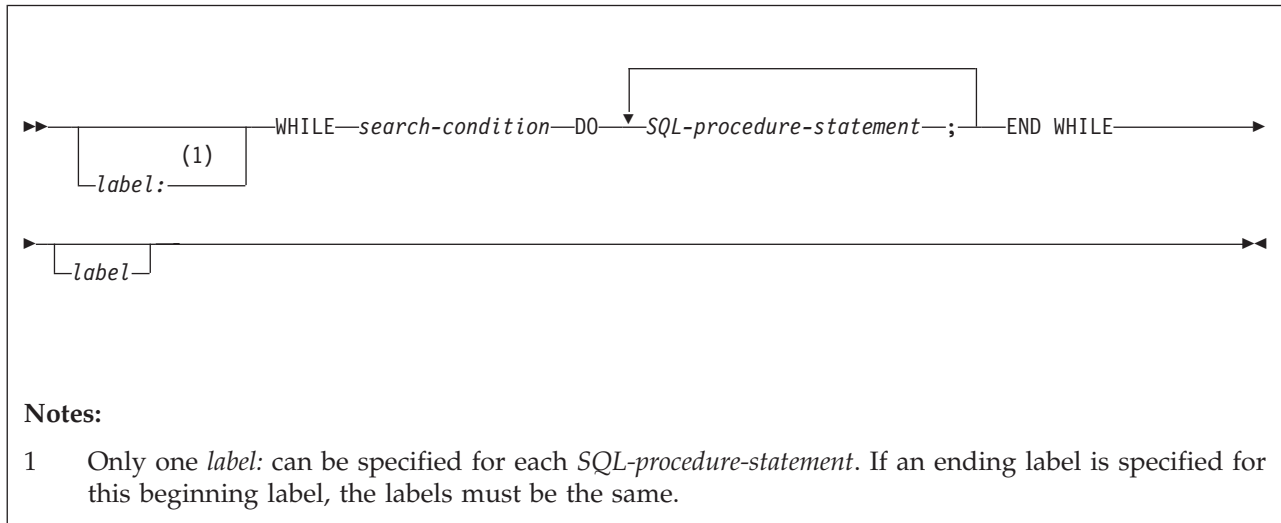
Example 2: The following example shows a trigger for an order system that allows orders to be recorded in an ORDERS table (ORDERNO, CUSTNO, PARTNO, QUANTITY) only if there is sufficient stock in the PARTS tables. When there is insufficient stock for an order, SQLSTATE '75001' is returned along with an appropriate error description.

```
CREATE TRIGGER CK_AVAIL
NO CASCADE BEFORE INSERT ON ORDERS
REFERENCING NEW AS NEW_ORDER
FOR EACH ROW MODE DB2SQL
WHEN (NEW_ORDER.QUANTITY > (SELECT ON_HAND FROM PARTS
                             WHERE NEW_ORDER.PARTNO = PARTS.PARTNO))
BEGIN ATOMIC
  SIGNAL SQLSTATE '75001' ('Insufficient stock for order');
END
```

WHILE statement

The WHILE statement repeats the execution of a statement or group of statements while a specified condition is true.

Syntax



Description

label

Specifies the label for the WHILE statement. If the ending label is specified, it must be the same as the beginning label.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

search-condition

Specifies a condition that is evaluated before each execution of the loop. If the condition is true, the SQL procedure statement in the loop is executed.

SQL-procedure-statement

Specifies the statements to be executed in the loop. The statement must be one of the statements listed under "SQL-procedure-statement" on page 1612.

Examples

Use a WHILE statement to fetch rows from a table while SQL variable *at_end*, which indicates whether the end of the table has been reached, is 0.

```
WHILE at_end = 0 DO
  FETCH c1 INTO
    v_firstname, v_midinit,
    v_lastname, v_edlevel, v_salary;
  IF SQLCODE=100 THEN SET at_end=1;
END IF;
END WHILE
```

SQL communication area (SQLCA)

An SQLCA is a structure or collection of variables that is updated after each SQL statement executes. An application program that contains executable SQL statements must provide exactly one SQLCA, with a few exceptions.

There are three exceptions:

- A program that is precompiled with the STDSQL(YES) option must not provide an SQLCA
- In some cases a Fortran program must provide more than one SQLCA.
- A Java program where an SQLCA is not applicable.

In all host languages except REXX (and Java where the SQLCA is not applicable), the SQL INCLUDE statement can be used to provide the declaration of the SQLCA.

In COBOL and assembler: The name of the storage area must be SQLCA.

In PL/I, and C: The name of the structure must be SQLCA. Every executable SQL statement must be within the scope of its declaration.

Unless noted otherwise, C is used to represent C/370™ and C/C++ programming languages.

In Fortran: The name of the COMMON area for the INTEGER variables of the SQLCA must be SQLCA1; the name of the COMMON area for the CHARACTER variables must be SQLCA2. An SQLCA definition is required for every subprogram that contains SQL statements. One is also needed for the main program if it contains SQL statements.

In REXX: DB2 generates the SQLCA automatically. A REXX procedure cannot use the INCLUDE statement. The REXX SQLCA has a somewhat different format from SQLCAs for the other languages. See “The REXX SQLCA” on page 1654 for more information on the REXX SQLCA.

Description of SQLCA fields

For the most part, COBOL, C, PL/I, and assembler use the same names for the SQLCA fields, and Fortran uses different names. However, there is one instance where C, PL/I, and assembler names differ from COBOL.

The names in the following table are those provided by the SQL INCLUDE statement.

Table 140. Fields of SQLCA

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLCAID	sqlcaid	Not used.	CHAR(8)	An "eye catcher" for storage dumps, containing the text 'SQLCA'. The sixth byte is 'L' if line number information is returned from parsing a dynamic statement or a native SQL procedure. The sixth byte is not set when processing an external SQL procedure.
SQLCABC	sqlcabc	Not used.	INTEGER	Contains the length of the SQLCA: 136.
SQLCODE (See note 1)	SQLCODE	SQLCOD	INTEGER	Contains the SQL return code. (See note 2)
				Code Means 0 Successful execution (though there might have been warning messages). positive Successful execution, but with a warning condition or other information. negative Error condition.
SQLERRML (See note 3)	sqlerrml (See note 3)	SQLTXL	SMALLINT	Length indicator for SQLERRMC, in the range 0 through 70. 0 means that the value of SQLERRMC is not pertinent.
SQLERRMC (See note 3)	sqlerrmc (See note 3)	SQLTXT	VARCHAR(70)	Contains one or more tokens, separated by X'FF', that are substituted for variables in the descriptions of error conditions. It may contain truncated tokens. A message length of 70 bytes indicates a possible truncation.
SQLERRP	sqlerrp	SQLERP	CHAR(8)	Provides a product signature and, in the case of an error, diagnostic information such as the name of the module that detected the error. In all cases, the first three characters are 'DSN' for DB2 for z/OS.
SQLERRD(1)	sqlerrd[0]	SQLERR(1)	INTEGER	<p>For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100).</p> <p>On successful return from an SQL procedure, contains the return status value from the SQL procedure.</p> <p>SQLERRD(1) can also contain an internal error code.</p>

Table 140. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLERRD(2)	sqlerrd[1]	SQLERR(2)	INTEGER	<p>For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100).</p> <p>SQLERRD(2) can also contain an internal error code.</p>

Table 140. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLERRD(3)	sqlerrd[2]	SQLERR(3)	INTEGER	<p>Contains the number of rows that qualified to be deleted, inserted, or updated after a DELETE, INSERT, UPDATE, or MERGE statement. The number excludes rows affected by either triggers or referential integrity constraints. For the OPEN of a cursor for a SELECT with a data change statement or for a SELECT INTO, SQLERRD(3) contains the number of rows affected by the embedded data change statement. The value is 0 if the SQL statement fails, indicating that all changes made in executing the statement canceled.</p> <p>For a DELETE statement the value will be -1 if the operation is a mass delete from a table in a segmented table space and the DELETE statement did not include selection criteria. If the delete was against a view, neither the DELETE statement nor the definition of the view included selection criteria.</p> <p>For a TRUNCATE statement, the value will be -1.</p> <p>For a PREPARE statement, contains the estimated number of rows selected. If the number of rows is greater than 2 147 483 647, a value of 2 147 483 647 is returned.</p> <p>For a REFRESH TABLE statement, SQLERRD(3) contains the number of rows inserted into the materialized query table.</p> <p>For a rowset-oriented FETCH, contains the number of rows fetched.</p> <p>For SQLCODES -911 and -913, SQLERRD(3) contains the reason code for the timeout or deadlock.</p> <p>When an error is encountered in parsing a dynamic statement, or when parsing, binding, or executing a native SQL procedure, SQLERRD(3) will contain the line number where the error was encountered. The sixth byte of SQLCAID must be 'L' for this to be a valid line number. This value will be meaningful only if the statement source contains new line control characters. This information is not returned for an external SQL procedure.</p>

Table 140. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLERRD(4)	sqlerrd[3]	SQLERR(4)	INTEGER	Generally, contains timerons, a short floating-point value that indicates a rough relative estimate of resources required (See note 4). It does not reflect an estimate of the time required. When preparing a dynamically defined SQL statement, you can use this field as an indicator of the relative cost of the prepared SQL statement. For a particular statement, this number can vary with changes to the statistics in the catalog. It is also subject to change between releases of DB2 for z/OS.
SQLERRD(5)	sqlerrd[4]	SQLERR(5)	INTEGER	Contains the position or column of a syntax error for a PREPARE or EXECUTE IMMEDIATE statement.
SQLERRD(6)	sqlerrd[5]	SQLERR(6)	INTEGER	Contains an internal error code.
SQLWARN0	SQLWARN0	SQLWRN(0)	CHAR(1)	Contains a blank if no other indicator is set to a warning condition (that is, no other indicator contains a W or Z). Contains a W if at least one other indicator contains a W or Z.
SQLWARN1	SQLWARN1	SQLWRN(1)	CHAR(1)	Contains a W if the value of a string column was truncated when assigned to a host variable. Contains an N for non-scrollable cursors and S for scrollable cursors after the OPEN CURSOR or ALLOCATE CURSOR statement. If subsystem parameter DISABSCL is set to YES, the field will not be set to N for non-scrollable cursors.
SQLWARN2	SQLWARN2	SQLWRN(2)	CHAR(1)	Contains a W if null values were eliminated from the argument of an aggregate function; not necessarily set to W for the MIN function because its results are not dependent on the elimination of null values.
SQLWARN3	SQLWARN3	SQLWRN(3)	CHAR(1)	Contains a W if the number of result columns is larger than the number of host variables. Contains a Z if fewer locators were provided in the ASSOCIATE LOCATORS statement than the stored procedure returned.
SQLWARN4	SQLWARN4	SQLWRN(4)	CHAR(1)	Contains a W if a prepared UPDATE or DELETE statement does not include a WHERE clause. For a scrollable cursor, contains a D for sensitive dynamic cursors, I for insensitive cursors, and S for sensitive static cursors after the OPEN CURSOR or ALLOCATE CURSOR statement; blank if cursor is not scrollable. If subsystem parameter DISABSCL is set to YES, the field will not be set to N for non-scrollable cursors.

Table 140. Fields of SQLCA (continued)

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLWARN5	SQLWARN5	SQLWRN(5)	CHAR(1)	Contains a W if the SQL statement was not executed because it is not a valid SQL statement in DB2 for z/OS. Contains a character value of 1 (read only), 2 (read and delete), or 4 (read, delete, and update) to reflect capability of the cursor after the OPEN CURSOR or ALLOCATE CURSOR statement. If subsystem parameter DISABSCL is set to YES, the field will not be set to N for non-scrollable cursors.
SQLWARN6	SQLWARN6	SQLWRN(6)	CHAR(1)	Contains a W if the addition of a month or year duration to a DATE or TIMESTAMP value results in an invalid day (for example, June 31). Indicates that the value of the day was changed to the last day of the month to make the result valid.
SQLWARN7	SQLWARN7	SQLWRN(7)	CHAR(1)	Contains a W if one or more nonzero digits were eliminated from the fractional part of a number used as the operand of a decimal multiply or divide operation.
SQLWARN8	SQLWARN8	SQLWRX(1)	CHAR(1)	Contains a W if a character that could not be converted was replaced with a substitute character. Contains a Y if there was an unsuccessful attempt to establish a trusted connection.
SQLWARN9	SQLWARN9	SQLWRX(2)	CHAR(1)	Contains a W if arithmetic exceptions were ignored during COUNT or COUNT_BIG processing. Contains a Z if the stored procedure returned multiple result sets.
SQLWARNA	SQLWARNA	SQLWRX(3)	CHAR(1)	Contains a W if at least one character field of the SQLCA or the SQLDA names or labels is invalid due to a character conversion error.
SQLSTATE	sqlstate	SQLSTT	CHAR(5)	Contains a return code for the outcome of the most recent execution of an SQL statement (See note 5).

Notes:

1. With the precompiler option STDSQL(YES) in effect, SQLCODE is replaced by SQLCADE in SQLCA.
2. For the specific meanings of SQL return codes, see *DB2 Codes*.
3. In COBOL, SQLERRM includes SQLERRML and SQLERRMC. In PL/I and C, the varying-length string SQLERRM is equivalent to SQLERRML prefixed to SQLERRMC. In assembler, the storage area SQLERRM is equivalent to SQLERRML and SQLERRMC. See the examples for the various host languages in "The included SQLCA" on page 1652.
4. The use of timerons may require special handling because they are floating-point values in an INTEGER array. In PL/I, for example, you could first copy the value into a BIN FIXED(31) based variable that coincides with a BIN FLOAT(24) variable.
5. For a description of SQLSTATE values, see *DB2 Codes*.

The included SQLCA

The description of the SQLCA that is given by INCLUDE SQLCA is shown for each of the host languages.

assembler:

```
SQLCA      DS    0F
SQLCAID    DS    CL8          ID
SQLCABC     DS    F          BYTE COUNT
SQLCODE     DS    F          RETURN CODE
SQLERRM     DS    H,CL70     ERR MSG PARMS
SQLERRP     DS    CL8          IMPL-DEPENDENT
SQLERRD     DS    6F
SQLWARN     DS    0C          WARNING FLAGS
SQLWARN0    DS    C'W' IF ANY
SQLWARN1    DS    C'W' = WARNING
SQLWARN2    DS    C'W' = WARNING
SQLWARN3    DS    C'W' = WARNING
SQLWARN4    DS    C'W' = WARNING
SQLWARN5    DS    C'W' = WARNING
SQLWARN6    DS    C'W' = WARNING
SQLWARN7    DS    C'W' = WARNING
SQLEXT     DS    0CL8
SQLWARN8    DS    C
SQLWARN9    DS    C
SQLWARNA    DS    C
SQLSTATE    DS    CL5
```

C:

```
#ifndef SQLCODE
struct sqlca
{
    unsigned char  sqlcaid[8];
    long          sqlcabc;
    long          sqlcode;
    short         sqlerrml;
    unsigned char  sqlerrmc[70];
    unsigned char  sqlerrp[8];
    long          sqlerrd[6];
    unsigned char  sqlwarn[11];
    unsigned char  sqlstate[5];
};
#define SQLCODE    sqlca.sqlcode
#define SQLWARN0    sqlca.sqlwarn[0]
#define SQLWARN1    sqlca.sqlwarn[1]
#define SQLWARN2    sqlca.sqlwarn[2]
#define SQLWARN3    sqlca.sqlwarn[3]
#define SQLWARN4    sqlca.sqlwarn[4]
#define SQLWARN5    sqlca.sqlwarn[5]
#define SQLWARN6    sqlca.sqlwarn[6]
#define SQLWARN7    sqlca.sqlwarn[7]
#define SQLWARN8    sqlca.sqlwarn[8]
#define SQLWARN9    sqlca.sqlwarn[9]
#define SQLWARNA    sqlca.sqlwarn[10]
#define SQLSTATE    sqlca.sqlstate
#endif
struct sqlca sqlca;
```

COBOL:

```
01 SQLCA.
   05 SQLCAID      PIC X(8).
   05 SQLCABC      PIC S9(9) COMP-4.
   05 SQLCODE      PIC S9(9) COMP-4.
   05 SQLERRM.
      49 SQLERRML  PIC S9(4) COMP-4.
```

```

      49 SQLERRMC  PIC X(70).
05  SQLERRP      PIC X(8).
05  SQLERRD      OCCURS 6 TIMES
                  PIC S9(9) COMP-4.

05  SQLWARN.
      10 SQLWARN0 PIC X.
      10 SQLWARN1 PIC X.
      10 SQLWARN2 PIC X.
      10 SQLWARN3 PIC X.
      10 SQLWARN4 PIC X.
      10 SQLWARN5 PIC X.
      10 SQLWARN6 PIC X.
      10 SQLWARN7 PIC X.
05  SQLEXT.
      10 SQLWARN8 PIC X.
      10 SQLWARN9 PIC X.
      10 SQLWARNA PIC X.
      10 SQLSTATE PIC X(5).

```

Fortran:

```

*
*   THE SQL COMMUNICATIONS AREA
*
      INTEGER      SQLCOD,
C      SQLERR(6),
C      SQLTXL*2
      COMMON /SQLCA1/SQLCOD, SQLERR,SQLTXL
      CHARACTER    SQLERP*8,
C      SQLWRN(0:7)*1,
C      SQLTXT*70,
C      SQLEXT*8,
C      SQLWRX(1:3)*1,
C      SQLSTT*5
      COMMON /SQLCA2/SQLERP,SQLWRN,SQLTXT,SQLWRX,
C      SQLSTT
      EQUIVALENCE (SQLWRX,SQLEXT)
*

```

PL/I:

```

DECLARE
1  SQLCA,
2  SQLCAID CHAR(8),
2  SQLCABC FIXED(31) BINARY,
2  SQLCODE FIXED(31) BINARY,
2  SQLERRM CHAR(70) VAR,
2  SQLERRP CHAR(8),
2  SQLERRD(6) FIXED(31) BINARY,
2  SQLWARN,
3  SQLWARN0 CHAR(1),
3  SQLWARN1 CHAR(1),
3  SQLWARN2 CHAR(1),
3  SQLWARN3 CHAR(1),
3  SQLWARN4 CHAR(1),
3  SQLWARN5 CHAR(1),
3  SQLWARN6 CHAR(1),
3  SQLWARN7 CHAR(1),
2  SQLEXT,
3  SQLWARN8 CHAR(1),
3  SQLWARN9 CHAR(1),
3  SQLWARNA CHAR(1),
3  SQLSTATE CHAR(5);

```

The REXX SQLCA

The REXX SQLCA consists of a set of variables, rather than a structure. DB2 makes the SQLCA available to your application automatically.

The following table lists the variables in a REXX SQLCA.

Table 141. Variables in a REXX SQLCA

Variable	Contents
SQLCODE	Contains the SQL return code.
SQLERRMC	Contains one or more tokens, separated by 'X'FF', that are substituted for variables in the descriptions of error conditions. It might contain truncated tokens. A message length of 70 bytes indicates a possible truncation.
SQLERRP	Provides a product signature and, in the case of an error, diagnostic information such as the name of the module that detected the error. For DB2 for z/OS, the product signature is 'DSN'.
SQLERRD.1	For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100). SQLERRD(1) can also contain an internal error code.
SQLERRD.2	For a sensitive static cursor, contains the number of rows in a result table when the cursor position is after the last row (that is, when SQLCODE is equal to +100). SQLERRD(2) can also contain an internal error code.
SQLERRD.3	Contains the number of rows that qualified for the operation after an SQL data change statement (but not rows deleted as a result of CASCADE delete). For the OPEN of a cursor for a SELECT with an SQL data change statement or for a SELECT INTO, SQLERRD(3) contains the number of rows affected by the embedded data change statement. Set to 0 if the SQL statement fails, indicating that all changes made in executing the statement were canceled. Set to -1 for a mass delete from a table in a segmented table space, for a truncate operation, or a delete from a view when neither the DELETE statement nor the definition of the view included selection criteria. For rowset-oriented FETCH statements, contains the number of rows returned in the rowset. For SQLCODES -911 and -913, SQLERRD(3) contains the reason code for the timeout or deadlock. After successful execution of the REFRESH TABLE statement, SQLERRD(3) contains the number of rows inserted into the materialized query table. When an error is encountered in parsing a dynamic statement, or when parsing, binding, or executing a native SQL procedure, SQLERRD(3) will contain the line number where the error was encountered. The sixth byte of SQLCAID must be 'L' for this to be a valid line number. This value will be meaningful only if the statement source contains new line control characters. This information is not returned for an external SQL procedure.
SQLERRD.4	Generally, contains timerons, a short floating-point value that indicates a rough relative estimate of resources required. This value does not reflect an estimate of the time required to execute the SQL statement. After you prepare an SQL statement, you can use this field as an indicator of the relative cost of the prepared SQL statement. For a particular statement, this number can vary with changes to the statistics in the catalog. This value is subject to change between releases of DB2 for z/OS.
SQLERRD.5	Contains the position or column of a syntax error for a PREPARE or EXECUTE IMMEDIATE statement.
SQLERRD.6	Contains an internal error code.

Table 141. Variables in a REXX SQLCA (continued)

Variable	Contents
SQLWARN.0	Contains a blank if no other indicator is set to a warning condition (that is, no other indicator contains a W or Z). Contains a W if at least one other indicator contains a W or Z.
SQLWARN.1	Contains a W if the value of a string column was truncated when assigned to a host variable. Contains an N for non-scrollable cursors and S for scrollable cursors after the OPEN CURSOR or ALLOCATE CURSOR statement.
SQLWARN.2	Contains a W if null values were eliminated from the argument of an aggregate function; not necessarily set to W for the MIN function because its results are not dependent on the elimination of null values.
SQLWARN.3	Contains a W if the number of result columns is larger than the number of host variables. Contains Z if the ASSOCIATE LOCATORS statement contains fewer locators than the stored procedure returned.
SQLWARN.4	Contains a W if a prepared UPDATE or DELETE statement does not include a WHERE clause. For a scrollable cursor, contains a D for sensitive dynamic cursors, I for insensitive cursors, and S for sensitive static cursors after the OPEN CURSOR or ALLOCATE CURSOR statement; otherwise, blank if cursor is not scrollable.
SQLWARN.5	Contains a W if the SQL statement was not executed because it is not a valid SQL statement in DB2 for z/OS. Contains a character value of 1 (read only), 2 (read and delete), or 4 (read, delete, and update) to reflect capability of the cursor after the OPEN CURSOR or ALLOCATE CURSOR statement.
SQLWARN.6	Contains a W if the addition of a month or year duration to a DATE or TIMESTAMP value results in an invalid day (for example, June 31). Indicates that the value of the day was changed to the last day of the month to make the result valid.
SQLWARN.7	Contains a W if one or more nonzero digits were eliminated from the fractional part of a number that was used as the operand of a decimal multiply or divide operation.
SQLWARN.8	Contains a W if a character that could not be converted was replaced with a substitute character.
SQLWARN.9	Contains a W if arithmetic exceptions were ignored during COUNT or COUNT_BIG processing. Contains a Z if the stored procedure returned multiple result sets.
SQLWARN.10	Contains a W if at least one character field of the SQLCA is invalid due to a character conversion error.
SQLSTATE	Contains a return code for the outcome of the most recent execution of an SQL statement.

SQL descriptor area (SQLDA)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement, and can be optionally used by the PREPARE, OPEN, FETCH, EXECUTE, and CALL statements. An SQLDA can be used in a DESCRIBE or PREPARE INTO statement, modified with the addresses of host variables, and then reused in a FETCH statement.

The meaning of the information in an SQLDA depends on the context in which it is used. For DESCRIBE and PREPARE INTO, DB2 sets the fields in the SQLDA to provide information to the application program. For OPEN, EXECUTE, FETCH, and CALL, the application program sets the fields in the SQLDA to provide DB2 with information:

DESCRIBE *statement-name* or PREPARE INTO

With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about a prepared statement. Each SQLVAR occurrence describes a column of the result table.

DESCRIBE TABLE

With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the columns of a table or view. Each SQLVAR occurrence describes a column of the specified table or view.

DESCRIBE CURSOR

With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the result set that is associated with the specified cursor. Each SQLVAR occurrence describes a column of the result set.

DESCRIBE INPUT

With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the input parameter markers of a prepared statement. Each SQLVAR occurrence describes an input parameter marker.

DESCRIBE PROCEDURE

With the exception of SQLN, DB2 sets fields of the SQLDA to provide information to an application program about the result sets returned by the specified stored procedure. Each SQLVAR occurrence describes a returned result set.

OPEN, EXECUTE, FETCH, and CALL

The application program sets fields of the SQLDA to provide information about host variables or output buffers in the application program to DB2. Each SQLVAR occurrence describes a host variable or output buffer.

- For OPEN and EXECUTE, each SQLVAR occurrence describes an input value that is substituted for a parameter marker in the associated SQL statement that was previously prepared.
- For FETCH, each SQLVAR occurrence describes a host variable or buffer in the application program that is to be used to contain an output value from a row of the result.
- For CALL, each SQLVAR occurrence describes a host variable that corresponds to a parameter in the parameter list for the stored procedure.

For information on the way to use the SQLDA, see *DB2 Application Programming and SQL Guide*.

Description of SQLDA fields

An SQLDA consists of four variables, a header, and an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR.

In DESCRIBE and PREPARE INTO, each occurrence of the SQLVAR describes the column of a table. In FETCH, OPEN, EXECUTE, and CALL, each occurrence describes a host variable.

The SQLDA Header

The fields in the SQLDA header have different usage depending on whether the SQLDA is being used in a DESCRIBE or PREPARE INTO statement or the SQLDA is being used in a FETCH, INSERT, OPEN, EXECUTE, or CALL statement.

The following table describes the fields in the SQLDA header.

Table 142. Fields of the SQLDA header

C name assembler, COBOL or PL/I name	Data type	Usage in DESCRIBE ¹ and PREPARE INTO	Usage in FETCH,INSERT, OPEN, EXECUTE, and CALL
sqldaid SQLDAID	CHAR(8)	<p>An “eye catcher” for storage dumps, containing the text 'SQLDA '.</p> <p>The 7th byte of the field is a flag that can be used to determine if more than one SQLVAR entry is needed for each column. For details, see “Determining how many SQLVAR occurrences are needed” on page 1661.</p> <p>For DESCRIBE CURSOR, the field is set to 'SQLRS'. If the cursor is declared WITH HOLD in a stored procedure, the high-order bit of the 8th byte is set to 1.</p> <p>For DESCRIBE PROCEDURE, it is set to 'SQLPR'.</p>	<p>A plus sign (+) in the 6th byte indicates that SQLNAME contains an overriding CCSID.</p> <p>A '2' in the 7th byte indicates the two SQLVAR entries were allocated for each column or parameter.</p> <p>A '3' in the 7th byte indicates that three SQLVAR entries were allocated for each column or parameter. Although three entries are never needed on input to DB2, an SQLDA with three entries might be used when the SQLDA was initialized by a DESCRIBE or PREPARE INTO with a USING BOTH clause.</p> <p>Otherwise, SQLDAID is not used.</p>
sqldabc SQLDABC	INTEGER	Length of the SQLDA, equal to SQLN _x * 44+16.	Length of the SQLDA, greater than or equal to SQLN _x * 44+16.

Table 142. Fields of the SQLDA header (continued)

C name assembler, COBOL or PL/I name	Data type	Usage in DESCRIBE ¹ and PREPARE INTO	Usage in FETCH,INSERT, OPEN, EXECUTE, and CALL
sqln SQLN	SMALLINT	<p>Unchanged by DB2. The field must be set to a value greater than or equal to zero before the statement is executed. The field indicates the total number of occurrences of SQLVAR. At the very least, the number should be:</p> <ul style="list-style-type: none"> For DESCRIBE INPUT, the number of parameter markers to be described. For other DESCRIBE or PREPARE INTO: the number of columns of the result, or a multiple of the columns of the result when there are multiple sets of SQLVAR entries because column labels are returned in addition to column names. 	Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero.
sqld SQLD	SMALLINT	<p>The number of columns described by occurrences of SQLVAR. Double that number if USING BOTH appears in the DESCRIBE or PREPARE INTO statement. Contains a 0 if the statement string is not a query.</p> <p>For DESCRIBE PROCEDURE, the number of result sets returned by the stored procedure. Contains a 0 if no result sets are returned.</p>	The number of host variables described by occurrences of SQLVAR.

Note:

- The third column of this table represents several forms of the DESCRIBE statement:
 - For DESCRIBE *output* and PREPARE INTO, the column pertains to columns of the result table.
 - For DESCRIBE CURSOR, the column pertains to a result set associated with a cursor.
 - For DESCRIBE INPUT, the column pertains to parameter markers.
 - For DESCRIBE PROCEDURE, the column pertains to the result sets returned by the stored procedure.

SQLVAR entries

For each column or host variable described by the SQLDA, there are both base SQLVAR entries and extended SQLVAR entries.

Base SQLVAR entry

The base SQLVAR entry is always present. The fields of this entry contain the base information about the column or host variable such as data type code, length attribute (except for LOBs), column name (or label), host variable address, and indicator variable address.

Extended SQLVAR entry

The extended SQLVAR entry is needed (for each column) if the result

includes any LOB or distinct type³⁷ columns. For distinct types, the extended SQLVAR contains the distinct type name. For LOBs, the extended SQLVAR contains the length attribute of the host variable and a pointer to the buffer that contains the actual length. If locators are used to represent LOBs, an extended SQLVAR is not necessary.

The extended SQLVAR entry is also needed for each column when the USING BOTH clause was specified, which indicates that both columns names and labels are returned. (**DESCRIBE output** is the only statement with the **USING BOTH** clause).

The fields in the extended SQLVAR that return LOB and distinct type information do not overlap, and the fields that return LOB and label information do not overlap. Depending on the combination of labels, LOBs and distinct types, more than one extended SQLVAR entry per column may be required to return the information. See “Determining how many SQLVAR occurrences are needed” on page 1661.

The following table shows how to map the base and extended SQLVAR entries. For an SQLDA that contains both base and extended SQLVAR entries, the base SQLVAR entries are in the first block, followed by a block of extended SQLVAR entries, which if necessary, are followed by a second block of extended SQLVAR entries. In each block, the number of occurrences of the SQLVAR entry is equal to the value in SQLD³⁸ even though many of the extended SQLVAR entries might be unused.

Table 143. Contents of SQLVAR arrays

					Set of SQLVAR entries		
LOBs	Distinct types ¹	7th byte of SQLDAID	SQLD	Minimum for SQLN ²	First set (Base)	Second set (Extended)	Third set (Extended)
USING BOTH clause not specified:							
No	No	Space	<i>n</i>	<i>n</i>	Column names, labels	Not Used	Not Used
Yes ³	Yes ³	2	<i>n</i>	<i>2n</i>	Column names, labels	LOBs, distinct types, or both	Not used
USING BOTH clause was specified:							
No	No	Space	<i>2n</i>	<i>2n</i>	Column names	Labels	Not used
Yes	No	2	<i>n</i>	<i>2n</i>	Column names	LOBs and labels	Not used
No	Yes	3	<i>n</i>	<i>3n</i>	Column names	Distinct types	Labels
Yes	Yes	3	<i>n</i>	<i>3n</i>	Column names	LOBs and distinct types	Labels

37. DESCRIBE INPUT does not return information about distinct types.

38. When an extended SQLVAR entry is present for each column for *labels* (and there are no LOB or distinct type columns in the result),

Table 143. Contents of SQLVAR arrays (continued)

				Set of SQLVAR entries			
LOBs	Distinct types ¹	7th byte of SQLDAID	SQLD	Minimum for SQLN ²	First set (Base)	Second set (Extended)	Third set (Extended)
Notes:							
1. DESCRIBE INPUT does not return information about distinct types.							
2. The number of columns or host variables that the SQLDA describes.							
3. Either LOBs, distinct types, or both are present.							
4. Here, the 7th byte is set to a space and SQLD is set to two times the number of columns in the result. For all other values of the 7th byte for USING BOTH, SQLD is set to the number of columns in the result, and the 7th byte can be used to determine how many SQLVAR entries are needed for each column of the result.							

Determining how many SQLVAR occurrences are needed:

The number of SQLVAR occurrences needed depends on the statement that the SQLDA was provided for and the data types of the columns or parameters being described.

If the USING BOTH clause was not specified for the statement and neither LOBs nor distinct types are present in the result, only one SQLVAR entry (a base entry) is needed for each column. The 7th byte of SQLDAID is set to a space. The SQLD is set to the number of columns in the result and represents the number of SQLVAR occurrences needed. If an insufficient number of SQLVAR occurrences were provided, DB2 returns a +236 warning in SQLCODE if the standards option was set. Otherwise, SQLCODE is zero.

If USING BOTH is specified and neither LOBs nor distinct types are present in the result, an extended SQLVAR entry per column is needed for the labels in addition to the base SQLVAR entry. The 7th byte of the SQLDAID is set to space. SQLD is set to the twice the number of columns in the result and represents the combined number of base and extended SQLVAR occurrences needed.

If LOBs, distinct types, or both are present in the results, one extended SQLVAR entry is needed per column in addition to the base SQLVAR entry with one exception. The exception is that when the USING BOTH clause is specified and distinct types are present in the results, two extended SQLVAR entries per column are needed. When a sufficient number of SQLVAR entries are provided in the SQLDA for both the base and extended SQLVARs, information for the LOBs and distinct types is returned. The 7th byte of SQLDAID is set to the number of SQLVAR entries that were used for each column:

- 2 Two SQLVAR entries per column (a base and an extended)
- 3 Three SQLVAR entries per column (a base and two extended)

SQLD is set to the number of columns in the result. Therefore, the value of the 7th byte of SQLDAID multiplied by the value of SQLD is the total number SQLVAR entries that were provided.

Otherwise, when an insufficient number of SQLVAR entries have been provided when LOBs or distinct types are present, DB2 indicates that by returning one of the following warnings in SQLCODE. DB2 also sets the 7th byte of SQLDAID to indicate how many SQLVAR entries are needed for each column of the result.

+237 There are insufficient SQLVAR entries to describe the data, and the data

includes distinct types. In this case, there were enough base SQLVAR entries to describe the data, so the base SQLVAR entries are set. However, sufficient extended SQLVAR entries were not provided so the distinct type names are not returned.

- +238 There are insufficient SQLVAR entries to describe the data, and the data includes LOBs. In this case no information is returned in the SQLVAR entries.
- +239 There are insufficient SQLVAR entries to describe the data, and the data includes distinct types. There weren't even enough base SQLVAR entries. In this case no information is returned in the SQLVAR entries.

Field descriptions of an occurrence of a base SQLVAR:

The fields of a base SQLVAR have different uses depending on the SQL statement.

The following table describes the contents of the fields of a base SQLVAR.

Table 144. Fields in an occurrence of a base SQLVAR

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE ¹ and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqltype SQLTYPE	SMALLINT	Indicates the data type of the column or parameter and whether it can contain null values. For a description of the type codes, see Table 146 on page 1666. For a distinct type, the data type on which the distinct type was based is placed in this field. The base SQLVAR provides no indication that this is part of the description of a distinct type.	Indicates the data type of the host variable and whether an indicator variable is provided. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string. For a description of the type codes, see "SQLTYPE and SQLLEN" on page 1666.
sqllen SQLLEN	SMALLINT	The length attribute of the column or parameter. For datetime data, the length of the string representation of the value. See "SQLTYPE and SQLLEN" on page 1666 for a description of allowable values. For LOBs, the value is 0 regardless of the length attribute of the LOB. For XML, the value is 0. Field SQLLONGLEN in the extended SQLVAR contains the length attribute.	The length attribute of the host variable. See "SQLTYPE and SQLLEN" on page 1666 for a description of allowable values. For LOBs, the value is 0 regardless of the length attribute of the LOB. Field SQLLONGLEN in the extended SQLVAR contains the length attribute. For XML AS BLOB, CLOB, or DBCLOB, sqllen is 0 as for LOB types.

Table 144. Fields in an occurrence of a base SQLVAR (continued)

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE ¹ and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqldata SQLDATA	pointer	<p>For string columns or parameters, SQLDATA contains X'0000zzzz', where zzzz is the associated CCSID. For character strings, SQLDATA can alternatively contain X'FFFF', which indicates bit data. Not used for other types of data.</p> <p>For datetime columns, SQLDATA can contain the CCSID of the string representation of the datetime value.</p> <p>For DESCRIBE PROCEDURE, the result set locator value associated with the result set.</p>	Contains the address of the host variable.
sqlind SQLIND	pointer	<p>Reserved</p> <p>For DESCRIBE PROCEDURE, it is set to -1.</p>	Contains the address of an associated indicator variable, if SQLTYPE is odd. Otherwise, the field is not used.
sqlname SQLNAME	VARCHAR(30)	<p>Contains the unqualified name or label of the column, or a string of length zero if the name or label does not exist. If the name is longer than 30 bytes, it is truncated at a byte boundary. For more information about column names, see Names of result columns.</p> <p>For DESCRIBE PROCEDURE, SQLNAME contains the cursor name used by the stored procedure to return the result set. The values for SQLNAME appear in the order the cursors were opened by the stored procedure.</p> <p>For DESCRIBE INPUT, SQLNAME is not used.</p>	<p>Can contain CCSID and/or host-variable-array dimension information. DB2 interprets the third and fourth byte of the data portion of SQLNAME as the CCSID of the host variable if all of the following are true and the third and fourth byte are not X'0000':</p> <ul style="list-style-type: none"> • The 6th byte of SQLDAID is '+' (x'4E') • SQLTYPE indicates the host variable is a string variable • The length of SQLNAME is 8 • The first two bytes of the data portion of SQLNAME are X'0000'. <p>If the third and fourth byte of the data portion of SQLNAME are X'0000', DB2 uses the appropriate default CCSID.</p> <p>For FETCH, OPEN, INSERT, and EXECUTE, if the length of SQLNAME is 8, and the first two bytes of the data portion of SQLNAME are X'0000', DB2 interprets the fifth through eighth bytes of the data portion of the SQLNAME field as follows:</p>

Table 144. Fields in an occurrence of a base SQLVAR (continued)

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE ¹ and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
(cont.) sqlname SQLNAME			<ul style="list-style-type: none"> • fifth and sixth bytes: a flag field that indicates the type of host variable that is being described by the current SQLDA entry. The values of this field are as follows: <ul style="list-style-type: none"> – X'0000' - host variable – X'0100' - XML host variable (XML AS BLOB, XML AS CLOB, XML AS DBCLOB) – X'0001' - host variable array – X'0101' - XML host variable array – X'0002' - special host variable that represents the value for 'n' in a multiple-row INSERT statement. • seventh and eighth bytes: if the sixth byte is X'01', a binary small integer (halfword) that represents the dimension of the host-variable-array, and the corresponding indicator-array if one is specified.

Notes:

1. The third column of this table represents several forms of the DESCRIBE statement.
 - For DESCRIBE *output* and PREPARE INTO, the column pertains to columns of the result table.
 - For DESCRIBE CURSOR, the column pertains to a result set associated with a cursor.
 - For DESCRIBE INPUT, the column pertains to parameter markers.
 - For DESCRIBE PROCEDURE, the column pertains to the result sets returned by the stored procedure.

Field descriptions of an occurrence of an extended SQLVAR:

The fields of an extended SQLVAR have different uses depending on the SQL statement.

The following table describes the contents of the fields of an extended SQLVAR entry.

Table 145. Fields in an occurrence of an extended SQLVAR

C name assembler, COBOL, or PL/I name	Data type	Usage in DESCRIBE ¹ and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
len.sqllonglen SQLLONGL SQLLONGLEN	INTEGER	The length attribute of a LOB (BLOB, CLOB, or DBCLOB) column.	The length attribute of a LOB (BLOB, CLOB, or DBCLOB) host variable. DB2 ignores the SQLLEN field in the base SQLVAR for these data types. The length attribute indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB.
*	INTEGER	Reserved.	Reserved.
sqldatalen SQLDATA SQLDATALEN	pointer	Not used.	Used only for LOB (BLOB, CLOB, and DBCLOB) host variables. If the value of the field is null, the actual length of the LOB is stored in the 4 bytes immediately before the start of the data, and SQLDATA points to the first byte of the field length. The actual length indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB. If the value of the field is not null, the field contains a pointer to a 4-byte long buffer that contains the actual length <i>in bytes</i> (even for DBCLOBs) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR. Regardless of whether this field is used, field SQLLONGLEN must be set.

Table 145. Fields in an occurrence of an extended SQLVAR (continued)

C name assembler, COBOL, or PL/I name	Data type	Usage in DESCRIBE ¹ and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqldatatype_name SQLTNAME SQLDATATYPE- NAME	VARCHAR(30)	<p>A SQLTNAME field of the extended SQLVAR is set to one of the following:</p> <ul style="list-style-type: none"> For a distinct type column, the database manager sets this field to the fully qualified distinct type name. If the qualified name is longer than 30 bytes, it is truncated. For a label, the database manager sets this field to label. <p>In the case that both a distinct type name and a label need to be returned in extended SQLVAR entries (USING BOTH has been specified), the distinct type name is returned in the first extended SQLVAR entry and the label in the second extended SQLVAR entry.</p>	Not used.

Note:

- The third column of this table represents several forms of the DESCRIBE statement.
 - For DESCRIBE *output* and PREPARE INTO, the column pertains to columns of the result table.
 - For DESCRIBE CURSOR, the column pertains to a result set associated with a cursor.
 - For DESCRIBE INPUT, the column pertains to parameter markers.
 - For DESCRIBE PROCEDURE, the column pertains to the result sets returned by the stored procedure.

SQLTYPE and SQLLEN:

The contents of the SQLTYPE and SQLLEN fields of the SQLDA depends on the SQL statement that is returning the value.

The following table shows the values that can appear in the SQLTYPE and SQLLEN fields of the SQLDA. In DESCRIBE and PREPARE INTO, an even value of SQLTYPE means the column does not allow nulls, and an odd value means the column does allow nulls. In DESCRIBE INPUT statements, only odd values are returned for SQLTYPE. In FETCH, OPEN, EXECUTE, and CALL, an even value of SQLTYPE means no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

Table 146. SQLTYPE and SQLLEN values for DESCRIBE, PREPARE INTO, FETCH, OPEN, EXECUTE, and CALL

SQLTYPE	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
	Column or parameter data type	SQLLEN	Host variable data type	SQLLEN
384/385	date	10 ¹	fixed-length character string representation of a date	length attribute of the host variable

Table 146. *SQLTYPE* and *SQLLEN* values for *DESCRIBE*, *PREPARE INTO*, *FETCH*, *OPEN*, *EXECUTE*, and *CALL* (continued)

SQLTYPE	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
	Column or parameter data type	SQLLEN	Host variable data type	SQLLEN
388/389	time	8 ²	fixed-length character string representation of a time	length attribute of the host variable
392/393	timestamp	26	fixed-length character string representation of a timestamp	length attribute of the host variable
400/401	N/A	N/A	NUL-terminated graphic string	length attribute of the host variable
404/405	BLOB	0 ³	BLOB or XML AS BLOB	Not used. ³
408/409	CLOB	0 ³	CLOB or XML AS CLOB	Not used. ³
412/413	DBCLOB	0 ³	DBCLOB or XML AS DBCLOB	Not used. ³
448/449	varying-length character string	length attribute of the column	varying-length character string	length attribute of the host variable
452/453	fixed-length character string	length attribute of the column	fixed-length character string	length attribute of the host variable
456/457	long varying-length character string	length attribute of the column	long varying-length character string	length attribute of the host variable
	SQLTYPE values 448/449 are returned instead of 456/457 for statements that are bound in Version 9 or later.			
460/461	N/A	N/A	NUL-terminated character string	length attribute of the host variable
464/465	varying-length graphic string	length attribute of the column	varying-length graphic string	length attribute of the host variable
468/469	fixed-length graphic string	length attribute of the column	fixed-length graphic string	length attribute of the host variable
472/473	long graphic string	length attribute of the column	long graphic string	length attribute of the host variable
	SQLTYPE values 464/465 are returned instead of 472/473 for statements that are bound in Version 9 or later.			
480/481	floating point	4 for single precision, 8 for double precision	floating point	4 for single precision, 8 for double precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
492/493	big integer ⁴	8	big integer	8
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2

Table 146. *SQLTYPE* and *SQLLEN* values for *DESCRIBE*, *PREPARE INTO*, *FETCH*, *OPEN*, *EXECUTE*, and *CALL* (continued)

SQLTYPE	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
	Column or parameter data type	SQLLEN	Host variable data type	SQLLEN
504/505	N/A	N/A	DISPLAY SIGN LEADING SEPARATE, NATIONAL SIGN LEADING SEPARATE	precision in byte 1; scale in byte 2
908/909	varying-length binary string	length attribute of the column	varying-length binary string	length attribute of the host variable
912/913	fixed-length binary string	length attribute of the column	fixed-length binary string	length attribute of the host variable
916/917	BLOB_FILE	267		
920/921	CLOB_FILE	267		
924/925	DBCLOB_FILE	267		
996/997	DECFLOAT(16)	8	DECFLOAT(16)	8
996/997	DECFLOAT(34)	16	DECFLOAT(34)	16
988/989	XML	0	Invalid. Instead, use one of the following: XML AS BLOB, XML AS CLOB, XML AS DBCLOB	Not used

Note:

1. SQLLEN might be different if a date installation exit is specified.
2. SQLLEN might be different if a time installation exit is specified.
3. Field SQLLONGLEN in the extended SQLVAR contains the length attribute of the column.
4. BIGINT is supported by other DB2 platforms.

SQLDATA:

Depending on the data type of the string column that the SQLVAR is describing, the SQLDATA field can contain different CCSID values.

The following table identifies the CCSID values that appear in the SQLDATA field when the SQLVAR element describes a string column.

Table 147. *CCSID* values for *SQLDATA*

Data type	Subtype	Bytes 1 and 2	Bytes 3 and 4
Character	SBCS data	X'0000'	CCSID
Character	mixed data	X'0000'	CCSID
Character	BIT data	X'0000'	X'FFFF'
Graphic	N/A	X'0000'	CCSID
Any other data type	N/A	N/A	N/A

Unrecognized and unsupported SQLTYPES

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as at the receiver of the data. This support is particularly important as new data types are added to the product.

New data types might not be supported by the sender or receiver of the data and might not be recognized by the sender or receiver of the data. Depending on the situation, the new data type might be returned, a compatible data type that is agreed to by both the sender and the receiver of the data might be returned, or an error might occur.

When the sender and receiver agree to use a compatible data type, the following table indicates the mapping that takes place. This mapping takes place when at least one of the sender or receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Table 148. Compatible data types for unsupported data types

Data type	Compatible data type
ROWID	VARCHAR(40) FOR BIT DATA

No indication is given in the SQLDA that the data type is substituted.

The included SQLDA

Only assembler, C, C++, COBOL, and PL/I C are supported for the SQLDA that is given by INCLUDE SQLDA.

assembler:

```
SQLTRIPL EQU    C'3'
SQLDOUBL EQU    C'2'
SQLSINGL EQU    C' '
*
        SQLSECT SAVE
*
SQLDA    DSECT
SQLDAID  DS      CL8      ID
SQLDABC  DS      F        BYTE COUNT
SQLN     DS      H        COUNT SQLVAR/SQLVAR2 ENTRIES
SQLD     DS      H        COUNT VARS (TWICE IF USING BOTH)
*
SQLVAR   DS      0F        BEGIN VARS
SQLVARN  DSECT ,          NTH VARIABLE
SQLTYPE  DS      H        DATA TYPE CODE
SQLLEN   DS      0H        LENGTH
SQLPRCSN DS      X        DEC PRECISION
SQLSCALE DS      X        DEC SCALE
SQLDATA  DS      A        ADDR OF VAR
SQLIND   DS      A        ADDR OF IND
SQLNAME  DS      H,CL30    DESCRIBE NAME
SQLVSIZ  EQU     *-SQLDATA
SQLSIZV  EQU     *-SQLVARN
*
SQLDA    DSECT
SQLVAR2  DS      0F        BEGIN EXTENDED FIELDS OF VARS
SQLVAR2N DSECT ,          EXTENDED FIELDS OF NTH VARIABLE
SQLLONGL DS      F        LENGTH
SQLRSVDL DS      F        RESERVED
SQLDATAL DS      A        ADDR OF LENGTH IN BYTES
SQLNAME  DS      H,CL30    DESCRIBE NAME
*
        SQLSECT RESTORE
```

In the above declaration, SQLSECT SAVE is a macro invocation that remembers the current CSECT name. SQLSECT RESTORE is a macro invocation that continues that CSECT.

C and C++:

```
#ifndef SQLDA_SIZE /* Permit duplicate Includes */
/**/
struct sqlvar
{
    short sqltype;
    short sqllen;
    char *sqldata;
    short *sqlind;
    struct sqlname
    {
        short length;
        char data[30];
    } sqlname;
};

/**/
struct sqlvar2
{
    struct
    {
        long sqllonglen;
        unsigned long reserved;
    } len;
    char *sqldatalen;
    struct sqldistinct_type
```



```

#define GETSQLDALONGLEN(daptr,n) ( \
    (long) (((struct sqlvar2 *) &((daptr);->sqlvar[(n) + \
    ((daptr)->sqld)])) \
    ->len.sqlllonglen))

/**/
/*****
/* SETSQLDALONGLEN(daptr,n,len) sets the sqlllonglen field of the */
/* sqlda pointed to by daptr to len for the nth entry. Use this only */
/* if the sqlda was doubled or tripled and the nth SQLVAR entry has */
/* a LOB datatype. */
/*****
#define SETSQLDALONGLEN(daptr,n,length) { \
    struct sqlvar2 *var2ptr; \
    var2ptr = (struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + ((daptr)->sqld)]); \
    var2ptr->len.sqlllonglen = (long ) (length); \
}

/**/
/*****
/* GETSQLDALENPTR(daptr,n) returns a pointer to the data length for */
/* the nth entry in the sqlda pointed to by daptr. Unlike the inline */
/* value (union sql8bytelen len), which is 8 bytes, the sqldatalen */
/* pointer field returns a pointer to a long (4 byte) integer. */
/* If the SQLDALEN pointer is zero, a NULL pointer is be returned. */
/* */
/* NOTE: Use this only if the sqlda has been doubled or tripled. */
/*****
#define GETSQLDALENPTR(daptr,n) ( \
    (((struct sqlvar2 *) &((daptr);->sqlvar[(n) + (daptr)->sqld]) \
    ->sqldatalen == NULL) ? \
    ((long *) NULL) : \
    ((long *) ((struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + (daptr)->sqld]) \
        ->sqldatalen ) )

/**/
/*****
/* SETSQLDALENPTR(daptr,n,ptr) sets a pointer to the data length for */
/* the nth entry in the sqlda pointed to by daptr. */
/* Use this only if the sqlda has been doubled or tripled. */
/*****
#define SETSQLDALENPTR(daptr,n,ptr) { \
    struct sqlvar2 *var2ptr; \
    var2ptr = (struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + ((daptr)->sqld)]); \
    var2ptr->sqldatalen = (char *) ptr; \
}

/**/
#define SQLDASIZE(n) \
    ( sizeof(struct sqlda) + ((n)-1) * sizeof(struct sqlvar) )
#endif /* SQLDASIZE */

```

COBOL (IBM COBOL only):

```

01 SQLDA.
   05 SQLDAID PIC X(8).
   05 SQLDABC PIC S9(9) BINARY.
   05 SQLN    PIC S9(4) BINARY.
   05 SQLD    PIC S9(4) BINARY.
   05 SQLVAR OCCURS 0 TO 750 TIMES DEPENDING ON SQLN.
   10 SQLVAR1.
       15 SQLTYPE PIC S9(4) BINARY.
       15 SQLLEN  PIC S9(4) BINARY.
       15 FILLER  REDEFINES SQLLEN.
           20 SQLPRECISION PIC X.
           20 SQLSCALE    PIC X.
       15 SQLDATA POINTER.
       15 SQLIND  POINTER.

```

```

15 SQLNAME.
    49 SQLNAMEL PIC S9(4) BINARY.
    49 SQLNAMEC PIC X(30).
10 SQLVAR2 REDEFINES SQLVAR1.
    15 SQLVAR2-RESERVED-1 PIC S9(9) BINARY.
    15 SQLLONGLEN          REDEFINES SQLVAR2-RESERVED-1
                          PIC S9(9) BINARY.
    15 SQLVAR2-RESERVED-2 PIC S9(9) BINARY.
    15 SQLDATALEN         POINTER.
    15 SQLDATATYPE-NAME.
        20 SQLDATATYPE-NAMEL PIC S9(4) BINARY.
        20 SQLDATATYPE-NAMEC PIC X(30).

```

PL/I:

```

DECLARE
  1 SQLDA BASED(SQLDAPTR),
  2 SQLDAID CHAR(8),
  2 SQLDABC FIXED(31) BINARY,
  2 SQLN    FIXED(15) BINARY,
  2 SQLD    FIXED(15) BINARY,
  2 SQLVAR(SQLSIZE REFER(SQLN)),
  3 SQLTYPE FIXED(15) BINARY,
  3 SQLLEN   FIXED(15) BINARY,
  3 SQLDATA  POINTER,
  3 SQLIND   POINTER,
  3 SQLNAME  CHAR(30) VAR;
/* */
DECLARE
  1 SQLDA2 BASED(SQLDAPTR),
  2 SQLDAID2 CHAR(8),
  2 SQLDABC2 FIXED(31) BINARY,
  2 SQLN2     FIXED(15) BINARY,
  2 SQLD2     FIXED(15) BINARY,
  2 SQLVAR2(SQLSIZE REFER(SQLN2)),
  3 SQLBIGLEN,
  4 SQLLONGL FIXED(31) BINARY,
  4 SQLRSVDL  FIXED(31) BINARY,
  3 SQLDATAL  POINTER,
  3 SQLTNAME  CHAR(30) VAR;
/* */
DECLARE SQLSIZE    FIXED(15) BINARY;
DECLARE SQLDAPTR   POINTER;
DECLARE SQLTRIPLED CHAR(1)   INITIAL('3');
DECLARE SQLDOUBLED CHAR(1)   INITIAL('2');
DECLARE SQLSINGLED CHAR(1)   INITIAL(' ');

```

Identifying an SQLDA in C or C++

A *descriptor-name* can be specified in the CALL, DESCRIBE, EXECUTE, FETCH, and OPEN statements. When the host application is written in C or C++, *descriptor-name* can be a pointer variable with pointer notation.

For example, *descriptor-name* could be declared as

```
sqlda *outsqlda;
```

Afterwards, it could be used in a statement like the following:

```
EXEC SQL DESCRIBE STMT1 INTO DESCRIPTOR :*outsqlda;
```

The REXX SQLDA

A REXX SQLDA consists of a set of REXX variables with a common stem. The stem must be a REXX variable name that contains no periods and is the same as the value of *descriptor-name* that you specify when you use the SQLDA in an SQL statement. DB2 does not support the INCLUDE SQLDA statement in REXX.

The following table shows the variable names in a REXX SQLDA. The values in the second column of the table are values that DB2 inserts into the SQLDA when the statement executes. Except where noted otherwise, the values in the third column of the table are values that the application must put in the SQLDA before the statement executes.

Table 149. Fields of a REXX SQLDA

Variable name	Usage in DESCRIBE and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
<i>stem</i> .SQLD	<p>The number of columns that are described in the SQLDA. Double that number if USING BOTH appears in the DESCRIBE or PREPARE INTO statement. Contains a 0 if the statement string is not a query.</p> <p>For DESCRIBE PROCEDURE, the number of result sets returned by the stored procedure. Contains a 0 if no result sets are returned.</p>	The number of host variables that are used by the SQL statement.
Each SQLDA contains <i>stem</i> .SQLD of the following variables. Therefore, $1 \leq n \leq \text{stem}.SQLD$. There is one occurrence of each variable for each column of the result table or host variable that is described by the SQLDA. This set of variables is equivalent to the SQLVAR structure in the SQLDA for other languages.		
<i>stem.n</i> .SQLTYPE	<p>Indicates the data type of the column or parameter and whether it can contain null values. For a description of the type codes, see "SQLTYPE and SQLLEN" on page 1666.</p> <p>For a distinct type, the data type on which the distinct type was based is placed in this field. The base SQLVAR provides no indication that this is part of the description of a distinct type.</p>	Indicates the data type of the host variable and whether an indicator variable is provided. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string. For a description of the type codes, see "SQLTYPE and SQLLEN" on page 1666.
<i>stem.n</i> .SQLLEN	For a column other than a DECIMAL or NUMERIC column, the length attribute of the column or parameter. For datetime data, the length of the string representation of the value. See "SQLTYPE and SQLLEN" on page 1666 for a description of allowable values.	For a host variable that does not have a decimal data type, the length attribute of the host variable. See "SQLTYPE and SQLLEN" on page 1666 for a description of allowable values.
<i>stem.n</i> .SQLLEN.SQLPRECISION	For a DECIMAL or NUMERIC column, the precision of the column or parameter.	For a host variable with a decimal data type, the precision of the host variable.
<i>stem.n</i> .SQLLEN.SQLSCALE	For a DECIMAL or NUMERIC column, the scale of the column or parameter.	For a host variable with a decimal data type, the scale of the host variable.

Table 149. Fields of a REXX SQLDA (continued)

Variable name	Usage in DESCRIBE and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
<i>stem.n</i> .SQLCCSID	For a string column or parameter, the CCSID of the column or parameter.	For a string host variable, the CCSID of the host variable.
<i>stem.n</i> .SQLUSECCSID	Not used.	Set to a new CCSID. If set, REXX will change the CCSID of the SQLDATA.
<i>stem.n</i> .SQLLOCATOR	For DESCRIBE PROCEDURE, the value of a result set locator.	Not used.
<i>stem.n</i> .SQLDATA	Not used.	Before EXECUTE or OPEN, contains the value of an input host variable. The application must supply this value. After FETCH, contains the values of an output host variable.
<i>stem.n</i> .SQLIND	Not used.	Before EXECUTE or OPEN, contains a negative number to indicate that the input host variable in <i>stem.n</i> .SQLDATA is null. The application must supply this value. After FETCH, contains a negative number if the value of the output host variable in <i>stem.n</i> .SQLDATA is null.
<i>stem.n</i> .SQLNAME	The name of the <i>n</i> th column in the result table. For DESCRIBE PROCEDURE, contains the cursor name that is used by the stored procedure to return the result set. The values for SQLNAME appear in the order that the cursors were opened by the stored procedure.	Not used.

DB2 catalog tables

DB2 for z/OS maintains a set of tables (in database DSNDB06) called the DB2 catalog.

About these topics

These topics describe that catalog by describing the columns of each catalog table.

The catalog tables describe such things as table spaces, tables, columns, indexes, privileges, application plans, and packages. Authorized users can query the catalog; however, it is primarily intended for use by DB2 and is therefore subject to change. All catalog tables are qualified by SYSIBM. Do not use this qualifier for user-defined tables.

The catalog tables are updated by DB2 during normal operations in response to certain SQL statements, commands, and utilities.

Additional information

Release dependency indicators: Some objects depend on functions in particular releases of DB2. If you are running on a release of DB2 and fall back to a previous release, an object that depends on the more recent release becomes frozen. The object is marked with a release dependency indicator and is unavailable until remigration. The release dependency indicator, which is listed in the IBMREQD column of the catalog tables, shows the release of DB2 upon which the objects depends. Release dependency indicators in IBMREQD are defined by the following values:

Value	Meaning
-------	---------

B	Version 1R3 dependency indicator, not from the machine-readable material (MRM) tape
C	Version 2R1 dependency indicator, not from MRM tape
D	Version 2R2 dependency indicator, not from MRM tape
E	Version 2R3 dependency indicator, not from MRM tape
F	Version 3R1 dependency indicator, not from MRM tape
G	Version 4 dependency indicator, not from MRM tape
H	Version 5 dependency indicator, not from MRM tape
I	Version 6 dependency indicator, not from MRM tape
J	Version 6 dependency indicator, not from MRM tape
K	Version 7 dependency indicator, not from MRM tape
L	Version 8 dependency indicator, not from MRM tape
M	Version 9 dependency indicator, not from MRM tape
N	Not from MRM tape, no dependency

I

Programming interface information

Not all catalog table columns are part of the general-use programming interface. Whether a column is part of this interface is indicated in a column labeled “Use” in the table that describes the column. The values that “Use” can assume are as follows:

Value	Meaning
-------	---------

G	Column is part of the general-use programming interface
S	Column is part of the product-sensitive interface
I	Column is for internal use only
N	Column is not used

For columns for which “Use” is N or I, the name of the column and its description do not appear in the explanation of the column.

Table spaces and indexes

DB2 catalog tables are contained in certain table spaces and have indexes.

The following table lists the table space and indexes for each catalog table and lists the index fields for each index. The indexes are in ascending order, except where noted.

Table 150. Table spaces and indexes for the catalog tables

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
SYSCOPY	SYSCOPY	DSNUCH01	DBNAME.TSNAME.START_RBA. ¹ TIMESTAMP ¹
		DSNUCX01	DSNAME
SYSCONTX	SYSCONTEXT	DSNCTX01	NAME
		DSNCTX02	SYSTEMAUTHID
		DSNCTX03	CONTEXTID
		DSNCTX04	DEFAULTROLE
	SYSCONTEXTAUTHIDS	DSNCDX01	CONTEXTID.AUTHID
		DSNCDX02	ROLE
	SYSCTXTTRUSTATTRS	DSNCAX01	CONTEXTID. NAME.VALUE
SYSDBASE	SYSCOLAUTH	DSNACX01	CREATOR.TNAME.COLNAME
	SYSCOLUMNS	DSNDCX01	TBCREATOR.TBNAME.NAME
		DSNDCX02	TYPESCHEMA.TYPENAME
	SYSFIELDS		
	SYSFOREIGNKEYS	DSNDRH01	CREATOR.TBNAME.RELNAME
	SYSINDEXES	DSNDXX01	CREATOR.NAME
		DSNDXX02	DBNAME.INDEXSPACE
		DSNDXX03	TBCREATOR.TBNAME.CREATOR. NAME
		DSNDXX04	INDEXTYPE
	SYSINDEXPART	DSNDRX01	IXCREATOR.IXNAME.PARTITION
		DSNDRX02	STORNAME

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
	SYSINDEXSPACESTATS	DSNRTX03	CREATOR.NAME
	SYSKEYS	DSNDKX01	IXCREATOR.IXNAME.COLNAME
	SYSRELS	DSNDLX01	REFTBCREATOR.REFTBNAME
		DSNDLX02	CREATOR.TBNAME
		DSNDLX03	IXOWNER.IXNAME
	SYSSYNONYMS	DSNDYX01	CREATOR.NAME
	SYSTABAUTH	DSNATX01	GRANTOR.GRANTORTYPE
		DSNATX02	GRANTEE.TCREATOR.TTNAME. GRANTEETYPE.UPDATECOLS. ALTERAUTH.DELETEAUTH. INDEXAUTH.INSERTAUTH. SELECTAUTH.UPDATEAUTH. CAPTUREAUTH.REFERENCESAUTH. REFCOLS.TRIGGERAUTH
		DSNATX03	GRANTEE.GRANTEETYPE.COLLID CONTOKEN
		DSNATX04	TCREATOR.TTNAME
	SYSTABLEPART	DSNDPX01	DBNAME.TSNAME.PARTITION
		DSNDPX02	STORNAME
		DSNDPX03	DBNAME.TSNAME.LOGICAL_PART
		DSNDPX04	IXCREATOR.IXNAME
	SYSTABLES	DSNDTX01	CREATOR.NAME
		DSNDTX02	DBID.OBID.CREATOR.NAME
		DSNDTX03	TBCREATOR.TBNAME
	SYSTABLESPACE	DSNDSX01	DBNAME.NAME
SYSDBAUT	SYSDATABASE	DSNDDH01	NAME

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNDDX02	GROUP_MEMBER
	SYSDBAUTH	DSNADH01	GRANTEE.NAME.GRANTEETYPE
		DSNADX01	GRANTOR.NAME.GRANTORTYPE
SYSDDF	IPLIST	DSNDUX01	LINKNAME.IPADDR
	IPNAMES	DSNFPX01	LINKNAME
	LOCATIONS	DSNFCX01	LOCATION
	LULIST	DSNFLX01	LINKNAME.LUNAME
		DSNFLX02	LUNAME
	LUMODES	DSNFMX01	LUNAME.MODENAME
	LUNAMES	DSNFNX01	LUNAME
	MODESELECT	DSNFDX01	LUNAME.AUTHID ¹ .PLANNAME ¹
	USERNAMES	DSNFEX01	TYPE.AUTHID ¹ .LINKNAME ¹
SYSEBCDC	SYSDUMMY1		
SYSGPAUT	SYSRESAUTH	DSNAGH01	GRANTEE.QUALIFIER. NAME.OBTYPE. GRANTEETYPE
		DSNAGX01	GRANTOR.QUALIFIER. NAME.OBTYPE. GRANTORTYPE
SYSGROUP	SYSSTOGROUP	DSNSSH01	NAME
	SYSVOLUMES		
SYSGRSNS	SYSROUTINES_OPTS	DSNR0X01	SCHEMA.ROUTINENAME. BUILDDATE.BUILDTIME

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
	SYSROUTINES_SRC	DSNRSX01	ROUTINENAME
		DSNRSX02	SCHEMA.ROUTINENAME. BUILDDATE. SEQNO
SYSHIST	SYSCOLDIST_HIST	DSNHFX01	TBOWNER.TBNAME. NAME.STATTIME
	SYSCOLUMNS_HIST	DSNHEX01	TBCREATOR.TBNAME. NAME.STATTIME
	SYSINDEXES_HIST	DSNHXX01	TBCREATOR.TBNAME. NAME.STATTIME
		DSNHXX02	CREATOR.NAME
	SYSINDEXPART_HIST	DSNHGX01	IXCREATOR.IXNAME. PARTITION.STATTIME
	SYSINDEXSTATS_HIST	DSNHIX01	OWNER.NAME. PARTITION.STATTIME
	SYSLOBSTATS_HIST	DSNHJX01	DBNAME.NAME.STATTIME
	SYSKEYTARGETS_HIST	DSNHKX01	IXSCHEMA.IXNAME. KEYSEQ.STATTIME
	SYSKEYTGTDIST_HIST	DSNTDX02	IXSCHEMA.IXNAME KEYSEQ.STATTIME
	SYSTABLEPART_HIST	DSNHXX01	DBNAME.TSNAME. PARTITION.STATNAME
	SYSTABLES_HIST	DSNHDX01	CREATOR.NAME.STATTIME
	SYSTABSTATS_HIST	DSNHBX01	OWNER.NAME. PARTITION.STATTIME
SYSJAVA	SYSJARCONTENTS	DSNJCX01	JARSCHEMA.JAR_ID
	SYSJAROBJECTS	DSNJOX01	JARSCHEMA.JAR_ID

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
	SYSJAVAOPTS	DSNJVX01	JARSCHEMA.JAR_ID
	SYSJAVAPATHS	DSNJIPX01	JARSCHEMA.JAR_ID.ORDINAL
		DSNJIPX02	PE_JARSCHEMA.PE_JAR_ID
SYSJAUXA LOB	SYSJARDATA	DSNJDY01	JAR_DATA
SYSJAUXB LOB	SYSJARCLASS_SOURCE	DSNJSX01	CLASS_SOURCE
SYSOBJ	SYSAUXRELS	DSNOXX01	TBOWNER.TBNAME
		DSNOXX02	AUXTBOWNER.AUXTBNAME
	SYSCONSTDEP	DSNCCX01	BSHEMA.BNAME.BTYPE
		DSNCCX02	DTBCREATOR.DTBNAME
	SYSDATATYPES	DSNODX01	SCHEMA.NAME
		DSNODX02	DATATYPEID ¹
	SYSDEPENDENCIES	DSNONX01	BSHEMA.BNAME. BCOLNAME.BTYPE. DSHEMA.DNAME. DCOLNAME.DTYPE
		DSNONX02	DSHEMA.DNAME. DCOLNAME.DTYPE. BSHEMA.BNAME. BCOLNAME.BTYPE
	SYSENVIRONMENT	DSNOEX01	ENVID
	SYSKEYCOLUSE	DSNCUX01	TBCREATOR.TBNAME. CONSTNAME.COLSEQ.
	SYSPARMS	DSNOPX01	SCHEMA.SPECIFICNAME. ROUTINETYPE.ROWTYPE ORDINAL.VERSION
		DSNOPX02	TYPESHEMA.TYPENAME. ROUTINETYPE.CAST_FUNCTION. OWNER.SCHEMA.SPECIFICNAME

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
 	SYSROUTINEAUTH	DSNOPX03	TYPESCHEMA.TYPENAME
		DSNOPX04	SCHEMA.SPECIFICNAME. ROUTINETYPE.VERSION
		DSNOAX01	GRANTOR.SCHEMA. SPECIFICNAME.ROUTINETYPE. GRANTEETYPE.EXECUTEAUTH. GRANTORTYPE
		DSNOAX02	GRANTEE.SCHEMA.SPECIFICNAME. ROUTINETYPE.GRANTEETYPE. EXECUTEAUTH.GRANTEDTS
 	SYSROUTINES	DSNOAX03	SCHEMA.SPECIFICNAME ROUTINETYPE
		DSNOFX01	NAME.PARM_COUNT. ROUTINETYPE.PARM_SIGNATURE. SCHEMA.PARM1.PARM2.PARM3. PARM4.PARM5.PARM6.PARM7. PARM8.PARM9.PARM10.PARM11. PARM12.PARM13.PARM14.PARM15. PARM16.PARM17.PARM18.PARM19. PARM20.PARM21.PARM22.PARM23. PARM24.PARM25.PARM26.PARM27. PARM28.PARM29.PARM30. VERSION
		DSNOFX02	SCHEMA.SPECIFICNAME. ROUTINETYPE.VERSION
		DSNOFX03	NAME.SCHEMA.CAST_FUNCTION. PARM_COUNT.PARM_SIGNATURE. PARM1
		DSNOFX04	ROUTINE_ID ¹
		DSNOFX05	SOURCESCHEMA.SOURCESPECIFIC. ROUTINETYPE
		DSNOFX06	SCHEMA.NAME.ROUTINETYPE. PARM_COUNT

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		DSNOFX07	NAME.PARM_COUNT. ROUTINETYPE. SCHEMA. PARM_SIGNATURE. PARM1.PARM2.PARM3. PARM4.PARM5.PARM6.PARM7. PARM8.PARM9.PARM10.PARM11. PARM12.PARM13.PARM14.PARM15. PARM16.PARM17.PARM18.PARM19. PARM20.PARM21.PARM22.PARM23. PARM24.PARM25.PARM26.PARM27. PARM28.PARM29.PARM30. VERSION
		DSNOFX08	JARSCHEMA. JAR_ID
	SYSSCHEMAAUTH	DSNSKX01	GRANTEE.SCHEMANAME. GRANTEETYPE
		DSNSKX02	GRANTOR.GRANTORTYPE
	SYSTABCONST	DSNCNX01	TBCREATOR.TBNAME.CONSTNAME
		DSNCNX02	IXOWNER IXNAME
	SYSTRIGGERS	DSNOTX01	SCHEMA.NAME.SEQNO
		DSNOTX02	TBOWNER.TBNAME
		DSNOTX03	SCHEMA.TRIGNAME
SYSPKAGE	SYSPACKAGE	DSNKKX01	LOCATION.COLLID.NAME. VERSION
		DSNKKX02	LOCATION.COLLID.NAME. CONTOKEN
	SYSPACKAUTH	DSNKAX01	GRANTOR.LOCATION.COLLID.NAME. GRANTORTYPE
		DSNKAX02	GRANTEE.LOCATION.COLLID. NAME.BINDAUTH.COPYAUTH. EXECUTEAUTH.GRANTEETYPE
		DSNKAX03	LOCATION.COLLID.NAME

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
	SYPACKDEP	DSNKDX01	DLOCATION.DCOLLID.DNAME. DCONTOKEN
		DSNKDX02	BQUALIFIER.BNAME.BTYPE
		DSNKDX03	BQUALIFIER.BNAME.BTYPE. DTYPE
	SYPACKLIST	DSNKLX01	LOCATION.COLLID.NAME
		DSNKLX02	PLANNAME.SEQNO.LOCATION. COLLID.NAME
	SYPACKSTMT	DSNKSX01	LOCATION.COLLID.NAME. CONTOKEN.STMTNOI.SECTNOI. SEQNO
	SYPKSYSTEM	DSNKYX01	LOCATION.COLLID.NAME. CONTOKEN.SYSTEM.ENABLE
	SYSPLSYSTEM	DSNKPX01	NAME.SYSTEM.ENABLE
SYSPLAN	SYSDBRM		
	SYSPLAN	DSNPPH01	NAME
	SYSPLANAUTH	DSNAPH01	GRANTEE.NAME.EXECUTEAUTH. GRANTEETYPE
		DSNAPX01	GRANTOR.GRANTORTYPE
	SYSPLANDEP	DSNGGX01	BCREATOR.BNAME.BTYPE
	SYSSTMT		
	SYSPLUXA	SYSROUTINESTEXT	
	SYSROLES	SYSOBJROLEDEP	DSNRDX01
			DSHEMA.DNAME.DTYPE.ROLENAME
		DSNRDX02	ROLENAME
	SYSROLES	DSNRLX01	NAME
	SYSRTSTS	SYSTABLESPACESTATS	DSNRTX01
			DBID.PSID.PARTITION.INSTANCE

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
 	SYSINDEXSPACESTAT	DSNRTX02	DBID.ISOBID.PARTITION.INSTANCE
SYSSEQ	SYSSEQUENCES	DSNSQX01	SCHEMA.NAME
		DSNSQX02	SEQUENCEID ¹
 	SYSSEQUENCEAUTH	DSNWCX01	SCHEMA.NAME
		DSNWCX02	GRANTOR.SCHEMA.NAME. GRANTORTYPE
		DSNWCX03	GRANTEE.SCHEMA.NAME. GRANTEETYPE
	SYSSEQUENCESDEP	DSNSRX01	DCREATOR.DNAME.DCOLNAME
		DSNSRX02	BSHEMA.BNAME.DTYPE
SYSSTATS	SYSCOLDIST	DSNTNX01	TBOWNER.TBNAME.NAME
	SYSCOLDISTSTATS	DSNTPX01	TBOWNER.TBNAME.NAME PARTITION
	SYSCOLSTATS	DSNTCX01	TBOWNER.TBNAME.NAME PARTITION
	SYSINDEXSTATS	DSNTXX01	OWNER.NAME.PARTITION
 	SYSKEYTARGETSTATS	DSNTKX01	IXSCHEMA.IXNAME.KEYSEQ. PARTITION
 	SYSKEYTGTDIST	DSNTDX01	IXSCHEMA.IXNAME.KEYSEQ
 	SYSKEYTGTDISTSTATS	DSNTSX01	IXSCHEMA.IXNAME.KEYSEQ. PARTITION
	SYSLOBSTATS	DSNLTN01	DBNAME.NAME
	SYSTABSTATS	DSNTTX01	OWNER.NAME.PARTITION

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
SYSSTR	SYSSTRINGS	DSNSSX01	OUTCCSID.INCCSID.IBMREQD
	SYSCHECKS	DSNSCX01	TBOWNER.TBNAME.CHECKNAME
	SYSCHECKS2	DSNCHX01	TBOWNER.TBNAME.CHECKNAME
	SYSCHECKDEP	DSNSDX01	TBOWNER.TBNAME.CHECKNAME COLNAME
SYSTARG 	SYSKEYTARGETS	DSNRKX01	IXSCHEMA.IXNAME. KEYSEQ
		DSNRKX02	DATATYPEID. KEYSPEC_INTERNAL
SYSUSER 	SYSUSERAUTH	DSNAUH01	GRANTEE GRANTEDTS. GRANTEETYPE
		DSNAUX01	GRANTOR.GRANTORTYPE
SYSVIEWS	SYSVIEWDEP	DSNGGX02	BCREATOR.BNAME.BTYPE
		DSNGGX03	BSCHEMA.BNAME.BTYPE
		DSNGGX04	BCREATOR.BNAME.BTYPE.DTYPE
		DSNVVX01	CREATOR.NAME.SEQNO.TYPE
	SYSVIEWS		
SYXML 	SYSXMLRELS	DSNXRX01	TBOWNER.TBNAME
		DSNXRX02	XMLTBOWNER.XMLTBNAME
	SYSXMLSTRINGS	DSNXSX01	STRINGID
		DSNXSX02	STRING
SYSXSR	XSROBJECTS	XSROBJ01	XSROBJECTID

Table 150. Table spaces and indexes for the catalog tables (continued)

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	INDEX SYSIBM. ...	INDEX FIELDS
		XSROBJ02	XSROBJECTSCHEMA.XSROBJECTNAME
		XSROBJ03	TARGETNAMESPACE.SCHEMALOCATION
		XSROBJ04	SCHEMALOCATION
	XSROBJECT- COMPONENTS	XSRCOMP01	XSRCOMPONENTID
		XSRCOMP02	TARGETNAMESPACE.SCHEMALOCATION
	XSROBJECT- HIERARCHIES	XSRHIER01	XSROBJECTID.TARGETNAMESPACE. SCHEMALOCATION
		XSRHIER02	XSROBJECTID.TARGETNAMESPACE
SYSXSRA1	XSROBJECTGRAMMAR	XSRXOG01	GRAMMAR
SYSXSRA2	XSROBJECTPROPERTY	XSRXOP01	PROPERTIES
SYSXSRA3	XSRCOMPONENT	XSRXCC01	COMPONENT
SYSXSRA4	XSRPROPERTY	XSRXCP01	PROPERTIES

Note: Index field is in descending order

SQL statements allowed on the catalog

Certain SQL statements can be used to change the value of certain options for existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables.

Table 151. SQL statements that can be used to change existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables

SQL statement	Index	Allowable clauses and usage notes
ALTER INDEX	IBM-defined	Only these clauses are allowed: CLOSE COPY FREEPAGE GBPCACHE NOT PADDED PADDED PCTFREE PIECESIZE BUFFERPOOL COMPRESS YES Any partitioning clause
		You cannot alter the GBPCACHE value for indexes DSNDX01, DSNDX02, and DSNDX03, which are on catalog table SYSIBM.SYSINDEXES.

Table 151. SQL statements that can be used to change existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables (continued)

SQL statement	Index	Allowable clauses and usage notes
ALTER INDEX	User-created	All clauses are allowed, except for BUFFERPOOL.
ALTER SEQUENCE		<p>The only clause allowed is MAXVALUE.</p> <p>You can only change the MAXVALUE value of the catalog sequence DSNSEQ_IMPLICITDB. The only value specific must be an integer between 1 and 60000, inclusive.</p>
ALTER TABLE		The only clause allowed is DATA CAPTURE CHANGES.
ALTER TABLESPACE		<p>Only these clauses are allowed:</p> <p>CLOSE FREEPAGE GBPCACHE LOCKMAX MAXROWS PCTFREE TRACKMOD</p> <p>You cannot alter the GBPCACHE or MAXROWS value of some catalog table spaces. Do not specify GBPCACHE for the following table spaces:</p> <ul style="list-style-type: none"> • DSNDB06.SYSDBASE • DSNDB06.SYSDBAUT • DSNDB06.SYSPKAGE • DSNDB06.SYSPLAN • DSNDB06.SYSUSER (exception: the attribute can be altered if authorization includes installation SYSADM authority.) <p>Do not specify MAXROWS or the LOCKSIZE keyword for the following table spaces:</p> <ul style="list-style-type: none"> • DSNDB06.SYSDBASE • DSNDB06.SYSDBAUT • DSNDB06.SYSGROUP • DSNDB06.SYSPLAN • DSNDB06.SYSSEQ • DSNDB06.SYSVIEWS <p>You can specify the LOCKSIZE keyword on the ALTER TABLESPACE statement for any catalog table spaces that are not LOB table spaces and that do not contain links.</p> <p>For DSNDB06.SYSSEQ, MAXROW can be specified only with a value of 1.</p>

Table 151. SQL statements that can be used to change existing catalog indexes, sequences, and table spaces, or to add indexes to any of the catalog tables (continued)

SQL statement	Index	Allowable clauses and usage notes
CREATE INDEX	User-created	<p>All clauses are allowed, except for:</p> <p>CLOSE YES</p> <p>CLUSTER</p> <p>UNIQUE</p> <p>DEFER YES (only on tables SYSINDEXES, SYSINDEXPART, and SYSKEYS)</p> <p>COMPRESS YES</p> <p>Any partitioning clause</p> <p>Indexes that are created with <i>key-expressions</i> are not allowed on the catalog.</p> <p>The only value allowed for BUFFERPOOL is BP0.</p> <p>You can create up to 500 indexes on the catalog.</p>
DROP INDEX	User-created	The statement has no clauses.

Reorganizing the catalog

The REORG TABLESPACE utility can be run on all the table spaces in the catalog database (DSNDB06) to reclaim unused or wasted space, which can affect performance.

The utility observes the PCTFREE and FREEPAGE values specified in the ALTER INDEX statement for all the catalog indexes and the following table spaces:

- DSNDB06.SYSCOPY
- DSNDB06.SYSDDF
- DSNDB06.SYSGPAUT
- DSNDB06.SYSGRTNS
- DSNDB06.SYSHIST
- DSNDB06.SYSJAVA
- DSNDB06.SYSJAUXA
- DSNDB06.SYSJAUXB
- DSNDB06.SYSOBJ
- DSNDB06.SYSPKAGE
- DSNDB06.SYSSEQ
- DSNDB06.SYSSEQ2
- DSNDB06.SYSSTR
- DSNDB06.SYSSTATS
- DSNDB06.SYSUSER
- DSNDB01.SCT02
- DSNDB01.SPT01

For details on running REORG TABLESPACE, see *DB2 Utility Guide and Reference*.

New and changed catalog tables

This release of DB2 for z/OS includes many new catalog tables as well as many catalog tables that have new or changed content.

Descriptions of the following catalog tables have been added:

- SYSIBM.SYSCONTEXT
- SYSIBM.SYSCTXTTRUSTATTRS
- SYSIBM.SYSCONTEXTAUTHIDS
- SYSIBM.SYSDEPENDENCIES
- SYSIBM.SYSENVIRONMENT
- SYSIBM.SYSINDEXSPACESTATS
- SYSIBM.SYSJAVAPATHS
- SYSIBM.SYSKEYTARGETS
- SYSIBM.SYSKEYTARGETS_HIST
- SYSIBM.SYSKEYTARGETSTATS
- SYSIBM.SYSKEYTGTDIST
- SYSIBM.SYSKEYTGTDIST_HIST
- SYSIBM.SYSKEYTGTDISTSTATS
- SYSIBM.SYSOBJROLEDEP
- SYSIBM.SYSROLES
- SYSIBM.SYSROUTINESTEXT
- SYSIBM.SYSTABLESPACESTATS
- SYSIBM.SYSXMLRELS
- SYSIBM.SYSXMLSTRINGS

The catalog tables that are listed in the following table have new or revised columns, column values, or column descriptions to support the new function in this release of DB2 for z/OS.

Table 152. Tables with new or revised columns, column values, or column descriptions

Table name	New column	Revised column
IPLIST		IPADDR
IPNAMES		IPADDR USERNAMES SECURITY_OUT
LOCATIONS	TRUSTED SECURE	
SYSAUXRELS	RELCREATED	TBOWNER AUXTBOWNER
SYSCHECKDEP		TBOWNER
SYSCHECKS	RELCREATED	TBOWNER
SYSCHECKS2	RELCREATED	TBOWNER
SYSCOLAUTH	GRANTORTYPE	GRANTEETYPE CREATOR
SYSOLDIST	QUANTILENO LOWVALUE HIGHVALUE	TYPE CARDF FREQUENCY TBOWNER

Table 152. Tables with new or revised columns, column values, or column descriptions (continued)

Table name	New column	Revised column
SYSOLDIST_HIST	QUANTILENO LOWVALUE HIGHVALUE	TYPE CARDF FREQUENCYF TBOWNER
SYSOLDISTSTATS	QUANTILENO LOWVALUE HIGHVALUE	TYPE CARDF FREQUENCYF TBOWNER
SYSOLSTATS		TBOWNER HIGHKEY HIGH2KEY LOWKEY LOW2KEY
SYSOLUMNS	RELCREATED	COLTYPE LENGTH HIGH2KEY LOW2KEY LENGTH2 DEFAULT DEFAULTVALUE TBOWNER
SYSOLUMNS_HIST		TBCREATOR COLTYPE LENGTH LENGTH2 TBOWNER
SYSCONSTDEP	DTBOWNER OWNERTYPE	DTBCREATOR
SYSOCOPY	LOGGED TTYPE INSTANCE RELCREATED	STYPE ICTYPE RELCREATED TIMESTAMP PIT_RBA
SYSDATABASE	IMPLICIT CREATORTYPE RELCREATED	
SYSDATATYPES	OWNERTYPE RELCREATED	CREATEDBY
SYSDBAUTH	GRANTEETYPE GRANTORTYPE	
SYSDBRM	PLCREATORTYPE RELCREATED	
SYSINDEXES	KEYTARGET_COUNT UNIQUE_COUNT IX_EXTENSION_TYPE COMPRESS OWNER OWNERTYPE DATAPEATFACTORF ENVID	PGSIZE TBCREATOR UNIQUERULE CLUSTERRATIOF CREATOR

Table 152. Tables with new or revised columns, column values, or column descriptions (continued)

Table name	New column	Revised column
SYSINDEXES_HIST	DATAPEATFACTORF	TBCREATOR CREATOR
SYSINDEXPART		IXCREATOR LEAFDIST FAROFFPOSF NEAROFFPOSF
SYSINDEXPART_HIST		IXCREATOR
SYSINDEXSTATS	DATAPEATFACTORF	OWNER
SYSINDEXSTATS_HIST	DATAPEATFACTORF	OWNER
SYSJAROBJECTS	OWNERTYPE	
SYSKEYS		COLSEQ ORDERING
SYSPACKAGE	OWNERTYPE ROUNDING	TYPE REOPTVAR
SYSPACKAUTH	GRANTORTYPE	GRANTEETYPE
SYSPACKDEP	DOWNERTYPE	BTYPE DTYPE BQUALIFIER
SYSPACKSTMT		SEQNO
SYSPARMS	VERSION OWNERTYPE	ROWTYPE PARMNAME CCSID
SYSPLAN	CREATORTYPE ROUNDING	REOPTVAR
SYSPLANAUTH	GRANTEETYPE GRANTORTYPE	
SYSPLANDEP		BTYPE BCREATOR
SYSRELS	RELCREATED	CREATOR REFTBCREATOR IXOWNER
SYSRESAUTH	GRANTEETYPE GRANTORTYPE	
SYSROUTINEAUTH	GRANTORTYPE	GRANTEETYPE
SYSROUTINES	VERSION CONTOKEN ACTIVE DEBUG_MODE TEXT_ENVID TEXT_ROWID TEXT OWNERTYPE PARAMETER_ VARCHARFORM RELCREATED PACKAGEPATH	ORIGIN FENCED CREATEDBY WLMENVIRONMENT EXTERNAL_SECURITY
SYSSCHEMAAUTH	GRANTEETYPE GRANTORTYPE	

Table 152. Tables with new or revised columns, column values, or column descriptions (continued)

Table name	New column	Revised column
SYSSEQUENCEAUTH	GRANTORTYPE	GRANTEETYPE
SYSSEQUENCES	OWNERTYPE RELCREATED	SEQTYPE CREATEDBY
SYSSEQUENCESDEP	DOWNER DOWNERTYPE	DTYPE
SYSSTMT	PLCREATORTYPE	
SYSSTOGROUP	CREATORTYPE DATACLAS MGMTCLAS STORCLAS RELCREATED	
SYSNONONYMS	CREATORTYPE RELCREATED	TBCREATOR
SYSTABAUTH	GRANTORTYPE	GRANTEETYPE SCREATOR TCREATOR
SYSTABCONST	CREATORTYPE RELCREATED	TBCREATOR IXOWNER
SYSTABLEPART	FORMAT REORG_LR_TS RELCREATED	IXCREATOR
SYSTABLES	APPEND OWNER OWNERTYPE RELCREATED	DBID OBID TYPE CREATOR TBCREATOR
SYSTABLES_HIST		CREATOR
SYSTABLESPACE	MAXPARTITIONS CREATORTYPE INSTANCE CLONE RELCREATED	LOG TYPE
SYSTABSTATS		OWNER
SYSTABSTATS_HIST		OWNER
SYSTRIGGERS	OWNERTYPE ENVID RELCREATED	OWNER CREATEDBY TRIGTIME TBOWNER TBNAME
SYSUSERAUTH	GRANTEETYPE GRANTORTYPE DEBUGSESSIONAUTH	
SYSVIEWDEP	DOWNER OWNERTYPE	DCREATOR
SYSVIEWS	OWNER OWNERTYPE RELCREATED	CREATOR
SYSVOLUMES	RELCREATED	

Table 152. Tables with new or revised columns, column values, or column descriptions (continued)

Table name	New column	Revised column
USERNAMES		TYPE

SYSIBM.IPLIST table

The SYSIBM.IPLIST table allows multiple IP addresses to be specified for a given LOCATION.

Insert rows into this table when you want to define a remote DB2 data sharing group. The same value for the IPADDR column cannot appear in both the SYSIBM.IPNAMES table and the SYSIBM.IPLIST table. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LINKNAME	VARCHAR(24) NOT NULL	This column is associated with the value specified in the LINKNAME column in the SYSIBM.LOCATIONS table and the SYSIBM.IPNAMES table. The values of the other columns in the SYSIBM.IPNAMES table apply to the server identified by the LINKNAME column in this row.	G
IPADDR	VARCHAR(254) NOT NULL	<p>This column contains an IPv4 or IPv6 address, or domain name of a remote TCP/IP host of the server. If WLM Domain Name Server workload balancing is used, this column must contain the member specific domain name. If Dynamic VIPA workload balancing is used, this column must contain the member specific Dynamic VIPA address. The IPADDR column must be specified as follows:</p> <ul style="list-style-type: none">• An IPv4 address must be left justified and is represented as a dotted decimal address. For example, '123.456.78.912' would be interpreted as an IPv4 address.• An IPv6 address must be left justified and is represented as a colon hexadecimal address. An example of an IPv6 address is '2001:0DB8:0000:0000:0008:0800:200C:417A', which can also be expressed in compressed form as '2001:DB8::8:800:200C:417A'.• A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is 'stlmvs1.svl.ibm.com'.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.IPNAMES table

The SYSIBM.IPNAMES table defines the remote DRDA servers DB2 can access using TCP/IP.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LINKNAME	VARCHAR(24) NOT NULL	The value specified in this column must match the value specified in the LINKNAME column of the associated row in SYSIBM.LOCATIONS.	G
SECURITY_OUT	CHAR(1) NOT NULL WITH DEFAULT 'A'	<p>This column defines the DRDA security option that is used when local DB2 SQL applications connect to any remote server associated with this TCP/IP host:</p> <p>A The option is “already verified”. Outbound connection requests contain an authorization ID and no password. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column.</p> <p>The authorization ID is not encrypted when it is sent to the partner. For encryption, refer to 'D'.</p> <p>D The option is “userid and security-sensitive data encryption”. Outbound connection requests contain an authorization ID and no password. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column.</p> <p>This option indicates that the userid and security-sensitive data are to be encrypted. For non-encryption, refer to 'A'.</p> <p>E The option is “userid, password, and security-sensitive data encryption”. Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table. The USERNAMES column must specify 'O'.</p> <p>This option indicates that the userid, password, and security-sensitive data are to be encrypted. For non-security-sensitive data encryption, refer to 'P'.</p> <p>If the applications connect to any remote server as trusted, the USERNAMES column must specify 'O' or 'S'.</p>	G

Column name	Data type	Description	Use
SECURITY_OUT (continued)		<p>P The option is “password”. Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table. The USERNAMES column must specify 'O'.</p> <p>This option indicates that the userid and the password are to be encrypted if cryptographic services are available at the requester and if the server supports encryption. Otherwise, the userid and the password are sent to the partner in clear text. For security-sensitive data encryption, see 'E'.</p> <p>If the applications connect to any remote server as trusted, the USERNAMES column must specify 'O' or 'S'.</p> <p>R The option is “RACF PassTicket”. Outbound connection requests contain a userid and a RACF PassTicket. The value specified in the LINKNAME column is used as the RACF PassTicket application name for the remote server.</p> <p>The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column.</p> <p>The authorization ID is not encrypted when it is sent to the partner.</p>	
USERNAMES	CHAR(1) NOT NULL WITH DEFAULT	<p>This column controls outbound authorization ID translation. Outbound translation is performed when an authorization ID is sent by DB2 to a remote server.</p> <p>O An outbound ID is subject to translation. Rows in the SYSIBM.USERNAMES table are used to perform ID translation.</p> <p>No translation or “come from” checking is performed on inbound IDs.</p> <p>S Row in the SYSIBM.USERNAMES table is used to obtain the system AUTHID used to establish a trusted connection.</p> <p>blank No translation occurs.</p>	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

Column name	Data type	Description	Use
IPADDR	VARCHAR(254) NOT NULL WITH DEFAULT	<p>This column contains an IPv4 or IPv6 address, or domain name of a remote TCP/IP host. The IPADDR column must be specified as follows:</p> <ul style="list-style-type: none"> • An IPv4 address must be left justified and is represented as a dotted decimal address. For example, '123.456.78.91' would be interpreted as an IPv4 address. • An IPv6 address must be left justified and is represented as a colon hexadecimal address. An example of an IPv6 address is '2001:0DB8:0000:0000:0008:0800:200C:417A', which can also be expressed in compressed form as '2001:DB8::8:800:200C:417A'. • A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is 'stlmvs1.svl.ibm.com'. 	G

SYSIBM.LOCATIONS table

The SYSIBM.LOCATIONS table contains a row for every accessible remote server. The row associates a LOCATION name with the TCP/IP or SNA network attributes for the remote server. Requesters are not defined in this table.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	A unique location name for the accessible server. This is the name by which the remote server is known to local DB2 SQL applications.	G
LINKNAME	VARCHAR(24) NOT NULL	Identifies the VTAM® or TCP/IP attributes associated with this location. For any LINKNAME specified, one or both of the following statements must be true: <ul style="list-style-type: none"> A row exists in SYSIBM.LUNAMES whose LUNAME matches the value specified in the SYSIBM.LOCATIONS LINKNAME column. This row specifies the VTAM communication attributes for the remote location. A row exists in SYSIBM.IPNAMES whose LINKNAME matches the value specified in the SYSIBM.LOCATIONS LINKNAME column. This row specifies the TCP/IP communication attributes for the remote location. 	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
PORT	VARCHAR(96) NOT NULL WITH DEFAULT	TCP/IP is used for outbound DRDA connections when the following statement is true: <ul style="list-style-type: none"> A row exists in SYSIBM.IPNAMES, where the LINKNAME column matches the value specified in the SYSIBM.LOCATIONS LINKNAME column. <p>If the above mentioned row is found, the value of the PORT column is interpreted as follows:</p> <ul style="list-style-type: none"> If PORT is blank, the default DRDA port (446) is used. If PORT is nonblank, the value specified for PORT can take one of two forms: <ul style="list-style-type: none"> If the value in PORT is left justified with 1-5 numeric characters, the value is assumed to be the TCP/IP port number of the remote database server. Any other value is assumed to be a TCP/IP service name, which can be converted to a TCP/IP port number using the TCP/IP getservbyname socket call. TCP/IP service names are not case sensitive. 	G

Column name	Data type	Description	Use
TPN	VARCHAR(192) NOT NULL WITH DEFAULT	Used only when the local DB2 begins an SNA conversation with another server. When used, TPN indicates the SNA LU 6.2 transaction program name (TPN) that will allocate the conversation. A length of zero for the column indicates the default TPN. For DRDA conversations, this is the DRDA default, which is X'07F6C4C2'. For DB2 private protocol conversations, this column is not used. When the server is DB2 Server for VSE & VM, TPN should contain the resource ID of that machine.	G
DBALIAS	VARCHAR(128) NOT NULL	Database alias. The name associated with the server. This name is used to access a remote database server. If DBALIAS is blank, the location name is used to access the remote database server. This column does not change the name of any database objects sent to the remote site that contains the location qualifier.	G
TRUSTED	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates whether the connection to the remote server can be trusted. This is restricted to TCP/IP only. This column is ignored for connections using SNA. Y Location is trusted. Access to the remote location requires trusted context defined at the remote location. N Location is not trusted.	G
SECURE	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates the use of the Secure Socket Layer (SSL) protocol for outbound DRDA connections when local DB2 applications connects to the remote database server using TCP/IP. Y Indicates a secure connection using SSL is required for the outbound DRDA connection. N Indicates a secure connection is not required for the outbound DRDA connection.	G

SYSIBM.LULIST table

The SYSIBM.LULIST table allows multiple LU names to be specified for a given LOCATION.

Insert rows into this table when you want to define a remote DB2 data sharing group. The same value for LUNAME column cannot appear in both the SYSIBM.LUNAMES table and the SYSIBM.LULIST table. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LINKNAME	VARCHAR(24) NOT NULL	The value of the LINKNAME column in the SYSIBM.LOCATIONS table with which this row is associated. This is also the value of the LUNAME column in the SYSIBM.LUNAMES table. The values of the other columns in the SYSIBM.LUNAMES row apply to the LU identified by the LUNAME column in this row of SYSIBM.LULIST.	G
LUNAME	VARCHAR(24) NOT NULL	The VTAM logical unit name (LUNAME) of the remote database system. This LUNAME must not exist in the LUNAME column of SYSIBM.LUNAMES.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.LUMODES table

Each row of the SYSIBM.LUMODES table provides VTAM with conversation limits for a specific combination of LUNAME and MODENAME. The table is accessed only during the initial conversation limit negotiation between DB2 and a remote LU. This negotiation is called *change-number-of-sessions* (CNOS) processing.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LUNAME	VARCHAR(24) NOT NULL	LU name of the server involved in the CNOS processing.	G
MODENAME	VARCHAR(24) NOT NULL	Name of a logon mode description in the VTAM logon mode table.	G
CONVLIMIT	SMALLINT NOT NULL	Maximum number of active conversations between the local DB2 and the other system for this mode. Used to override the number in the DSESLIM parameter of the VTAM APPL definition statement for this mode.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.LUNAMES table

The SYSIBM.LUNAMES table must contain a row for each remote SNA client or server that communicates with DB2.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
LUNAME	VARCHAR(24) NOT NULL	Name of the LU for one or more accessible systems. A blank string indicates the row applies to clients whose LU name is not specifically defined in this table. All other column values for a given row in this table are for clients and servers associated with the row's LU name.	G
SYSMODENAME	VARCHAR(24) NOT NULL WITH DEFAULT	Mode used to establish inter-system conversations. A blank indicates the default mode IBMDB2LM (DB2 private protocol access and for collecting sysplex balancing information from remote data sharing groups). If private protocols are used to access a remote DB2 LU or if the remote LU is a member of a DB2 data sharing group, use a separate mode other than the default mode.	G
SECURITY_IN	CHAR(1) NOT NULL WITH DEFAULT 'A'	This column defines the security options that are accepted by this DB2 when an SNA client connects to DB2: V The option is "verify". An incoming connection request must include one of the following: a userid and password, a userid and RACF PassTicket, or a Kerberos security ticket. A The option is "already verified". A request does not need a password, although a password is checked if it is sent. With this option, an incoming connection request is accepted if it includes any of the following: a userid, a userid and password, a userid and RACF PassTicket, or a Kerberos security ticket. If the USERNAMES column contains 'T' or 'B', RACF is not invoked to validate incoming connection requests that contain only a userid.	G

Column name	Data type	Description	Use
SECURITY_OUT	CHAR(1) NOT NULL WITH DEFAULT 'A'	<p>This column defines the security option that is used when local DB2 SQL applications connect to any remote server associated with this LUNAME:</p> <p>A The option is “already verified”. Outbound connection requests contain an authorization ID and no password.</p> <p>The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column.</p> <p>R The option is “RACF PassTicket”. Outbound connection requests contain a userid and a RACF PassTicket. The server's LU name is used as the RACF PassTicket application name.</p> <p>The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column.</p> <p>P The option is “password”. Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table or RACF, depending upon the value specified in the ENCRYPTPSWDS column.</p> <p>The USERNAMES column must specify 'B' or 'O'.</p>	G
ENCRYPTPSWDS	CHAR(1) NOT NULL WITH DEFAULT 'N'	<p>This column only applies to DB2 for z/OS partners. It is provided to support connectivity to prior releases of DB2 that are unable to support RACF PassTickets.</p> <p>For connections between DB2 Version 5 and later, use the SECURITY_OUT='R' option instead of the ENCRYPTPSWDS='Y' option.</p> <p>N No, passwords are not in internal RACF encrypted format. This is the default.</p> <p>Y Yes for outbound requests, the encrypted password is extracted from RACF and sent to the server. For inbound requests, the password is treated as encrypted.</p>	G
MODESELECT	CHAR(1) NOT NULL WITH DEFAULT 'N'	<p>Whether to use the SYBIBM.MODESELECT table:</p> <p>N Use default modes: IBMDB2LM (for DB2 private protocol) and IBMRDB (for DRDA).</p> <p>Y Searches SYSIBM.MODESELECT for appropriate mode name.</p>	G

Column name	Data type	Description	Use
USERNAMES	CHAR(1) NOT NULL WITH DEFAULT	<p>This column controls inbound and outbound authorization ID translation, and “come from” checking.</p> <p>Inbound translation and “come from” checking are performed when an authorization ID is received from a remote client.</p> <p>Outbound translation is performed when an authorization ID is sent by DB2 to a remote server.</p> <p>When 'I', 'O', or 'B' is specified in this column, rows in the SYSIBM.USERNAMES table are used to perform ID translation.</p> <p>I An inbound ID is subject to translation and “come from” checking.</p> <p> No translation is performed on outbound IDs.</p> <p>O No translation or “come from” checking is performed on inbound IDs.</p> <p> An outbound ID is subject to translation.</p> <p>B An inbound ID is subject to translation and “come from” checking.</p> <p> An outbound ID is subject to translation.</p> <p>blank No translation occurs.</p>	G
GENERIC	CHAR(1) NOT NULL WITH DEFAULT 'N'	<p>Indicates whether DB2 should use its real LU name or generic LU name to identify itself to the partner LU, which is identified by this row.</p> <p>N The real VTAM LU name of this DB2 subsystem</p> <p>Y The VTAM generic LU name of this DB2 subsystem</p>	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.MODESELECT table

The SYSIBM.MODESELECT table associates a mode name with any conversation created to support an outgoing SQL request. Each row represents one or more combinations of LUNAME, authorization ID, and application plan name.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
AUTHID	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the SQL request. Blank (the default) indicates that the MODENAME specified for the row is to apply to all authorization IDs.	G
PLANNAME	VARCHAR(24) NOT NULL WITH DEFAULT	Plan name associated with the SQL request. Blank (the default) indicates that the MODENAME specified for the row is to apply to all plan names.	G
LUNAME	VARCHAR(24) NOT NULL	LU name associated with the SQL request.	G
MODENAME	VARCHAR(24) NOT NULL	Name of the logon mode in the VTAM logon mode table to be used in support of the outgoing SQL request. If blank, IBMDB2LM is used for DB2 private protocol connections and IBMRDB is used for DRDA connections.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSAUXRELS table

The SYSIBM.SYSAUXRELS table contains one row for each auxiliary table created for a LOB column. A base table space that is partitioned must have one auxiliary table for each partition of each LOB column.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the base table.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the base table.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the LOB column in the base table.	G
PARTITION	SMALLINT NOT NULL	Partition number if the base table space is partitioned. Otherwise, the value is 0.	G
AXTBOWNER	VARCHAR(128) NOT NULL	The schema of the auxiliary table.	G
AUXTBNAME	VARCHAR(128) NOT NULL	Name of the auxiliary table.	G
AUXRELOBID	INTEGER NOT NULL	Internal identifier of the relationship between the base table and the auxiliary table.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
RELCREATED 	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSCHECKDEP table

The SYSIBM.SYSCHECKDEP table contains one row for each reference to a column in a check constraint.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table on which the check constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the check constraint is defined.	G
CHECKNAME	VARCHAR(128) NOT NULL	Name of the check constraint.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column that the check constraint refers to.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSCHECKS table

The SYSIBM.SYSCHECKS table contains one row for each check constraint.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table on which the check constraint is defined.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the creator of the check constraint.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database for the check constraint.	S
OBID	SMALLINT NOT NULL	Internal identifier of the check constraint.	S
TIMESTAMP	TIMESTAMP NOT NULL	Time when the check constraint was created.	G
RBA	CHAR(6) FOR BIT DATA NOT NULL WITH DEFAULT	The log RBA when the check constraint was created.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the check constraint is defined.	G
CHECKNAME	VARCHAR(128) NOT NULL	Check constraint name.	G
CHECKCONDITION	VARCHAR(7400) NOT NULL	Text of the check constraint.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSCHECKS2 table

The SYSIBM.SYSCHECKS2 table contains one row for each check constraint for catalog tables created in or after Version 7. Check constraints for catalog tables created before Version 7 are not included in this table.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table on which the check constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the check constraint is defined.	G
CHECKNAME	VARCHAR(128) NOT NULL	Check constraint name.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL	SQL path at the time the check constraint was created. The path is used to resolve unqualified cast function names that are used in the constraint definition.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSCOLAUTH table

The SYSIBM.SYSCOLAUTH table records the UPDATE or REFERENCES privileges that are held by users on individual columns of a table or view.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID or role of the user who granted the privileges. Could also be PUBLIC or PUBLIC followed by an asterisk. (PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see <i>DB2 Administration Guide</i> .).	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID or role of the user who holds the privilege or the name of an application plan or package that uses the privilege. PUBLIC for a grant to PUBLIC. PUBLIC followed by an asterisk for a grant to PUBLIC AT ALL LOCATIONS.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: blank An authorization ID L Role P An application plan or a package. The grantee is a package if COLLID is not blank.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the table or view on which the update privilege is held.	G
TNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
COLNAME	VARCHAR(128) NOT NULL	Name of the column to which the UPDATE privilege applies.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
	VARCHAR(128) NOT NULL WITH DEFAULT	Not used	N
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	If GRANTEE is a package, its collection name. Otherwise, the value is blank.	G

Column name	Data type	Description	Use
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	If GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, the value is blank.	S
PRIVILEGE	CHAR(1) NOT NULL WITH DEFAULT	Indicates which privilege this row describes: R Row pertains to the REFERENCES privilege. blank Row pertains to the UPDATE privilege.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: L Role blank Authorization ID that is not a role	G

SYSIBM.SYSCOLDIST table

The SYSIBM.SYSCOLDIST table contains one or more rows for the first key column of an index key.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
	SMALLINT NOT NULL	Not used	N
STATTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column. If NUMCOLUMNS is greater than 1, this name identifies the first column name of the set of columns associated with the statistics.	G
COLVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: C Cardinality F Frequent value H Histogram Statistics N Non-padded frequent value	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	For TYPE='C', the number of distinct values for the column group. For TYPE='H', the number of distinct values for the column group in a quantile indicated by QUANTILENO.	S
COLGROUPCOLNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS. This is an updatable column.	S
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G

Column name	Data type	Description	Use
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	<p>Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%.</p> <p>When TYPE='H', this is the percentage of rows in table which falls in the quantile indicated by QUANTILENO whose range is limited by [LOWVALUE, HIGHVALUE].</p> <p>Statistics are not collected for an index on a ROWID column.</p>	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	<p>Ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable.</p>	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	<p>For TYPE='H', this is the lower bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.</p>	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	<p>For TYPE='H', this is the higher bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.</p>	G

SYSIBM.SYSCOLDISTSTATS table

The SYSIBM.SYSCOLDISTSTATS table contains zero or more rows per partition for the first key column of a partitioning index or a data-partitioned secondary index.

No row is inserted if the index is a non-partitioned index. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
	SMALLINT NOT NULL	Not used	N
STATTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
PARTITION	SMALLINT NOT NULL	Partition number for the table space that contains the table in which the column is defined.	G
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column. If NUMCOLUMNS is greater than 1, this name identifies the first column name of the set of columns associated with the statistics.	G
COLVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: C Cardinality F Frequent value H Histogram statistics N Non-padded frequent value	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	If TYPE is C, the value is the number of distinct values for the column group. If TYPE is N or TYPE is F, the value is the number of rows or keys in the partition for which the FREQUENCYF value applies. If TYPE is H, the number of distinct values for the column group in a quantile indicated by QUANTILENO.	S
COLGROUPCOLNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS. This is an updatable column.	S

Column name	Data type	Description	Use
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	<p>Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%.</p> <p>When TYPE='H', this is the percentage of rows in table which falls in the quantile indicated by QUANTILENO whose range is limited by [LOWVALUE, HIGHVALUE].</p> <p>Statistics are not collected for an index on a ROWID column.</p>	G
	VARCHAR(1000) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only	I
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	Ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the lower bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the higher bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G

SYSIBM.SYSCOLDIST_HIST table

The SYSIBM.SYSCOLDIST_HIST table contains rows from SYSCOLDIST.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
TBOWNER	VARCHAR(128) NOT NULL	The schema of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column. If NUMCOLUMNS is greater than 1, this name identifies the first column name of the set of columns associated with the statistics.	G
COLVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: C Cardinality F Frequent value H Histogram Statistics N Non-padded frequent value	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	When TYPE='C', this is the number of distinct values for the column group. When TYPE='H', this is the number of distinct values for the column group in a quantile indicated by QUANTILENO. The value is -1 if statistics have not been gathered.	S
COLGROUPCOLNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS.	S
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G

Column name	Data type	Description	Use
FREQUENCYF	FLOAT(8) NOT NULL DEFAULT -1	<p>Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of '1' indicates 100%. A value of '.153' indicates 15.3%.</p> <p>When TYPE='H', this is the percentage of rows in table which falls in the quantile indicated by QUANTILENO whose range is limited by [LOWVALUE, HIGHVALUE].</p> <p>Statistics are not collected for an index on a ROWID column. The value is -1 if statistics have not been gathered.</p>	G
IBMREQD	CHAR(1) NOT NULL DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	Ordinary sequence number of a quantile in the whole consecutive value range, from low to high. This column is not updatable.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the lower bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	For TYPE='H', this is the higher bound for the quantile indicated by QUANTILENO. Not used if TYPE is not 'H'. This column is not updatable.	G

SYSIBM.SYSCOLSTATS table

The SYSIBM.SYSCOLSTATS table contains partition statistics for selected columns. For each column, a row exists for each partition in the table.

Rows are inserted when RUNSTATS collects either indexed column statistics or non-indexed column statistics for a partitioned table space. No row is inserted if the table space is nonpartitioned. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
HIGHKEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Highest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Second highest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
LOWKEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Lowest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Second lowest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column, a node ID column, or a column of an XML table. If the column has a non-character data type, the data might not be printable. If the partition is empty, the value is a string of length 0.	S
COLCARD	INTEGER NOT NULL	Number of distinct column values in the partition.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. If the value is '0001-01-02.00.00.00.000000', which indicates that an ALTER TABLE statement was executed to change the length of a VARCHAR column, RUNSTATS should be run to update the statistics before they are used.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
PARTITION	SMALLINT NOT NULL	Partition number for the table space that contains the table in which the column is defined.	G
TBOWNER	VARCHAR(128) NOT NULL	Schema or qualifier of the table that contains the column.	G

Column name	Data type	Description	Use
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
COLCARDATA	VARCHAR(1000) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only	I
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	<p>The type of statistics gathered:</p> <p>blank Statistics have not been collected or varchar column statistical values are padded.</p> <p>N Varchar column statistical values are not padded.</p> <p>This is an updatable column.</p>	G

SYSIBM.SYSCOLUMNS table

The SYSIBM.SYSCOLUMNS table contains one row for every column of each table and view.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view which contains the column.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table or view that contains the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of the column in the table or view; for example 4 (out of 10).	G

Column name	Data type	Description	Use
COLTYPE	CHAR(8) NOT NULL	<p>The type of the column specified in the definition of the column:</p> <p>INTEGER Large integer</p> <p>SMALLINT Small integer</p> <p>FLOAT Floating-point</p> <p>CHAR Fixed-length character string</p> <p>VARCHAR Varying-length character string</p> <p>LONGVAR Varying-length character string (for columns that were added before Version 9)</p> <p>DECIMAL Decimal</p> <p>GRAPHIC Fixed-length graphic string</p> <p>VARG Varying-length graphic string</p> <p>LONGVARG Varying-length graphic string (for columns that were added before Version 9)</p> <p>DATE Date</p> <p>TIME Time</p> <p>TIMESTAMP Timestamp</p> <p>BLOB Binary large object</p> <p>CLOB Character large object</p> <p>DBCLOB Double-byte character large object</p> <p>ROWID Row ID data type</p> <p>DISTINCT Distinct type</p> <p>XML XML data type</p> <p>BIGINT Big integer</p> <p>BINARY Fixed-length binary string</p> <p>VARBIN Varying-length binary string</p> <p>DECFLOAT Decimal floating point</p>	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	Length attribute of the column or, in the case of a decimal column, its precision. The number does not include the internal prefixes that are used to record the actual length and null state, where applicable. INTEGER 4 SMALLINT 2 BIGINT 8 FLOAT 4 or 8 CHAR Length of string VARCHAR Maximum length of string LONGVAR Maximum length of string (for columns that were added before Version 9) DECIMAL Precision of number DECFLOAT 8 or 16 GRAPHIC Number of DBCS characters VARGRAPHIC Maximum number of DBCS characters LONGVARG Maximum number of DBCS characters (for columns that were added before Version 9) BINARY Length of string VARBINARY Maximum length of string DATE 4 TIME 3 TIMESTAMP 10	G
LENGTH (continued)	SMALLINT NOT NULL	BLOB 4 - For a table, a field of length of 4 is stored in the base table. The maximum length of the LOB column is found in LENGTH2. CLOB 4 - For a table, a field of length of 4 is stored in the base table. The maximum length of the CLOB column is found in LENGTH2. DBCLOB 4 - For a table, a field of length of 4 is stored in the base table. The maximum length of the DBCLOB column is found in LENGTH2. ROWID 17 - The maximum length of the stored portion of the identifier. XML 6 DISTINCT The length of the source data type.	G
SCALE	SMALLINT NOT NULL	Scale of decimal data. Zero if not a decimal column.	G

Column name	Data type	Description	Use
NULLS	CHAR(1) NOT NULL	Whether the column can contain null values: N No Y Yes The value can be N for a view column that is derived from an expression or a function. Nevertheless, such a column allows nulls when an outer select list refers to it.	G
	INTEGER NOT NULL	Not used	N
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Second highest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable. If the table is empty, the value is a string of length 0. This is an updatable column.	S
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Second lowest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable. If the table is empty, the value is a string of length 0. This is an updatable column.	S
UPDATES	CHAR(1) NOT NULL	Whether the column can be updated: N No Y Yes The value is N if the column is: <ul style="list-style-type: none"> • Derived from a function or expression • A column with a row ID data type (or a distinct type based on a row ID type) • A read-only view 	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G

Column name	Data type	Description	Use
DEFAULT	CHAR(1) NOT NULL	<p>The contents of this column are meaningful only if the TYPE column for the associated SYSTABLES row indicates that this is for a table (T) or a created temporary table (G).</p> <p>Default indicator:</p> <p>A The column has a row ID data type (COLTYPE='ROWID') and the GENERATED ALWAYS attribute.</p> <p>B The column has a default value that depends on the data type of the column.</p> <p>Data type</p> <p>Default Value</p> <p>Numeric 0</p> <p>Fixed-length character or graphic string Blanks</p> <p>Fixed-length binary string Hexadecimal zeros</p> <p>Varying-length string A string length of 0</p> <p>Date The current date</p> <p>Time The current time</p> <p>Timestamp The current timestamp</p> <p>D The column has a row ID data type (COLTYPE='ROWID') and the GENERATED BY DEFAULT attribute.</p> <p>E The column is defined with the FOR EACH ROW ON UPDATE and GENERATED ALWAYS attributes.</p> <p>F The column is defined with the FOR EACH ROW ON UPDATE and GENERATED BY DEFAULT attributes.</p> <p>I The column is defined with the AS IDENTITY and GENERATED ALWAYS attributes.</p> <p>J The column is defined with the AS IDENTITY and GENERATED BY DEFAULT attributes.</p> <p>K The column is defined for the implicit DOCID column for a base table that contains XML data.</p> <p>L The column is defined with the AS SECURITY LABEL attribute.</p> <p>N The column has no default value.</p>	G

Column name	Data type	Description	Use
DEFAULT (continued)	CHAR(1) NOT NULL	Default indicator:	G
		S The column has a default value that is the value of the SQL authorization ID of the process at the time a default value is used.	
		U The column has a default value that is the value of the SESSION_USER special register at the time a default value is used.	
		Y If the NULLS column is Y, the column has a default value of null.	
		If the NULLS column is N, the default value depends on the data type of the column.	
		Data type	
		Default Value	
		Numeric	
		0	
		Fixed-length character string	
		Blanks	
		Fixed-length graphic string	
		Blanks	
		Fixed-length binary string	
		Hexadecimal blanks	
		Varying-length string	
		A string length of 0	
		Date The current date	
		Time The current time	
		Timestamp	
		The current timestamp	
		1 The column has a default value that is the string constant found in the DEFAULTVALUE column of this table row.	
		The column has a graphic data type and has a default value that is the graphic string found in the DEFAULTVALUE column of this table row.	
		2 The column has a default value that is the floating-point constant found in the DEFAULTVALUE column of this table row.	
		3 The column has a default value that is the decimal constant found in the DEFAULTVALUE column of this table row.	
		4 The column has a default value that is the integer constant found in the DEFAULTVALUE column of this table row.	
		5 The column has a default value that is the hexadecimal character string found in the DEFAULTVALUE column of this table row.	

Column name	Data type	Description	Use
DEFAULT (continued)		Default indicator:	G
	CHAR(1) NOT NULL	<p>6 The column has a default value that is the UX string found in the DEFAULTVALUE column of this table row.</p> <p>7 The column has a graphic data type and has a default value that is the character string constant found in the DEFAULTVALUE column of this table row.</p> <p>8 The column has a character data type and has a default value that is the graphic string constant found in the DEFAULTVALUE column of this table row.</p> <p>9 The column has a default value that is the DECFLOAT constant found in the DEFAULTVALUE column of this table row.</p>	
KEYSEQ	SMALLINT NOT NULL	The numeric position of the column within the primary key of the table. The value is 0 if it is not part of a primary key.	G
FOREIGNKEY	CHAR(1) NOT NULL	<p>Applies to character or CLOB columns, where it indicates the subtype of the data:</p> <p>B BIT data</p> <p>M MIXED data</p> <p>S SBCS data</p> <p>blank Indicates one of the following subtypes:</p> <ul style="list-style-type: none"> MIXED data if the encoding scheme is UNICODE, or if the encoding scheme is not UNICODE and the value of MIXED DATA on installation panel DSNTIPS is YES SBCS data if the encoding scheme is not UNICODE and the value of MIXED DATA on the installation panel DSNTIPS is NO. <p>For views defined prior to Version 7, subtype information is not available and the default (MIXED or SBCS) is used. This is an updatable column.</p>	G
FLDPROC	CHAR(1) NOT NULL	<p>Whether the column has a field procedure:</p> <p>N No</p> <p>Y Yes</p> <p>blank The column is for a view defined prior to Version 7. Views defined after Version 7 contain Y or N.</p>	G
LABEL	VARCHAR(90) NOT NULL	The column label provided by the user with a LABEL statement; otherwise, the value is an empty string.	G
STATTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. If the value is '0001-01-02.00.00.00.000000', which indicates that an ALTER TABLE statement was executed to change the length of a VARCHAR column, RUNSTATS should be run to update the statistics before they are used. This is an updatable column.	G

Column name	Data type	Description	Use
DEFAULTVALUE	VARCHAR(1536) NOT NULL WITH DEFAULT	<p>This field is meaningful only if the column being described is for a table (the TYPE column of the associated SYSTABLES row is T for table or G for created temporary table).</p> <p>When the DEFAULT column is 1, 2, 3, 4, 5, 6, 7, 8, or 9, this field contains the default value of the column.</p> <p>If the default value is a string constant or a hexadecimal constant (DEFAULT is 1, 5, 6, 7, or 8 respectively), the value is stored without delimiters.</p> <p>If the default value is a numeric constant (DEFAULT is 2, 3, 4, or 9), the value is stored as specified by the user, including sign and decimal point representation, or special constant values, as appropriate for the constant.</p> <p>When the DEFAULT column is S or U and the default value was specified when a new column was defined with the ALTER TABLE statement, this field contains the value of the CURRENT SQLID or SESSION_USER special register at the time the ALTER TABLE statement was executed. Remember that this default value applies only to rows that existed before the ALTER TABLE statement was executed.</p> <p>When the DEFAULT column is L and the column was added as a new column with the ALTER TABLE statement, this field contains the security label of the user at the time the ALTER TABLE statement was executed. Remember that this default value applies only to rows that existed before the ALTER TABLE statement was executed.</p>	G
COLCARDF	FLOAT NOT NULL WITH DEFAULT	Estimated number of distinct values in the column. For an indicator column, this is the number of LOBs that are not null and have a length greater than zero. The value is -1 if statistics have not been gathered. The value is -2 for the first column of an index of an auxiliary table. This is an updatable column.	S
COLSTATUS	CHAR(1) NOT NULL WITH DEFAULT	<p>Indicates the status of the definition of a column:</p> <p>I The definition is incomplete because a LOB table space, auxiliary table, or index on an auxiliary table has not been created for the column.</p> <p>blank The definition is complete.</p>	G
LENGTH2	INTEGER NOT NULL WITH DEFAULT	<p>Maximum length of the data retrieved from the column. Possible values are:</p> <p>0 Column is not a LOB or ROWID column</p> <p>40 For a ROWID column, the length of the returned value</p> <p>1 to 2,147,483,647 bytes For a LOB column, the maximum length</p>	G
DATATYPEID	INTEGER NOT NULL WITH DEFAULT	<p>For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type.</p> <p>If the column was created prior to Version 6, the value is 0.</p>	S

Column name	Data type	Description	Use
SOURCETYPEID	INTEGER NOT NULL WITH DEFAULT	For a built-in data type, 0. For a distinct type, the internal ID of the built-in data type upon which the distinct type is based. If the column was created prior to Version 6, the value is 0.	S
TYPESHEMA	VARCHAR(128) NOT NULL WITH DEFAULT 'SYSIBM'	If COLTYPE is 'DISTINCT', the schema of the distinct type. Otherwise, the value is 'SYSIBM'.	G
TYPENAME	VARCHAR(128) NOT NULL WITH DEFAULT	If COLTYPE is 'DISTINCT', the name of the distinct type. Otherwise, the value is the same as the value of the COLTYPE column. TYPENAME is set only for columns created in Version 6 or later. The value for columns created earlier is not filled in.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the column was created. The value is '0001-01-01.00.00.000000' if the column was created prior to migration to Version 6 or if the column is in a catalog table.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics gathered: blank Statistics have not been collected or varchar column statistical values are padded. N Varchar column statistical values are not padded. This is an updatable column.	G
PARTKEY_COLSEQ	SMALLINT NOT NULL WITH DEFAULT	The numeric position of the column within the partitioning key of the table. The value is 0 if it is not part of the partitioning key. This column is applicable only if the table uses table-controlled partitioning.	G
PARTKEY_ORDERING	CHAR(1) NOT NULL WITH DEFAULT	Order of the column in the partitioning key: A Ascending D Descending blank Column is not used as part of a partitioning key This column is applicable only if the table uses table-controlled partitioning.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when alter occurred.	G
CCSID	INTEGER NOT NULL WITH DEFAULT	CCSID of the column. 0 if the object was created prior to Version 8 or is not a string column	G
HIDDEN	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates whether the column is implicitly hidden: P Partially hidden. The column is implicitly hidden from SELECT *. N Not hidden. The column is visible to all SQL statements.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSCOLUMNS_HIST table

The SYSIBM.SYSCOLUMNS_HIST table contains rows from SYSCOLUMNS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view that contains the column.	G
TBCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table or view that contains the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of the column in the table or view. For example 4 (out of 10).	G

Column name	Data type	Description	Use
COLTYPE	CHAR(8) NOT NULL	<p>The type of the column specified in the definition of the column:</p> <p>INTEGER Large integer</p> <p>SMALLINT Small integer</p> <p>FLOAT Floating-point</p> <p>CHAR Fixed-length character string</p> <p>VARCHAR Varying-length character string</p> <p>LONGVAR Varying-length character string (for columns that were added before Version 9)</p> <p>DECIMAL Decimal</p> <p>GRAPHIC Fixed-length graphic string</p> <p>VARG Varying-length graphic string</p> <p>LONGVARG Varying-length graphic string (for columns that were added before Version 9)</p> <p>DATE Date</p> <p>TIME Time</p> <p>TIMESTAMP Timestamp</p> <p>BLOB Binary large object</p> <p>CLOB Character large object</p> <p>DBCLOB Double-byte character large object</p> <p>ROWID Row ID data type</p> <p>DISTINCT Distinct type</p> <p>XML XML data type</p> <p>BIGINT Big integer</p> <p>BINARY Fixed-length binary string</p> <p>VARBIN Varying-length binary string</p> <p>DECFLOAT Decimal floating point</p>	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	<p>Length attribute of the column or, in the case of a decimal column, its precision. The number does not include the internal prefixes that are used to record the actual length and null state, where applicable.</p> <p>INTEGER 4</p> <p>SMALLINT 2</p> <p>FLOAT 4 or 8</p> <p>CHAR Length of string</p> <p>VARCHAR Maximum length of string</p> <p>LONGVAR Maximum length of string (for columns that were added before Version 9)</p> <p>DECIMAL Precision of number</p> <p>GRAPHIC Number of DBCS characters</p> <p>VARGRAPHIC Maximum number of DBCS characters</p> <p>LONGVARG Maximum number of DBCS characters (for columns that were added before Version 9)</p> <p>DATE 4</p> <p>TIME 3</p> <p>TIMESTAMP 10</p> <p>BLOB 4 - The length of the field that is stored in the base table. The maximum length of the LOB column is found in LENGTH2.</p> <p>CLOB 4 - The length of the field that is stored in the base table. The maximum length of the CLOB</p> <p>DBCLOB 4 - The length of the field that is stored in the base table. The maximum length of the DBCLOB column is found in LENGTH2.</p> <p>ROWID 17 - The maximum length of the stored portion of the identifier.</p> <p>DISTINCT The length of the source data type.</p> <p>XML 6</p> <p>BIGINT 8</p> <p>BINARY The length of the string</p> <p>VARBINARY The maximum length of string</p> <p>DECFLOAT 8 or 16</p>	G
LENGTH2	INTEGER NOT NULL	<p>Maximum length of the data retrieved from the column.</p> <p>Possible values are:</p> <p>0 Column is not a LOB or ROWID column</p> <p>40 For a ROWID column, the length of the returned value</p> <p>1 to 2 147 483 647 bytes For a LOB column, the maximum length</p>	G

Column name	Data type	Description	Use
NULLS	CHAR(1) NOT NULL	Whether the column can contain null values: N No Y Yes	G
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Second highest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable.	S
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	Second lowest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. If the value is '0001-01-02.00.00.00.000000', which indicates that an ALTER TABLE statement was executed to change the length of a VARCHAR column, RUNSTATS should be run to update the statistics before they are used.	G
COLCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Estimated number of distinct values in the column. For an indicator column, this is the number of LOBs that are not null and have a length greater than zero. The value is -1 if statistics have not been gathered. The value is -2 for the first column of an index of an auxiliary table.	S
IBMREQD	CHAR(1) NOT NULL DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics gathered: blank Statistics have not been collected or varchar column statistical values are padded. N Varchar column statistical values are not padded. This is an updatable column.	G

SYSIBM.SYSCONSTDEP table

The SYSIBM.SYSCONSTDEP table records dependencies on check constraints or user-defined defaults for a column.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	Name of the object on which the dependency exists.	G
BSHEMA	VARCHAR(128) NOT NULL	Schema of the object on which the dependency exists.	G
BTYPE	CHAR(1) NOT NULL	Type of object on which the dependency exists: F Function instance	G
DTBNAME	VARCHAR(128) NOT NULL	Name of the table to which the dependency applies.	G
DTBCREATOR	VARCHAR(128) NOT NULL	The schema of the table to which the dependency applies.	G
DCONSTNAME	VARCHAR(128) NOT NULL	If DTYPE = 'C', the unqualified name of the check constraint. If DTYPE = 'D', a column name.	G
DTYPE	CHAR(1) NOT NULL	Type of object: C Check constraint D User-defined default constant	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DTBOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the table or a zero length string for tables that were created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID R Role	G

SYSIBM.SYSCONTEXT table

The SYSIBM.SYSCONTEXT table contains one row for each trusted context.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the trusted context.	G
CONTEXTID	INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY	Internal context ID.	G
DEFINER	VARCHAR(128) NOT NULL	Authorization ID or role that defined the trusted context.	G
DEFINERTYPE	CHAR(1) NOT NULL	The type of the definer: L Role blank Authorization ID	G
SYSTEMAUTHID	VARCHAR(128) NOT NULL	The DB2 primary authorization ID that is used to establish the connection. For remote requests, SYSTEMAUTHID is derived from the system user ID that is provided by an external entity, such as a middleware server. For local requests, SYSTEMAUTHID depends on one of the following sources of the address space: BATCH USER parameter on JOB statement RRSAF USER parameter on JOB statement or RACF user TSO TSO logon ID	G
DEFAULTROLE	VARCHAR(128) NOT NULL	Name of the trusted context default role.	G
OBJECTOWNERTYPE	CHAR(1) NOT NULL	Whether the ROLE AS OBJECT OWNER AND QUALIFIER clause is specified in the definition of this trusted context: L ROLE AS OBJECT OWNER AND QUALIFIER is specified. A role owns any object created in the trusted context. The role is used as the default for the CURRENT SCHEMA special register. The role is included in the SQL PATH. blank ROLE AS OBJECT OWNER is not specified. An authorization ID owns any object created in the trusted context.	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the trusted context is created.	G
ALTEREDTS	TIMESTAMP NOT NULL	The time when the trusted context is last altered.	G

Column name	Data type	Description	Use
ENABLED	CHAR(1) NOT NULL	The status of the trusted context: Y Enabled N Disabled	G
ALLOWPUBLIC	CHAR(1) NOT NULL	Whether the connection is allowed to be reused for PUBLIC: Y Connection reuse is allowed N Connection reuse is not allowed	G
AUTHENTICATE-PUBLIC	CHAR(1) NOT NULL	Whether authentication is required for PUBLIC when ALLOWPUBLIC is Y: Y Authentication token is required for PUBLIC. For local requests, the token is the password. For remote requests, the token can be a password, a RACF passticket, or a KERBEROS token N Authentication is not required	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
REMARKS	VARCHAR(762) NOT NULL	A character string that is provided using the COMMENT statement.	G
DEFAULT-SECURITYLABEL	VARCHAR(24) NOT NULL	Name of the context default RACF security label.	G

SYSIBM.SYSCONTEXTAUTHIDS table

The SYSIBM.SYSCONTEXTAUTHIDS table contains one row for each authorization ID with which the trusted context can be used.

Column name	Data type	Description	Use
CONTEXTID	INTEGER NOT NULL	The internal trusted context ID.	G
AUTHID	VARCHAR(128) NOT NULL	The primary authorization ID that can reuse a connection or the RACF profile name that contains the primary authorization IDs that are permitted to use the connection in the identified trusted context. The AUTHID value starts with and is a profile name.	G
AUTHENTICATE	CHAR(1) NOT NULL	Whether authentication is required for the authorization ID in the AUTHID column: Y Authentication token is required for the authorization ID. For local requests, the token is the password. For remote requests, the token can be a password, a RACF passticket, or a Kerberos token N Authentication is not required	G
ROLE	VARCHAR(128) NOT NULL	The role for the authorization ID in the AUTHID column. The role supersedes the default role that is defined for the trusted context.	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the authorization ID is added to the trusted context.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
SECURITYLABEL	VARCHAR(24) NOT NULL	RACF security label for AUTHID. The security label supersedes the default security label, if any, that is defined for the context.	G

SYSIBM.SYSCOPY table

The SYSIBM.SYSCOPY table contains information needed for recovery.

Column name	Data type	Description	Use
DBNAME	CHAR(8) NOT NULL	Name of the database.	G
TSNAME	CHAR(8) NOT NULL	Name of the target table space or index space.	G
DSNUM	INTEGER NOT NULL	Data set number within the table space. For partitioned table spaces, this value corresponds to the partition number for a single partition copy, or 0 for a copy of an entire partitioned table space or index space.	G
ICTYPE	CHAR(1) NOT NULL	Type of operation: A ALTER B REBUILD INDEX C CREATE D CHECK DATA LOG(NO) (no log records for the range are available for RECOVER utility) E RECOVER (to current point) F COPY FULL YES I COPY FULL NO M MODIFY RECOVERY utility P RECOVER TOCOPY or RECOVER TORBA (partial recovery point) Q QUIESCE R LOAD REPLACE LOG(YES) S LOAD REPLACE LOG(NO) T TERM UTILITY command V REPAIR VERSIONS utility W REORG LOG(NO) X REORG LOG(YES) Y LOAD LOG(NO) Z LOAD LOG(YES)	G
	CHAR(6) NOT NULL	Not used	N
START_RBA	CHAR(6) NOT NULL WITH DEFAULT FOR BIT DATA	A 48-bit positive integer that contains the LRSN of a point in the DB2 recovery log. (The LRSN is the RBA in a non-data-sharing environment.) <ul style="list-style-type: none"> For ICTYPE I or F, the starting point for all updates since the image copy was taken For ICTYPE P, the point after the log-apply phase of point-in-time recovery For ICTYPE Q, the point after all data sets have been successfully quiesced For ICTYPE R or S, the end of the log before the start of the LOAD utility and before any data is changed For ICTYPE T, the end of the log when the utility is terminated For other values of ICTYPE, the end of the log before the start of the RELOAD phase of the LOAD or REORG utility. 	G

Column name	Data type	Description	Use
FILESEQNO	INTEGER NOT NULL	Tape file sequence number of the copy.	G
DEVTYPE	CHAR(8) NOT NULL	Device type the copy is on.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DSNAME	CHAR(44) NOT NULL	For ICTYPE='P' (RECOVER TOCOPY only), 'T', or 'F', DSNAME contains the data set name. Otherwise, DSNAME contains the name of the database and table space or index space in the form, <i>database-name.space-name</i> , or DSNAME is blank for any row migrated from a DB2 release prior to Version 4.	G
	CHAR(6) NOT NULL	Not used	N
SHRLEVEL	CHAR(1) NOT NULL	SHRLEVEL parameter on COPY (for ICTYPE F or I only): C Change R Reference blank Does not describe an image copy or was migrated from Version 1 Release 1 of DB2.	G
DSVOLSER	VARCHAR(1784) NOT NULL	The volume serial numbers of the data set. A list of 6-byte numbers separated by commas. Blank if the data set is cataloged.	G
TIMESTAMP	TIMESTAMP NOT NULL WITH DEFAULT	The date and time when the row was inserted. This is the date and time recorded in ICDATE and ICTIME. The use of TIMESTAMP is recommended over that of ICDATE and ICTIME, because the latter two columns might not be supported in later DB2 releases. For the COPYTOCOPY utility, this value is the date and time when the row was inserted for the primary local site or primary recovery site copy. For an EXCHANGE DATA statement, this is the time that the statement is run.	G
ICBACKUP	CHAR(2) NOT NULL WITH DEFAULT	Specifies the type of image copy contained in the data set: blank LOCALSITE primary copy (first data set named with COPYDDN) LB LOCALSITE backup copy (second data set named with COPYDDN) RP RECOVERYSITE primary copy (first data set named with RECOVERYDDN) RB RECOVERYSITE backup copy (second data set named with RECOVERYDDN)	G
ICUNIT	CHAR(1) NOT NULL WITH DEFAULT	Indicates the media that the image copy data set is stored on: D DASD T Tape blank Medium is neither tape nor DASD, the image copy is from a DB2 release prior to Version 2 Release 3, or ICTYPE is not 'T' or 'F'.	G

Column name	Data type	Description	Use
STYPE	CHAR(1) NOT NULL WITH DEFAULT	When ICTYPE=A , the values are: A A partition was added to a table. C A column was added to a table and an index in different commit scopes. E The data set numbers of a base table and its associated clone table are exchanged. G An index was regenerated L The logging attribute of the table space was altered to LOGGED. N An index was altered to not padded O The logging attribute of the table space was altered to NOT LOGGED. P An index was altered to padded R A table was altered to rotate partitions. V A column in a table was altered for a numeric data type change and the column is in an index. Z A column that is in the key of an index that was versioned prior to DB2 Version 8 was altered. When ICTYPE=C , the values are: L The logging attribute of the table space was altered to LOGGED. O The logging attribute of the table space was altered to NOT LOGGED. When ICTYPE=F , the values are: C DFSMS concurrent copy ("I" instance of the table space) J DFSMS concurrent copy ("J" instance of the table space) S LOAD REPLACE(NO) V ALTER INDEX NOT PADDED W REORG LOG(NO) X REORG LOG(YES) blank DB2 image copy	G

Column name	Data type	Description	Use
STYPE (continued)		<p>The MERGECOPY utility, when used to merge an embedded copy with subsequent incremental copies, also produces a record that contains ICTYPE=F and the STYPE of the original image copy (R, S, W, or X).</p> <p>When ICTYPE = M and the MODIFY RECOVERY utility was executed to delete SYSCOPY and/or SYSLGRNX records, the value is R.</p> <p>When ICTYPE=P, the values are:</p> <p>C Recover to a point in time without using logonly with consistency.</p> <p>L Recover to a point in time using logonly without consistency.</p> <p>M Recover to a point in time using logonly with consistency.</p> <p>blank Recover to a point in time without using logonly without consistency.</p> <p>When ICTYPE=Q and option WRITE(YES) is in effect when the quiesce point is taken, the value is W.</p> <p>When ICTYPE=R or S, the values are:</p> <p>A Resetting REORG pending status</p> <p>T First materializing the default value for a row change timestamp column</p> <p>When ICTYPE=T, this field indicates which COPY utility was terminated by the TERM UTILITY command or the START DATABASE command with the ACCESS(FORCE) option. The values are:</p> <p>F COPY FULL YES</p> <p>I COPY FULL NO</p> <p>When ICTYPE=W or X, the values are:</p> <p>A Resetting REORG pending status or REBALANCE</p> <p>T First materializing the default value for a row change timestamp column</p> <p>For other values of ICTYPE, the value is blank.</p>	
PIT_RBA	CHAR(6) NOT NULL WITH DEFAULT FOR BIT DATA	<p>When ICTYPE=P, this field contains the LRSN for the point in the DB2 log. (The LRSN is the RBA in a non-data-sharing environment). For other ICTYPEs, this field is X'000000000000'.</p> <p>When ICTYPE=P, this field indicates the stop location of a point-in-time recovery. If a record contains ICTYPE=P and PIT_RBA=X'000000000000', the copy pending status is active and a full image copy is required. If such a record is encountered during fallback processing of RECOVER, the recover job fails, and a point-in-time recovery is required. PIT_RBA can be zero if the point-in-time recovery is completed by the fall-back processing of RECOVER, or if ICTYPE=P from a prior release of DB2.</p> <p>When ICTYPE=F or I and SHRLEVEL=C, this column contains the current RBA or LRSN that corresponds to the point in the DB2 log when the SHRLEVEL CHANGE copy completes.</p>	G

Column name	Data type	Description	Use
GROUP_MEMBER	CHAR(8) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the operation. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment at the time the operation was performed.	G
OTYPE	CHAR(1) NOT NULL WITH DEFAULT 'T'	Type of object that the recovery information is for: I Index space T Table space	G
LOWDSNUM	INTEGER NOT NULL WITH DEFAULT	Partition number of the lowest partition in the range for SYSCOPY records created for REORG and LOAD REPLACE for resetting a REORG pending status. Version number of an index for SYSCOPY records created for a COPY (ICTYPE=F) of an index space (OTYPE=I). (An index is versioned when a VARCHAR column in the index key is lengthened.) When ICTYPE = F or I, DSNUM = 0 and OTYPE is not equal to I, LOWDSNUM = 1.	G
HIGHDSNUM	INTEGER NOT NULL WITH DEFAULT	Partition number of the highest partition in the range. This column is valid only for SYSCOPY records created for REORG and LOAD REPLACE for resetting REORG pending status. When ICTYPE = F or I, DSNUM = 0 and OTYPE is not equal to I, HIGHDSNUM is the number of the highest partition that is copied.	G
COPYPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of pages written to the copy data set. For inline copies, this number might include pages appearing more than once in the copy data set.	G
NPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	The number of pages in the table space or index at the time of COPY. This number might include pre-formatted pages that are not actually copied.	G
CPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of changed pages.	G
JOBNAME	CHAR(8) NOT NULL WITH DEFAULT	Job name of the utility.	G
AUTHID	CHAR(8) NOT NULL WITH DEFAULT	Authorization ID of the utility.	G
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	When ICTYPE= B, F, I, S, W, or X, the version number of the oldest format of data for an object. For other values of ICTYPE, the value is -1.	G
LOGICAL_PART	INTEGER NOT NULL WITH DEFAULT	Logical partition number.	G

Column name	Data type	Description	Use
LOGGED	CHAR(1) NOT NULL WITH DEFAULT	Indicates the logging attribute of the table space at the time the SYSCOPY record is written: <ul style="list-style-type: none"> Y — indicates that the logging attribute of the table space is LOGGED N — indicates that the logging attribute of the table spaces is NOT LOGGED blank — indicates that the row was inserted prior to Version 9. For a non-LOB table spaces or an index space, blank indicates that the logging attribute is LOGGED. 	G
TTYPE	CHAR(8) NOT NULL WITH DEFAULT		G
TTYPE (cont)		<p>When ICTYPE=E, this column indicates if the full recovery reset the object: blank The full recovery reset the object N The full recovery did not reset the object</p> <p>When ICTYPE=I, TTYPE of S indicates that the directory pages for the index image copy are at the front of each partition and are indicated with a 'V' or '8'.</p> <p>When ICTYPE=P, R, S, W, X, this column indicates the row format for the table space or partition. RRF Indicates that the row format is the reordered row format BRF Indicates that the row format is the basic row format</p> <p>When ICTYPE=M and STYPE=R, this column indicates whether the MODIFY RECOVERY utility deleted rows from SYSIBM.SYSLGRNX. blank MODIFY RECOVERY deleted rows from SYSIBM.SYSLGRNX. N MODIFY RECOVERY did not delete rows from SYSIBM.SYSLGRNX.</p> <p>When ICTYPE=T, TTYPE of B indicates that a broken page was detected during copy.</p>	
INSTANCE	SMALLINT NOT NULL WITH DEFAULT 1	<p>When STYPE = E and ICTYPE = A, INSTANCE indicates the data set instance number of a base object after an EXCHANGE statement completes. The value of the INSTANCE column for the last data exchange will match the value of the INSTANCE column for the SYSIBM.SYSTABLESPACE table.</p> <p>For an image copy, INSTANCE indicates the instance number of the current base objects (table and index).</p>	G
RELCREATED	CHAR(1) NOT NULL WITH DEFAULT	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSCTXTTRUSTATTRS table

The SYSIBM.SYSCTXTTRUSTATTRS table contains one row for each list of attributes for a given trusted context.

Column name	Data type	Description	Use
CONTEXTID	INTEGER NOT NULL	The internal trusted context ID.	G
NAME	VARCHAR(128) NOT NULL	Name of the trust attribute. Possible values including the following attributes: <ul style="list-style-type: none">• An IPv4 address is represented as a dotted decimal IP address. An example of an IPv4 address is '9.112.46.111'.• An IPv6 address is represented as a colon hexadecimal address. An example of an IPv6 address is '2001:0DB8:0000:0000:0008:0800:200C:417A', which can also be expressed in a compressed form as '2001:DB8::8:800:200C:417A'.• A domain name which is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined.• A job or started task name for local applications. If the job name ends with *, any job name that matches the characters prior to * in the specified job name are considered for establishing the trusted connection.• A network access security zone name in the RACF SERVAUTH class.	G
VALUE	VARCHAR(254) NOT NULL	The value of the trust attribute.	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the attribute is created.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSDATABASE table

The SYSIBM.SYSDATABASE table contains one row for each database, except for database DSNDB01.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Database name.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the database.	G
STGROUP	VARCHAR(128) NOT NULL	Name of the default storage group of the database; blank for a system database.	G
BPOOL	CHAR(8) NOT NULL	Name of the default buffer pool of the table space; blank for a system table space.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database. If there were 32511 databases or more when this database was created, the DBID is a negative number.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the database.	G
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
	TIMESTAMP NOT NULL WITH DEFAULT	Not used	N
TYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of database: blank Not a work file database or a TEMP database. W A work file database. The database is DSNDB07, or it was created with the WORKFILE clause and used as a work file database by a member of a DB2 data sharing group.	G
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that uses this work file database. This column is blank if the work file database was not created in a DB2 data sharing environment, or if the database is not a work file database as indicated by the TYPE column.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the database. For DSNDB04 and DSNDB06, the value is '1985-04-01.00.00.000000'.	G

Column name	Data type	Description	Use
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER DATABASE statement was applied. If no ALTER DATABASE statement has been applied, ALTEREDTS has the value of CREATEDTS.	G
ENCODING_SCHEME	CHAR(1) NOT NULL WITH DEFAULT 'E'	Default encoding scheme for the database: E EBCDIC A ASCII U UNICODE blank For DSNDB04, a work file database, and a TEMP database.	G
SBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default SBCS CCSID for the database. For a TEMP database, a work file database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
DBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default DBCS CCSID for the database. If mixed data is not used and the CCSID for the database is defined as EBCDIC or ASCII, the default value is 0. For a TEMP database, a work file database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
MIXED_CCSID	INTEGER NOT NULL WITH DEFAULT	Default mixed CCSID for the database. If mixed data is not used and the CCSID for the database is defined as EBCDIC or ASCII, the default value is 0. For a TEMP database, a work file database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
INDEXBP	CHAR(8) NOT NULL WITH DEFAULT 'BP0'	Name of the default buffer pool for indexes.	G
IMPLICIT	CHAR(1) NOT NULL WITH DEFAULT 'N'	Indicates whether the database was implicitly created: Y The database was implicitly created N The database was explicitly created	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSDATATYPES table

The SYSIBM.SYSDATATYPES table contains one row for each distinct type defined to the system.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the distinct type.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the distinct type.	G
NAME	VARCHAR(128) NOT NULL	Name of the distinct type.	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the distinct type.	G
SOURCESCHEMA	VARCHAR(128) NOT NULL	Schema of the source data type.	G
SOURCETYPE	VARCHAR(128) NOT NULL	Name of the source type.	G
METATYPE	CHAR(1) NOT NULL	The class of data type: T Distinct type	G
DATATYPEID	INTEGER NOT NULL	Internal identifier of the distinct type.	S
SOURCETYPEID	INTEGER NOT NULL	Internal ID of the built-in data type on which the distinct type is based.	S
LENGTH	INTEGER NOT NULL	Maximum length or precision of a distinct type that is based on the IBM-defined DECIMAL data type.	G
SCALE	SMALLINT NOT NULL	Scale for a distinct type that is based on the IBM-defined DECIMAL type. For all other distinct types, the value is 0.	G
SUBTYPE	CHAR(1) NOT NULL	Subtype of the distinct type, which is based on the subtype of the source type: B The subtype is FOR BIT DATA. S The subtype is FOR SBCS DATA. M The subtype is FOR MIXED DATA. blank The source type is not a character type.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the distinct type was created.	G
ENCODING_SCHEME	CHAR(1) NOT NULL	Encoding scheme of the distinct type: A ASCII E EBCDIC U UNICODE	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G
OWNERTYPE 	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G
RELCREATED 	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSDBAUTH table

The SYSIBM.SYSDBAUTH table records the privileges that are held by users over databases.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID or role of the user who granted the privileges. Could also be PUBLIC or PUBLIC followed by an asterisk. (PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see <i>DB2 Administration Guide</i> .).	G
GRANTEE	VARCHAR(128) NOT NULL	Application ID of the user who holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G
NAME	VARCHAR(24) NOT NULL	Database name.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: blank Authorization ID L Role	G
	CHAR(1) NOT NULL	Not used	N
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. blank Not applicable C DBCTRL D DBADM L SYSCTRL M DBMAINT S SYSADM	G
CREATETABAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create tables within the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
CREATETSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create table spaces within the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
DBADMAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBADM authority over the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
DBCTRLAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBCTRL authority over the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
DBMAINTAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBMAINT authority over the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
DISPLAYDBAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the DISPLAY command for the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
DROPAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the ALTER DATABASE and DROP DATABASE statement: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
IMAGCOPYAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the COPY, MERGECOPY, MODIFY, and QUIESCE utilities on the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
LOADAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the LOAD utility to load tables in the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
REORGAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the REORG utility to reorganize table spaces and indexes in the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
RECOVERDBAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the RECOVER and REPORT utilities on table spaces in the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
REPAIRAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the DIAGNOSE and REPAIR utilities on table spaces and indexes in the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
STARTDBAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the START command against the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
STATSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the CHECK and RUNSTATS utilities against the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
STOPAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the STOP command against the database: blank Privilege is not held G Privilege held with the GRANT option Y Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSDBRM table

The SYSIBM.SYSDBRM table contains one row for each DBRM of each application plan.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the DBRM.	G
TIMESTAMP	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	Consistency token.	S
PDSNAME	VARCHAR(132) NOT NULL	Name of the partitioned data set of which the DBRM is a member.	G
PLNAME	VARCHAR(24) NOT NULL	Name of the application plan of which this DBRM is a part.	G
PLCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the application plan.	G
	CHAR(8) NOT NULL	Not used	N
	CHAR(6) NOT NULL	Not used	N
QUOTE	CHAR(1) NOT NULL	SQL string delimiter for the SQL statements in the DBRM: N Apostrophe Y Quotation mark	G
COMMA	CHAR(1) NOT NULL	Decimal point representation for SQL statements in the DBRM: N Period Y Comma	G
HOSTLANG	CHAR(1) NOT NULL	The host language used: B Assembler language C OS/VS COBOL D C F Fortran P PL/I 2 VS COBOL II or IBM COBOL Release 1 (formerly called COBOL/370) 3 IBM COBOL (Release 2 or subsequent releases) 4 C++	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
CHARSET	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the system CCSID for SBCS data was 290 (Katakana) when the program was precompiled: A No K Yes	G

Column name	Data type	Description	Use
MIXED	CHAR(1) NOT NULL WITH DEFAULT	Indicates if mixed data was in effect when the application program was precompiled (for more on when mixed data is in effect, see “Character strings” on page 73): N No Y Yes	G
DEC31	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether DEC31 was in effect when the program was precompiled (for more on when DEC31 is in effect, see “Arithmetic with two decimal operands” on page 183): blank No Y Yes	G
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	Version identifier for the DBRM.	G
PRECOMPTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the DBRM was precompiled.	G
PLCREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSDEPENDENCIES table

The SYSIBM.SYSDEPENDENCIES table records the dependencies between objects.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	Name of the object on which another object is dependent. If BTYPE is F, the name is the specific name of the function.	G
BSHEMA	VARCHAR(128) NOT NULL	Schema or qualifier of the object on which another object is dependent.	G
BCOLNAME	VARCHAR(128) NOT NULL WITH DEFAULT	Column name of the object on which another object is dependent.	G
BCOLNO	SMALLINT NOT NULL WITH DEFAULT	Column number of the object on which another object is dependent.	G
BTYPE	CHAR(1) NOT NULL	The type of object that is identified by BNAME, BSHEMA, and BCOLNAME: F Function	G
BOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the object on which another object is dependent.	G
BOWNERTYPE	CHAR(1) NOT NULL	Type of creator of the object on which another object is dependent: L Role blank Authorization ID that is not a role	G
DNAME	VARCHAR(128) NOT NULL	Name of the object that has dependencies on another object.	G
DSCHEMA	VARCHAR(128) NOT NULL	Schema or qualifier of the object that has dependencies on another object.	G
DCOLNAME	VARCHAR(128) NOT NULL	Column name of the object that has dependencies on another object.	G
DCOLNO	SMALLINT NOT NULL WITH DEFAULT	Column number of the object that has dependencies on another object.	G
DTYPE	CHAR(1) NOT NULL	The type of the object that is identified by DSCHEMA, DNAME, and DCOLNAME: I Index	G
DOWNER	VARCHAR(128) NOT NULL	Authorization ID of the owner of the object that has dependencies on another object.	G

Column name	Data type	Description	Use
DOWNERTYPE	CHAR(1) NOT NULL	Type of creator of the object that has dependencies on another object: L Role blank Authorization ID if not a role	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSDUMMY1 table

The SYSIBM.SYSDUMMY1 table contains one row. The table is used for SQL statements in which a table reference is required, but the contents of the table are not important.

Unlike the other catalog tables, which reside in Unicode table spaces, SYSIBM.SYSDUMMY1 resides in table space SYSEBCDC, which is an EBCDIC table space.

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSENVIRONMENT table

The SYSIBM.SYSENVIRONMENT table records the environment variables when an object is created.

Column name	Data type	Description	Use
ENVID	INTEGER NOT NULL	Internal identifier of the environment.	G
CURRENT_SCHEMA	VARCHAR(128) NOT NULL	The current schema.	G
RELCREATED	CHAR(1) NOT NULL	The release when the environment information is created. See Release dependency indicators for values.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL	The schema path.	G
APPLICATION_ENCODING_CCSID	INTEGER NOT NULL	The CCSID of the application environment.	G
ORIGINAL_ENCODING_CCSID	INTEGER NOT NULL	The original CCSID of the statement text string.	G
DECIMAL_POINT	CHAR(1) NOT NULL	The decimal point indicator: C Comma P Period	G
MIN_DIVIDE_SCALE	CHAR(1) NOT NULL	The minimum divide scale: N The usual rules apply for decimal division in SQL Y Retain at least three digits to the right of the decimal point after any decimal division.	G
STRING_DELIMITER	CHAR(1) NOT NULL	The string delimiter that is used in COBOL string constants: A Apostrophe (') Q Quote (")	G
SQL_STRING_DELIMITER	CHAR(1) NOT NULL	The SQL string delimiter that is used in string constants: A Apostrophe (') Q Quote (")	G
MIXED_DATA	CHAR(1) NOT NULL	Uses mixed DBCS data: N No mixed data Y Mixed data	G
DECIMAL_ARITHMETIC	CHAR(1) NOT NULL	The rules that are to be used for CURRENT PRECISION and when both operands in a decimal operation have a precision of 15 or less: 1 DEC15 specifies that the rules do not allow a precision greater than 15 digits 2 DEC31 specifies that the rules allow a precision of up to 31 digits	G

Column name	Data type	Description	Use
DATE_FORMAT	CHAR(1) NOT NULL	The date format: I ISO - yyyy-mm-dd J JIS - yyyy-mm-dd U USA - mm/dd/yyyy E EUR - dd.mm.yyyy L Locally defined by an installation exit routine	G
TIME_FORMAT	CHAR(1) NOT NULL	The time format: I ISO - hh.mm.ss J JIS - hh.mm.ss U USA - hh:mm AM or hh:mm PM E EUR - hh.mm.ss L Locally defined by an installation exit routine	G
FLOAT_FORMAT	CHAR(1) NOT NULL	The floating point format: I IEEE floating point format S System/390 floating point format	G
HOST_LANGUAGE	CHAR(8) NOT NULL	The host language: • ASM • C • CPP • IBMCOB • PLI • FORTRAN	G
CHARSET	CHAR(1) NOT NULL	The character set: A Alphanumeric	G
FOLD	CHAR(1) NOT NULL	FOLD is only applicable when HOST_LANGUAGE is C or CPP. Otherwise FOLD is blank. N Lower case letters in SBCS ordinary identifiers are not folded to uppercase Y Lower case letters in SBCS ordinary identifiers are folded to uppercase blank Not applicable	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The rounding mode that is used when arithmetic and casting operations are performed on DECFLOAT data: C ROUND_CEILING D ROUND_DOWN F ROUND_FLOOR G ROUND_HALF_DOWN E ROUND_HALF_EVEN H ROUND_HALF_UP U ROUND_UP	G

SYSIBM.SYSFIELDS table

The SYSIBM.SYSFIELDS table contains one row for every column that has a field procedure.

Column name	Data type	Description	Use
TBCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of this column in the table.	G
NAME	VARCHAR(128) NOT NULL	Name of the column.	G
FLDTYPE	VARCHAR(24) NOT NULL	Data type of the encoded values in the field (This columns might contain statistical values from a prior release.): INTEGER Large integer SMALLINT Small integer FLOAT Floating-point CHAR Fixed-length character string VARCHAR Varying-length character string DECIMAL Decimal GRAPHIC Fixed-length graphic string VARG Varying-length graphic string	G
LENGTH	SMALLINT NOT NULL	The length attribute of the field; or, for a decimal field, its precision.(This columns might contain statistical values from a prior release.) The number does not include the internal prefixes that can be used to record actual length and null state. INTEGER 4 SMALLINT 2 FLOAT 8 CHAR Length of string VARCHAR Maximum length of string DECIMAL Precision of number GRAPHIC Number of DBCS characters VARG Maximum number of DBCS characters	G
SCALE	SMALLINT NOT NULL	Scale if FLDTYPE is DECIMAL; otherwise, the value is 0.	G

Column name	Data type	Description	Use
FLDPROC	VARCHAR(24) NOT NULL	For a row describing a field procedure, the name of the procedure. (This columns might contain statistical values from a prior release.)	G
WORKAREA	SMALLINT NOT NULL	For a row describing a field procedure, the size, in bytes, of the work area required for the encoding and decoding of the procedure. (This columns might contain statistical values from a prior release.)	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
EXITPARML	SMALLINT NOT NULL	For a row describing a field procedure, the length of the field procedure parameter value block. (This columns might contain statistical values from a prior release.)	G
PARMLIST	VARCHAR(735) NOT NULL	For a row describing a field procedure, the parameter list following FIELDPROC in the statement that created the column, with insignificant blanks removed. (This columns might contain statistical values from a prior release.)	G
EXITPARM	VARCHAR(1530) NOT NULL WITH DEFAULT FOR BIT DATA	For a row describing a field procedure, the parameter value block of the field procedure (the control block passed to the field procedure when it is invoked). (This columns might contain statistical values from a prior release.)	G

SYSIBM.SYSFOREIGNKEYS table

The SYSIBM.SYSFOREIGNKEYS table contains one row for every column of every foreign key.

Column name	Data type	Description	Use
CREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table that contains the column.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table that contains the column.	G
RELNAME	VARCHAR(128) NOT NULL	Constraint name for the constraint for which the column is part of the foreign key.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column.	G
COLNO	SMALLINT NOT NULL	Numeric place of the column in its table.	G
COLSEQ	SMALLINT NOT NULL	Numeric place of the column in the foreign key.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSINDEXES table

The SYSIBM.SYSINDEXES table contains one row for every index.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the index.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the index is defined.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table.	G
UNIQUERULE	CHAR(1) NOT NULL	Whether the index is unique: D No (duplicates are allowed) U Yes P Yes, and it is a primary index (As in prior releases of DB2, a value of P is used for primary keys that are used to enforce a referential constraint.) C Yes, and it is an index used to enforce UNIQUE constraint N Yes, and it is defined with UNIQUE WHERE NOT NULL R Yes, and it is an index used to enforce the uniqueness of a non-primary parent key G Yes, and it is an index used to enforce the uniqueness of values in a column defined as ROWID GENERATED BY DEFAULT X Yes, and it is an index used to enforce the uniqueness of values in a column that is used to identify or find XML values associated with a specific row.	G
COLCOUNT	SMALLINT NOT NULL	The number of columns in the key.	G
CLUSTERING	CHAR(1) NOT NULL	Whether CLUSTER was specified for the index: N No Y Yes	G
CLUSTERED	CHAR(1) NOT NULL	Whether the table is actually clustered by the index: N A significant number of rows are not in clustering order, or statistics have not been gathered. Y Most of the rows are in clustering order. blank Not applicable. This is an updatable column that can also be changed by the RUNSTATS utility.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database.	S

Column name	Data type	Description	Use
OBID	SMALLINT NOT NULL	Internal identifier of the index fan set descriptor.	S
ISOBID	SMALLINT NOT NULL	Internal identifier of the index page set descriptor.	S
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the index.	G
INDEXSPACE	VARCHAR(24) NOT NULL	Name of the index space.	G
	INTEGER NOT NULL	Not used	N
	INTEGER NOT NULL	Not used	N
NLEAF	INTEGER NOT NULL	Number of active leaf pages in the index. The value is -1 if statistics have not been gathered. This is an updatable column.	S
NLEVELS	SMALLINT NOT NULL	Number of levels in the index tree. If the index is partitioned, it is the maximum of the number of levels in the index tree for all the partitions. The value is -1 if statistics have not been gathered. This is an updatable column.	S
BPOOL	CHAR(8) NOT NULL	Name of the buffer pool used for the index.	G
PGSIZE	SMALLINT NOT NULL	Contains the value 4, 8, 16, or 32 which indicates the size, in KB, of the leaf pages in the index. If the index was created prior to Version 9, the value will be 4096 for a 4 KB page size.	G
ERASERULE	CHAR(1) NOT NULL	Whether the data sets are erased when dropped. The value is meaningless if the index is partitioned: N No Y Yes	G
	VARCHAR(24) NOT NULL	Not used	N
CLOSERULE	CHAR(1) NOT NULL	Whether the data sets are candidates for closure when the limit on the number of open data sets is reached: N No Y Yes	G
SPACE	INTEGER NOT NULL	Number of kilobytes of DASD storage allocated to the index, as determined by the last execution of the STOSPACE utility. The value is 0 if the index is not related to a storage group, or if STOSPACE has not been run. If the index space is partitioned, the value is the total kilobytes of DASD storage allocated to all partitions that are defined in a storage group.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
CLUSTERRATIO	SMALLINT NOT NULL WITH DEFAULT	Percentage of rows that are in clustering order. For a partitioning index, it is the weighted average of all index partitions in terms of the number of rows in the partition. The value is 0 if statistics have not been gathered. The value is -2 if the index is for an auxiliary table. This is an updatable column.	S
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the index.	G
	SMALLINT NOT NULL	Internal use only	I
	SMALLINT NOT NULL	Not used	N
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. This is an updatable column.	G
INDEXTYPE	CHAR(1) NOT NULL WITH DEFAULT	The index type: 2 Type 2 index blank Type 1 index D Data-partitioned secondary index P An index that is both partitioned and is a partitioning index (index that is on a table that uses table-controlled partitioning).	G
FIRSTKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values of the first key column. This number is an estimate if updated while collecting statistics on a single partition. The value is -1 if statistics have not been gathered. This is an updatable column.	S
FULLKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values of the key. The value is -1 if statistics have not been gathered. This is an updatable column.	S
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the index. If the index was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER INDEX statement was executed for the index. If no ALTER INDEX statement has been applied, ALTEREDTS has the value of CREATEDTS. If the index was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
PIECESIZE	INTEGER NOT NULL WITH DEFAULT	Maximum size of a data set in kilobytes for secondary indexes. A value of zero (0) indicates that the index is a partitioning index or that the index was created in a DB2 release prior to Version 5.	G

Column name	Data type	Description	Use
COPY	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether COPY YES was specified for the index, which indicates if the index can be copied and if SYSIBM.SYSLGRNX recording is enabled for the index. N No Y Yes	G
COPYLRN	CHAR(6) NOT NULL WITH DEFAULT X'000000000000' FOR BIT DATA	The value can be either an RBA or LRSN. (LRSN is only for data sharing.) If the index is currently defined as COPY YES, the value is the RBA or LRSN when the index was created with COPY YES or altered to COPY YES, not the current RBA or LRSN. If the index is currently defined as COPY NO, the value is set to X'000000000000' if the index was created with COPY NO; otherwise, if the index was altered to COPY NO, the value in COPYLRN is not changed when the index is altered to COPY NO.	G
CLUSTERRATIOF	FLOAT NOT NULL WITH DEFAULT	When multiplied by 100, the value of the column is the percentage of rows that are in clustering order. For example, a value of '.9125' indicates 91.25%. For a partitioning index, it is the weighted average of all index partitions in terms of the number of rows in the partition. The value is 0 if statistics have not been gathered. The value is -2 if the index is for an auxiliary table, a node ID index or an XML index. This is an updatable column.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. This is an updatable column.	G
REMARKS	VARCHAR(762) NOT NULL WITH DEFAULT	A character field string provided by the user with the COMMENT statement.	G
PADDED	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether keys within the index are padded for varying-length column data: Y The index contains varying-length character or graphic data and is PADDED (the varying-length columns are padded to their maximum length). N The index contains varying-length character or graphic data and is NOT PADDED (the varying-length columns are not padded to their maximum length). Index-only access to all column data is possible. blank The index does not contain varying-length character or graphic data. The value is blank for indexes that have been created or altered prior to Version 8.	G
VERSION	SMALLINT NOT NULL WITH DEFAULT	The version of the data row format for this index. A value of zero indicates that a version-creating alter has never occurred against this index.	G
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the oldest format of data in the index space and any image copies of the index.	G

Column name	Data type	Description	Use
CURRENT_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the newest format of data in the index space. A zero indicates that the index space has never had been versioned. After the version number reaches the maximum value, the number will wrap back to one.	G
RELCREATED	CHAR(1) NOT NULL WITH DEFAULT	Release of DB2 that was used to create the object, blank for indexes created before Version 8. For all other values, see Release dependency indicators.	G
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered.	G
		Not used	N
	CHAR(1) NOT NULL		
KEYTARGET_COUNT	SMALLINT NOT NULL WITH DEFAULT	The number of key-targets for an extended index. The value is 0 for a simple index.	G
UNIQUE_COUNT	SMALLINT NOT NULL WITH DEFAULT	The value is 0 for a simple index or if the index has no unique key. Otherwise, the value is the number of key-targets that are required for the unique key of the index.	G
IX_EXTENSION_TYPE	CHAR(1) NOT NULL WITH DEFAULT	Identifies the type of extended index: N Node ID index S Index on a scalar expression T Spatial index V XML index blank Simple index	G
COMPRESS	CHAR(1) NOT NULL WITH DEFAULT ' N'	Indicates whether index compression is active: N Index compression is not active Y Index compression is active	G
OWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the index, empty string for indexes created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G
DATAREPEAT- FACTORF	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This statistic is only collected when the STATCLUS subsystem parameter is set to ENHANCED. This number is -1 if statistics have not been collected. The valid value is -1 or any value that is equal to or greater than 1. This is an updatable column.	G
ENVID	INTEGER NOT NULL WITH DEFAULT	Internal environment identifier.	G

SYSIBM.SYSINDEXES_HIST table

The SYSIBM.SYSINDEXES_HIST table contains rows from SYSINDEXES.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the index.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the index is defined.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table.	G
CLUSTERING	CHAR(1) NOT NULL	Whether CLUSTER was specified when the index was created: N No Y Yes	G
NLEAF	INTEGER NOT NULL WITH DEFAULT -1	Number of active leaf pages in the index. The value is -1 if statistics have not been gathered.	S
NLEVELS	SMALLINT NOT NULL WITH DEFAULT -1	Number of levels in the index tree. If the index is partitioned, it is the maximum of the number of levels in the index tree for all the partitions. The value is -1 if statistics have not been gathered.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
FIRSTKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of distinct values of the first key column. This number is an estimate if updated while collecting statistics on a single partition. The value is -1 if statistics have not been gathered.	S
FULLKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of distinct values of the key. The value is -1 if statistics have not been gathered.	S
CLUSTERRATIOF	FLOAT(8) NOT NULL	Percentage of rows that are in clustering order. For a partitioning index, it is the weighted average of all index partitions in terms of the number of rows in the partition. The value is 0 if statistics have not been gathered. The value is -2 if the index is for an auxiliary table.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of kilobytes of DASD storage allocated to the index space partition. The value is -1 if statistics have not been gathered.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered.	G
DATAREPEAT- FACTORF 	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This statistic is only collected when the STATCLUS subsystem parameter is set to ENHANCED. This number is -1 if statistics have not been collected. The valid value is -1 or any value that is equal to or greater than 1. This is an updatable column.	G

SYSIBM.SYSINDEXPART table

The SYSIBM.SYSINDEXPART table contains one row for each nonpartitioned secondary index (NPSI) and one row for each partition of a partitioning index or a data-partitioned secondary index.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number; Zero if index is not partitioned.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXCREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
PQTY	INTEGER NOT NULL	For user-managed data sets, the value is the primary space allocation in units of 4KB storage blocks or -1. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER INDEX statement. Unlike PQTY, however, PRIQTY asks for space in 1KB units. A value of -1 indicates that either of the following cases is true: <ul style="list-style-type: none"> • PRIQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for PRIQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement. 	G
SQTY	SMALLINT NOT NULL	For user-managed data sets, the value is the secondary space allocation in units of 4KB storage blocks or -1. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER INDEX statement. Unlike SQTY, however, SECQTY asks for space in 1KB units. A value of -1 indicates that either of the following cases is true: <ul style="list-style-type: none"> • SECQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for SECQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement. If the value does not fit into the column, the value of the column is 32767. See the description of column SECQTYI.	G
STORATYPE	CHAR(1) NOT NULL	Type of storage allocation: E Explicit, and STORNAME names an integrated catalog facility catalog I Implicit, and STORNAME names a storage group	G
STORNAME	VARCHAR(128) NOT NULL	Name of storage group or integrated catalog facility catalog used for space allocation.	G

Column name	Data type	Description	Use
VCATNAME	VARCHAR(24) NOT NULL	Name of integrated catalog facility catalog used for space allocation.	G
	INTEGER NOT NULL	Not used	N
	INTEGER NOT NULL	Not used	N
LEAFDIST	INTEGER NOT NULL	100 times the average number of leaf pages between successive active leaf pages of the index. The value is -1 if statistics have not been gathered. The value is -2 if the index is an auxiliary index, a node ID index, or an XML index.	S
	INTEGER NOT NULL	Not used	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
LIMITKEY	VARCHAR(512) NOT NULL WITH DEFAULT FOR BIT DATA	The high value of the limit key of the partition in an internal format. An empty string if the index is not partitioned or for a data-partitioned secondary index (DPSI). If any column of the key has a field procedure, the internal format is the encoded form of the value.	S
FREEPAGE	SMALLINT NOT NULL	Number of pages that are loaded before a page is left as free space.	G
PCTFREE	SMALLINT NOT NULL	Percentage of each leaf or nonleaf page that is left as free space.	G
SPACE	INTEGER NOT NULL WITH DEFAULT	Number of kilobytes of DASD storage allocated to the index space partition, as determined by the last execution of the STOSPACE utility. 0 The STOSPACE or RUNSTATS utility has not been run. -1 The index was defined with the DEFINE NO clause, which defers the physical creation of the data sets until data is first inserted into the index, and data has yet to be inserted into the index. The value is updated by STOSPACE if the index is related to a storage group. The value is updated by RUNSTATS if the utility is executed as RUNSTATS INDEX with UPDATE(ALL) or UPDATE(SPACE).	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.0000000'.	G

Column name	Data type	Description	Use
	CHAR(1) NOT NULL	Not used	N
GBPCACHE	CHAR(1) NOT NULL WITH DEFAULT	Group buffer pool cache option specified for this index or index partition. blank Only changed pages are cached in the group buffer pool. A Changed and unchanged pages are cached in the group buffer pool. N No data is cached in the group buffer pool.	G
FAROFFPOSF	FLOAT NOT NULL WITH DEFAULT -1	Number of referred to rows far from optimal position because of an insert into a full page. The value is -1 if statistics have not been gathered. The value is -2 if the index is an auxiliary index, a node ID index, or an XML index. The column is not applicable for an index on an auxiliary table.	S
NEAROFFPOSF	FLOAT NOT NULL WITH DEFAULT -1	Number of referred to rows near, but not at optimal position, because of an insert into a full page. The value is -2 if the index is an auxiliary index, a node ID index, or an XML index. Not applicable for an index on an auxiliary table.	S
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of RIDs in the index that refer to data rows or LOBs. The value is -1 if statistics have not been gathered.	S
SECQTYI	INTEGER NOT NULL WITH DEFAULT	Secondary space allocation in units of 4KB storage. For user-managed data sets, the value is the secondary space allocation in units of 4KB blocks.	G
IPREFIX	CHAR(1) NOT NULL WITH DEFAULT 'I'	The first character of the instance qualifier for this index's data set name. 'I' or 'J' are the only valid characters for this field. The default is 'I'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER INDEX statement was executed for the index. If no ALTER INDEX statement has been applied, the value is '0001-01-01.00.00.00.000000'.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. This is an updatable column.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Number of data sets. The value is -1 if statistics have not been gathered. This is an updatable column.	G
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is -1 if statistics have not been gathered. This is an updatable column. This value is only for the last DSNUM for the object.	G

Column name	Data type	Description	Use
PSEUDO_DEL_ENTRIES	INTEGER NOT NULL WITH DEFAULT -1	Number of pseudo deleted entries (entries that are logically deleted but still physically present in the index). For a non-unique index, value is the number of RIDs that are pseudo deleted. For a unique index, the value is the number of keys and RIDs that are pseudo deleted. The value is -1 if statistics have not been gathered. This is an updatable column.	G
LEAFNEAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages physically near previous leaf page for successive active leaf pages. The value is -1 if statistics have not been gathered. This is an updatable column.	S
LEAFFAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages located physically far away from previous leaf pages for successive (active leaf) pages accessed in an index scan. The value is -1 if statistics have not been gathered. This is an updatable column.	S
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the oldest format of data in the index part and any image copies of the index part.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT -1	Time when the partition was created.	G
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered.	G

SYSIBM.SYSINDEXPART_HIST table

The SYSIBM.SYSINDEXPART_HIST table contains rows from SYSINDEXPART.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number. Zero if index is not partitioned.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXCREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
PQTY	INTEGER NOT NULL	<p>For user-managed data sets, the value is the primary space allocation in units of 4KB storage blocks or -1.</p> <p>For user-specified values of PRIQTY other than -1, the value is set to the primary space allocation only if RUNSTATS INDEX with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER INDEX statement. Unlike PQTY, however, PRIQTY asks for space in 1KB units.</p> <p>A value of -1 indicates that either of the following cases is true:</p> <ul style="list-style-type: none"> • PRIQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for PRIQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement. <p>If a storage group is not used, the value is 0.</p>	G
SECQTYI	INTEGER NOT NULL	<p>For user-managed data sets, the value is the secondary space allocation in units of 4KB storage blocks or -1.</p> <p>For user-specified values of SECQTY other than -1, the value is set to the secondary space allocation only if RUNSTATS INDEX with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER INDEX statement. Unlike SQTY, however, SECQTY asks for space in 1KB units.</p> <p>A value of -1 indicates that either of the following cases is true:</p> <ul style="list-style-type: none"> • SECQTY was not specified for a CREATE INDEX statement or for any subsequent ALTER INDEX statements. • -1 was the most recently specified value for SECQTY, either on the CREATE INDEX statement or a subsequent ALTER INDEX statement. <p>If a storage group is not used, the value is 0.</p>	G

Column name	Data type	Description	Use
LEAFDIST	INTEGER NOT NULL WITH DEFAULT -1	100 times the average number of leaf pages between successive active leaf pages of the index. The value is -1 if statistics have not been gathered.	S
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of kilobytes of DASD storage allocated to the index space partition. The value is -1 if statistics have not been gathered.	G
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
FAROFFPOSF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of referred to rows far from optimal position because of an insert into a full page. The value is -1 if statistics have not been gathered. The column is not applicable for an index on an auxiliary table.	S
NEAROFFPOSF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of referred to rows near, but not at optimal position, because of an insert into a full page. Not applicable for an index on an auxiliary table. The value is -1 if statistics have not been gathered.	S
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of RIDs in the index that refer to data rows or LOBs. The value is -1 if statistics have not been gathered.	S
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is -1 if statistics have not been gathered. This value is only for the last DSNUM for the object.	G
PSEUDO_DEL_ENTRIES	INTEGER NOT NULL WITH DEFAULT -1	Number of pseudo deleted entries. The value is -1 if statistics have not been gathered.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Data set number within the table space. For partitioned index spaces, this value corresponds to the partition number for a single partition copy, or 0 for a copy of an entire partitioned index space. The value is -1 if statistics have not been gathered.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
LEAFNEAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages physically near previous leaf page for successive active leaf pages. The value is -1 if statistics have not been gathered. This is an updatable column.	S
LEAFFAR	INTEGER NOT NULL WITH DEFAULT -1	Number of leaf pages located physically far away from previous leaf pages for successive (active leaf) pages accessed in an index scan. The value is -1 if statistics have not been gathered. This is an updatable column.	S
AVGKEYLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of keys within the index. The value is -1 if statistics have not been gathered.	G

SYSIBM.SYSINDEXSPACESTATS table

The SYSIBM.SYSINDEXSPACESTATS table contains real time statistics for index spaces.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
UPDATESTATTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp that the row in the SYSINDEXSPACESTATS table is inserted or last updated.	G
NLEVELS	SMALLINT	The number of levels in the index tree. A null value indicates that the number of levels is unknown.	G
NPAGES	INTEGER	The number of distinct pages with active rows in the associated table. This is an updatable column.	G
NLEAF	INTEGER	The number of leaf pages in the index. This is an updatable column.	G
NACTIVE	INTEGER	The number of active pages in the index space or partition. This value is equivalent to the number of pre-formatted pages. A null value indicates that the number of active pages is unknown.	G
SPACE	INTEGER	The amount of space, in KB, that is allocated to the index space or partition. For multi-piece, linear page sets, this value is the amount of space in all data sets. A null value indicates the amount of space is unknown.	G
EXTENTS	SMALLINT	The number of extents in the index space or partition. For multi-piece index spaces, this value is the number of extents for the last data sets. For a data set that is stripped across multiple volumes, the value is the number of logical extents. A null value indicates the number of extents is unknown.	G
LOADRLASTTIME	TIMESTAMP	The timestamp that the LOAD REPLACE utility was last run on the index space or partition. A null value indicates that the LOAD REPLACE utility has never been run on the index space or partition or that the timestamp is unknown.	G
REBUILDLASTTIME	TIMESTAMP	The timestamp that the REBUILD INDEX utility was last run on the index space or partition. A null value indicates that the timestamp that the REBUILD INDEX was last run is unknown.	G
REORGLASTTIME	TIMESTAMP	The timestamp when the REORG INDEX utility was last run on the index space or partition, or if the REORG INDEX utility has not been run, the time when the index space or partition was created. A null value indicates that the timestamp is unknown.	G

Column name	Data type	Description	Use
REORGINSERTS	INTEGER	The number of index entries that have been inserted into the index space or partition since the last time the REORG, REBUILD INDEX, or LOAD REPLACE utilities were run, or since the object was created.	G
		A null value indicates that the number of inserted index entries is unknown.	
REORGDELETES	INTEGER	The number of index entries that have been deleted from the index space or partition since the last time the REORG, REBUILD INDEX, or LOAD REPLACE utilities were run, or since the object was created.	G
		A null value indicates that the number of deleted index entries is unknown.	
REORGAPPEND-INSERT	INTEGER	The number of index entries that have a key value that is greater than the maximum key value in the index or partition that have been inserted into the index space or partition since the last time the REORG, REBUILD INDEX, or LOAD REPLACE utilities were run, or since the object was created.	G
		A null value indicates that the number of inserted index entries is unknown.	
REORGPSEUDO-DELETES	INTEGER	The number of index entries that have been pseudo-deleted since the last REORG, REBUILD INDEX, or LOAD REPLACE on the index space or partition, or since the object was created. A pseudo-delete is a RID entry that has been marked as deleted.	G
		A null value indicates that the number of pseudo-deleted index entries is unknown.	
REORGMASSDELETE	INTEGER	The number of mass deletes from a segmented or LOB table space, or the number of dropped tables from a segmented table space since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created.	G
		A null value indicates that the number of mass deletes is unknown.	

Column name	Data type	Description	Use
REORGLAFLNEAR	INTEGER	<p>The net number of leaf pages located physically near previous pages for successive active leaf pages that occurred since the last REORG, REBUILD INDEX, or LOAD REPLACE, or since the object was created.</p> <p>The distance between leaf pages is optimal if the difference is 1 and considered near if the distance is 2-16.</p> <p>An index page is added during a page split and the distance between the predecessor and successor pages can lower this count if the distance between the two was near. The distance between the predecessor and new page increase the count if they are near. The distance between the new page and successor increment the count if they are near.</p> <p>If a leaf page is deleted the distance between the new predecessor and successor pages can increment this count if the distance between the two is near. The distance between the predecessor and the deleted page decrement the count if it was near. The distance between the successor and the deleted page decrement the count if it was near.</p> <p>A null value means that the value is unknown. A negative value is possible in some cases.</p>	G
REORGLAFLFAR	INTEGER	<p>The net number of leaf pages located physically far away from previous leaf pages for successive active leaf pages that occurred since the last REORG, REBUILD INDEX, or LOAD REPLACE, or since the object was created.</p> <p>The distance between leaf pages is optimal if the difference is 1 and considered far if the distance is greater than 16.</p> <p>An index page is added during a page split and the distance between the predecessor and successor pages can decrement this count if the distance between the two was far. The distance between the predecessor and new page increment the count if they are far. The distance between the new page and successor increment the count if they are far.</p> <p>If a leaf page is deleted the distance between the new predecessor and successor pages can increment this count if the distance between the two is far. The distance between the predecessor and the deleted page decrement the count if it was far. The distance between the successor and the deleted page decrement the count if it was far.</p> <p>A null value means that the value is unknown.</p>	G
REORGNUMLEVELS	INTEGER	<p>The number of levels in the index tree that were added or removed since the last REORG, REBUILD INDEX, or LOAD REPLACE, or the object was created.</p> <p>A null value means that the number of added or deleted levels is unknown.</p>	G

Column name	Data type	Description	Use
STATSLASTTIME	TIMESTAMP	The timestamp of the last time that the RUNSTATS utility is run on the table space or partition. A null value means that RUNSTATS has never been run on the index space or partition, or that the timestamp of the last RUNSTATS is unknown.	G
STATSINSERTS	INTEGER	The number of records or LOBs that have been inserted into the table space or partition since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of inserted records or LOBs is unknown.	G
STATSDELETES	INTEGER	The number of index entries that have been deleted since the last RUNSTATS on the index space or partition, or since the object was created. A null value means that the number of deleted index entries is unknown.	G
STATSMASSDELETES	INTEGER	The number of times that the index or index space partition was mass deleted since the last RUNSTATS, or the object was created. A null value indicates that the number of mass deletes is unknown.	G
COPYLASTTIME	TIMESTAMP	The timestamp of the last full image copy on the index space or partition. A null value means that COPY has never been run on the index space or partition, or that the timestamp of the last full image copy is unknown.	G
COPYUPDATED-PAGES	INTEGER	The number of distinct types that have been updated since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of updated pages is unknown.	G
COPYCHANGES	INTEGER	The number of insert, update, and delete operations since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of insert, update, and delete operations is unknown.	G
COPYUPDATELRSN	CHAR(6) FOR BIT DATA	The LRSN or RBA of the first update that occurs after the last time the COPY utility was run. A null value indicates that the LRSN or RBA is unknown.	G
COPYUPDATETIME	TIMESTAMP	The timestamp of the first update that occurs after the last time that the COPY utility was run. A null value indicates that the timestamp is unknown.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

Column name	Data type	Description	Use
DBID	SMALLINT NOT NULL	The internal identifier of the database.	G
ISOBID	SMALLINT NOT NULL	The internal identifier of the index space page set descriptor.	I
PSID	SMALLINT NOT NULL	The internal identifier of the table space page set descriptor for the table space that is associated with the index.	G
PARTITION	SMALLINT NOT NULL	The data set number within the index space. For partitioned index spaces, this value corresponds to the partition number for a single partition. For non-partitioned table spaces, this value is 0.	G
INSTANCE	SMALLINT NOT NULL WITH DEFAULT 1	Indicates if the object is associated with data set 1 or 2. This is an updatable column.	G
TOTALENTRIES	BIGINT	The number of entries, including duplicate entries, in the index space or partition. A null value indicates that the number of entries is unknown.	G
DBNAME	VARCHAR(24) NOT NULL	The name of the database.	G
NAME	VARCHAR(128) NOT NULL	The name of the index.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the index.	G
INDEXSPACE	VARCHAR(24) NOT NULL	The name of the index space.	G
LASTUSED	DATE	The date when the index is used for SELECT, FETCH, searched UPDATE, searched DELETE, or used to enforce referential integrity constraints. The default value is NULL.	G

SYSIBM.SYSINDEXSTATS table

The SYSIBM.SYSINDEXSTATS table contains one row for each partition of a partitioning index or a data-partitioned secondary index.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
FIRSTKEYCARD	INTEGER NOT NULL	For the index partition, number of distinct values of the first key column.	S
FULLKEYCARD	INTEGER NOT NULL	For the index partition, number of distinct values of the key.	S
NLEAF	INTEGER NOT NULL	Number of active leaf pages in the index partition.	S
NLEVELS	SMALLINT NOT NULL	Number of levels in the index tree.	S
	SMALLINT NOT NULL	Not used	N
	SMALLINT NOT NULL	Not used	N
CLUSTERRATIO	SMALLINT NOT NULL	For the index partition, the percentage of rows that are in clustering order. The value is 0 if statistics have not been gathered.	N
STATTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
PARTITION	SMALLINT NOT NULL	Partition number of the index.	G
OWNER	VARCHAR(128) NOT NULL	The schema of the index.	G
NAME	VARCHAR(128) NOT NULL	Name of the index.	G
KEYCOUNT	INTEGER NOT NULL	Total number of rows in the partition.	S
FIRSTKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the first key column.	S

Column name	Data type	Description	Use
FULLKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the key.	S
KEYCOUNTF	FLOAT WITH DEFAULT -1	Total number of rows in the partition.	S
CLUSTERRATIOF	FLOAT NOT NULL WITH DEFAULT	For the index partition, the value, when multiplied by 100, is the percentage of rows that are in clustering order. For example, a value of '.9125' indicates 91.25%. The value is 0 if statistics have not been gathered.	G
	VARCHAR(1000) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only	I
DATAREPEAT- FACTORF 	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This statistic is only collected when the STATCLUS subsystem parameter is set to ENHANCED. This number is -1 if statistics have not been collected. The valid value is -1 or any value that is equal to or greater than 1. This is an updatable column.	G

SYSIBM.SYSINDEXSTATS_HIST table

The SYSIBM.SYSINDEXSTATS_HIST table contains rows from SYSINDEXSTATS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NLEAF	INTEGER NOT NULL WITH DEFAULT -1	Number of active leaf pages in the index partition. The value is -1 if statistics have not been gathered.	S
NLEVELS	SMALLINT NOT NULL WITH DEFAULT -1	Number of levels in the index tree. The value is -1 if statistics have not been gathered.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
PARTITION	SMALLINT NOT NULL	Partition number of the index.	G
OWNER	VARCHAR(128) NOT NULL	The schema of the index.	G
NAME	VARCHAR(128) NOT NULL	Name of the index.	G
FIRSTKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the first key column. The value is -1 if statistics have not been gathered.	S
FULLKEYCARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	For the index partition, number of distinct values of the key. The value is -1 if statistics have not been gathered.	S
KEYCOUNTF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of rows in the partition. The value is -1 if statistics have not been gathered.	S
CLUSTERRATIOF	FLOAT(8) NOT NULL	For the index partition, the value, when multiplied by 100, is the percentage of rows that are in clustering order. For example, a value of '0.9125' indicates 91.25%. The value is 0 if statistics have not been gathered.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DATAPEAT- FACTORF	FLOAT NOT NULL WITH DEFAULT -1	The anticipated number of data pages that will be touched following an index key order. This statistic is only collected when the STATCLUS subsystem parameter is set to ENHANCED. This number is -1 if statistics have not been collected. The valid value is -1 or any value that is equal to or greater than 1. This is an updatable column.	G

SYSIBM.SYSJARCLASS_SOURCE table

The SYSIBM.SYSJARCLASS_SOURCE table is an auxiliary table for SYSIBM.SYSJARCONTENTS.

Column name	Data type	Description	Use
CLASS_SOURCE	CLOB(10M) NOT NULL	The contents of the class in the JAR file.	G

SYSIBM.SYSJARCONTENTS table

The SYSIBM.SYSJARCONTENTS table contains Java class source for an installed JAR file.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
CLASS	VARCHAR(384) NOT NULL	The class name contained in the JAR file.	G
CLASS_SOURCE_ROWID	ROWID NOT NULL GENERATED ALWAYS	ID used to support CLOB data type.	G
CLASS_SOURCE	CLOB(10M) NOT NULL	The contents of the class in the JAR file.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSJARDATA table

The SYSIBM.SYSJARDATA table is an auxiliary table for SYSIBM.SYSJAROBJECTS.

Column name	Data type	Description	Use
JAR_DATA	BLOB(100M) NOT NULL	The contents of the JAR file.	G

SYSIBM.SYSJAROBJECTS table

The SYSIBM.SYSJAROBJECTS table contains binary large object representing the installed JAR file.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
OWNER	VARCHAR(128) NOT NULL	Authorization ID of the owner of the JAR object.	G
JAR_DATA_ROWID	ROWID NOT NULL GENERATED ALWAYS	ID used to support BLOB data type.	G
JAR_DATA	BLOB(100M) NOT NULL	The contents of the JAR file. This is an updatable column.	G
PATH	VARCHAR(2048) NOT NULL	The class resolution path of the JAR file. This is an updatable column.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the JAR object was created.	G
ALTEREDTS	TIMESTAMP NOT NULL	Time when the JAR object was altered.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G

SYSIBM.SYSJAVAOPS table

The SYSIBM.SYSJAVAOPS table contains build options used during INSTALL_JAR.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
BUILDSchema	VARCHAR(128) NOT NULL	Schema name for BUILDNAME.	G
BUILDNAME	VARCHAR(128) NOT NULL	Procedure used to create the routine.	G
BUILDOWNER	VARCHAR(128) NOT NULL	Authorization ID used to create the routine.	G
DBRMLIB	VARCHAR(256) NOT NULL	PDS name where DBRM is located.	G
HPJCOMPILE_OPTS	VARCHAR(512) NOT NULL	HPJ compile options used to install the routine.	G
BIND_OPTS	VARCHAR(2048) NOT NULL	Bind options used to install the routine.	G
POBJECT_LIB	VARCHAR(256) NOT NULL	PDSE name where program object is located.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSJAVAPATHS table

The SYSIBM.SYSJAVAPATHS table contains the complete class resolution path of a JAR file, and records the dependencies that one JAR file has on the JAR files in its Java path.

Column name	Data type	Description	Use
JARSCHEMA	VARCHAR(128) NOT NULL	The schema of the JAR file.	G
JAR_ID	VARCHAR(128) NOT NULL	The name of the JAR file.	G
OWNER	VARCHAR(128) NOT NULL	Authorization ID of the owner of the JAR object.	G
ORDINAL	SMALLINT NOT NULL	The ordinal number of the path element within the JAR file's Java path.	G
PE_CLASS_PATTERN	VARCHAR(2048) NOT NULL	The pattern for the names of the classes that are to be searched for in this path element's JAR file.	G
PE_JARSCHEMA	VARCHAR(128) NOT NULL	The schema of this path element's JAR file.	G
PE_JAR_ID	VARCHAR(128) NOT NULL	The name of this path element's JAR file.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSKEYCOLUSE table

The SYSIBM.SYSKEYCOLUSE table contains a row for every column in a unique constraint (primary key or unique key) from the SYSIBM.SYSTABCONST table.

Column name	Data type	Description	Use
CONSTNAME	VARCHAR(128) NOT NULL	Name of the constraint.	G
TBCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the table on which the constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the constraint is defined.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column	G
COLSEQ	SMALLINT NOT NULL	Numeric position of the column in the key (the first position in the key is 1).	G
COLNO	SMALLINT NOT NULL	Numeric position of the column in the table on which the constraint is defined.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSKEYS table

The SYSIBM.SYSKEYS table contains one row for each column of an index key.

Column name	Data type	Description	Use
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXCREATOR	VARCHAR(128) NOT NULL	Schema or qualifier of the index.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the column of the key.	G
COLNO	SMALLINT NOT NULL	Numeric position of the column in the table. For example, 4 (out of 10).	G
COLSEQ	SMALLINT NOT NULL	Numeric position of the column in the key for an index on columns. For example, 4 (out of 4). The value is meaningless for an index that is based on expressions.	G
ORDERING	CHAR(1) NOT NULL	Order of the column in the key: blank Index is an index based on expressions A Ascending order D Descending order R Random order	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSKEYTARGETS table

The SYSIBM.SYSKEYTARGETS table contains one row for each key-target that is participating in an extended index definition.

Column name	Data type	Description	Use
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXSCHEMA	VARCHAR(128) NOT NULL	Qualifier of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
COLNO	SMALLINT NOT NULL	Numeric position of the column in the table if the expression is a single column. Otherwise the value is 0. For XML indexes, this field is also 0.	G
ORDERING	CHAR(1) NOT NULL	Order of the key: A Ascending	G
TYPESCHEMA	VARCHAR(128) NOT NULL	Schema of the data type.	G
TYPENAME	VARCHAR(128) NOT NULL	Name of the data type.	G
DATATYPEID	INTEGER NOT NULL	The internal ID of the data type.	G
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, this column contains 0. For a distinct type, this column contains the internal ID of the built-in type on which the distinct type is based.	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	<p>The length attribute of the key-target or its precision for a decimal key-target. The number does not include the internal prefixes that are used to record the actual length and null states, when applicable.</p> <p>data type value of the LENGTH column</p> <p>INTEGER 4</p> <p>SMALLINT 2</p> <p>FLOAT 4 or 8</p> <p>CHAR The length of the string</p> <p>VARCHAR The maximum length of the string</p> <p>DECIMAL The precision of the number</p> <p>GRAPHIC The number of DBCS characters</p> <p>VARGRAPHIC The maximum number of DBCS characters</p> <p>DATE 4</p> <p>TIME 3</p> <p>TIMESTAMP 10</p> <p>BIGINT 8</p> <p>BINARY The length of the string</p> <p>VARBINARY The maximum length of the string</p> <p>DECFLOAT 8 or 16</p>	G
LENGTH2	INTEGER NOT NULL	<p>The maximum length of the data that is retrieved from the column. Possible values include the following values:</p> <p>0 Not a ROWID column</p> <p>40 For a ROWID column, the length of the value that is returned</p>	G
SCALE	SMALLINT NOT NULL	The scale of decimal data. SCALE contains 0 if the key is not a decimal key.	G
NULLS	CHAR(1) NOT NULL	<p>Whether the key can contain null values:</p> <p>N No</p> <p>Y Yes. Y also indicates that the index is an XML index.</p>	G
CCSID	INTEGER NOT NULL	The CCSID of the key. CCSID contains 0 if the key is a non-character type key.	G
SUBTYPE	CHAR(1) NOT NULL	<p>SUBTYPE applies to character keys only and indicated the subtype of the data:</p> <p>B BIT data</p> <p>M MIXED data</p> <p>S SBCS data</p> <p>blank non-character data</p>	G

Column name	Data type	Description	Use
	VARCHAR(512) NOT NULL FOR BIT DATA	Internal use.	I
CREATEDTS	TIMESTAMP NOT NULL	The timestamp for when the key-target is created.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 in which the key-target is created. SeeRelease dependency indicators for values.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DERIVED_FROM	VARCHAR(4000) NOT NULL	For an index on a scalar expression, DERIVED_FROM contains the text of the scalar expression that is used to generated the key-target value. For an XML index, this is the XML pattern that is used to generate the key-target value. Otherwise DERIVED_FROM contains an empty string.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'. STATSTIME is an updatable column.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	The number of distinct values for the key-target. The value is -2 if the index is a node ID index. For an XML value index, the statistic is collected for the second key target (the DOCID column). For all other key targets of the XML value index, a value of -2 is set.	G
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second highest key-value. HIGH2KEY is an updatable column.	G
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second lowest key-value. LOW2KEY is an updatable column.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics that are gathered: N VARCHAR column statistical values are not padded blank Statistics have not been collects or VARCHAR column statistical values are padded STATS_FORMAT is an updatable column.	G

SYSIBM.SYSKEYTARGETSTATS table

The SYSIBM.SYSKEYTARGETSTATS table contains partition statistics for selected key-targets. For each key-target, a row exists for each partition in the table.

Rows are inserted when RUNSTATS collects indexed key statistics or non-indexed key statistics for a partitioned table space. No row is inserted if the table space is nonpartitioned. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
HIGHKEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The highest key value.	S
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second highest key-value.	S
LOWKEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The lowest key value.	S
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second lowest key-value.	S
PARTITION	SMALLINT NOT NULL	The partition number of the table space.	G
	VARCHAR(1000) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only.	I
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

Column name	Data type	Description	Use
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	The type of statistics that are gathered: N VARCHAR column statistical values are not padded blank Statistics have not been collected or VARCHAR column statistical values are padded	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values for the key target.	G

SYSIBM.SYSKEYTARGETS_HIST table

The SYSIBM.SYSKEYTARGETS_HIST table contains rows from the SYSKEYTARGETS table.

Whenever rows are added or changed in SYSKEYTARGETS, the rows are also written to this table. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
IXNAME	VARCHAR(128) NOT NULL	Name of the index.	G
IXSCHEMA	VARCHAR(128) NOT NULL	Qualifier of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
TYPESCHEMA	VARCHAR(128) NOT NULL	Schema of the data type.	G
TYPENAME	VARCHAR(128) NOT NULL	Name of the data type.	G
DATATYPEID	INTEGER NOT NULL	The internal ID of the data type.	G
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, this field contains 0. For a distinct type, this field contains the internal ID of the built-in type on which the distinct type is based.	G

Column name	Data type	Description	Use
LENGTH	SMALLINT NOT NULL	<p>The length attribute of the key-target or its precision for a decimal key-target. The number does not include the internal prefixes that are used to record the actual length and null states, when applicable.</p> <p>data type value of the LENGTH column INTEGER 4 SMALLINT 2 FLOAT 4 or 8 CHAR The length of the string VARCHAR The maximum length of the string DECIMAL The precision of the number GRAPHIC The number of DBCS characters VARGRAPHIC The maximum number of DBCS characters DATE 4 TIME 3 TIMESTAMP 10 BIGINT 8 BINARY The length of the string VARBINARY The maximum length of the string DECFLOAT 8 or 16</p>	G
LENGTH2	INTEGER NOT NULL	<p>The maximum length of the data that is retrieved from the column. Possible values include the following values:</p> <p>0 Not a ROWID column 40 For a ROWID column, the length of the value that is returned</p>	G
SCALE	SMALLINT NOT NULL	The scale of decimal data. SCALE contains 0 if the key is not a decimal key.	G
NULLS	CHAR(1) NOT NULL	<p>Whether the key can contain null values:</p> <p>N No Y Yes</p>	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'. STATSTIME is an updatable column.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	The number of distinct values for the key-target. The value is -2 if the index is a node ID index. For an XML value index, the statistic is collected for the second key target (the DOCID column). For all other key targets of the XML value index, a value of -2 is set.	G

Column name	Data type	Description	Use
HIGH2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second highest key-value.	G
LOW2KEY	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	The second lowest key-value.	G
STATS_FORMAT	CHAR(1) NOT NULL WITH DEFAULT	<p>The type of statistics that are gathered:</p> <p>N VARCHAR column statistical values are not padded</p> <p>blank Statistics have not been collected or VARCHAR column statistical values are padded</p>	G

SYSIBM.SYSKEYTGTDIST table

The SYSIBM.SYSKEYTGTDIST table contains one or more rows for the first key-target of an extended index key.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If the RUNSTATS utility updated the statistics, this column contains the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	The numeric position of the key-target in the index.	G
KEYVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	KEYVALUE contains the data of a frequently occurring value. If the value has a non-character data type, the data might not be printable.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics that are gathered: C Cardinality F Frequent value N Non-padded frequent value H Histogram statistics	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='C', CARDF contains the number of distinct values for the key group. When TYPE='H', CARDF contains the number of distinct values for the key group in a quantile indicated by QUANTILENO.	G
KEYGROUPKEYNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	KEYGROUPKEYNO contains a value that identifies the set of keys that are associated with the statistics. KEYGROUPKEYNO contains 0 if the statistics are only associated with a single key. If the statistics are associated with more than a single key, KEYGROUPKEYNO contains an array of SMALLINT key numbers with a dimension that is equal to the value in NUMKEYS.	G
NUMKEYS	SMALLINT NOT NULL WITH DEFAULT -1	The number of keys that are associated with the statistics.	G

Column name	Data type	Description	Use
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	<p>When TYPE='F' or 'N', FREQUENCYF contains a value that indicates the percentage of entries in the index that have the value that is contained in the KEYVALUE column.</p> <p>When TYPE='H', FREQUENCYF contains a value that indicates the percentage of entries in the index that have a value that is in the range of the quantile that is indicated in the QUALTILENO column.</p> <p>To determine the percentage from the value of FREQUENCYF, multiply the value by 100. For example, a value of '1' indicates 100 percent. A value of '.153' indicates '15.3' percent.</p>	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	QUANTILENO contains an ordinary sequence number of a quantile in the whole consecutive value range, from low to high.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', LOWVALUE contains the lower bound for the quantile that is in QUANTILENO. LOWVALUE is not used if TYPE does not equal 'H'.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', HIGHVALUE contains the upper bound for the quantile that is in QUANTILENO. HIGHVALUE is not used if TYPE does not equal 'H'.	G

SYSIBM.SYSKEYTGTDISTSTATS table

The SYSIBM.SYSKEYTGTDISTSTATS table contains zero or more rows per partition for the first key-target of a data-partitioned secondary index.

Rows are inserted when RUNSTATS scans a data-partitioned secondary index. No row is inserted if the index is a secondary index. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, STATSTIME contains the timestamp of the most recent RUNSTATS. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
PARTITION	SMALLINT NOT NULL	The partition number of the table space that contains the index in which the key is defined.	G
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	Numeric position of the key-target in the index.	G
KEYVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	KEYVALUE contains the data of a frequently occurring value. If the value has a non-character data type, the data might not be printable.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics that are gathered: C Cardinality F Frequent value N Non-padded frequent value H Histogram statistics	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='C', CARDF contains the number of distinct values for the key group. When TYPE='H', CARDF contains the number of distinct values for the key group in the quantile that is in QUANTILENO.	G
KEYGROUPKEYNO	VARCHAR(254) NOT NULL WITH DEFAULT	Identifies the set of keys that are associated with the statistics. If the statistics are only associated with a single key, KEYGROUPKEYNO contains a zero length value. Otherwise, KEYGROUPKEYNO contains an array of SMALLINT key numbers that have a dimension that is equal to the value in NUMKEYS.	G
NUMKEYS	SMALLINT NOT NULL WITH DEFAULT	Identifies the number of keys that are associated with the statistics.	G

Column name	Data type	Description	Use
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='F' or 'N', FREQUENCYF contains the percentage of entries in the index that have the value that is specified in KEYVALUE when the number of entries is multiplied by 100. For example, a value of '1' indicates 100 percent. A value of '.153' indicates 15.3 percent. When TYPE='H', FREQUENCYF contains the percentage of entries in the index that have a value that is in the range of the quantile that is indicated in QUANTILENO.	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	QUANTILENO contains an ordinary sequence number of a quantile in the whole consecutive value range, from low to high.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', LOWVALUE is the lower bound for the quantile that is indicated in QUANTILENO. LOWVALUE is not used if TYPE does not equal 'H'.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', HIGHVALUE is the upper bound for the quantile that is indicated in QUANTILENO. HIGHVALUE is not used if TYPE does not equal 'H'.	G
	VARCHAR(1000)	Internal use only	I

SYSIBM.SYSKEYTGTDIST_HIST table

The SYSIBM.SYSKEYTGTDIST_HIST table contains rows from the SYSKEYTGTDIST table. Whenever rows are added or changed in SYSKEYTGTDIST, the rows are also written to this table.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If the RUNSTATS utility updated the statistics, this column contains the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
IXSCHEMA	VARCHAR(128) NOT NULL	The qualifier of the index.	G
IXNAME	VARCHAR(128) NOT NULL	The name of the index.	G
KEYSEQ	SMALLINT NOT NULL	The numeric position of the key-target in the index.	G
KEYVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	KEYVALUE contains the data of a frequently occurring value. If the value has a non-character data type, the data might not be printable.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics that are gathered: C Cardinality F Frequent value N Non-padded frequent value H Histogram statistics	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	When TYPE='C', CARDF contains the number of distinct values for the key group. When TYPE='H', CARDF contains the number of distinct values for the key group in a quantile indicated by QUANTILENO.	G
KEYGROUPKEYNO	VARCHAR(254) NOT NULL WITH DEFAULT FOR BIT DATA	KEYGROUPKEYNO contains a value that identifies the set of keys that are associated with the statistics. KEYGROUPKEYNO contains 0 if the statistics are only associated with a single key. If the statistics are associated with more than a single key, KEYGROUPKEYNO contains an array of SMALLINT key numbers with a dimension that is equal to the value in NUMKEYS.	G
NUMKEYS	SMALLINT NOT NULL WITH DEFAULT -1	The number of keys that are associated with the statistics.	G

Column name	Data type	Description	Use
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	<p>When TYPE='F' or 'N', FREQUENCYF contains the percentage of entries in the index that have the value that is specified in KEYVALUE when the number of entries is multiplied by 100. For example, a value of '1' indicates 100 percent. A value of '.153' indicates 15.3 percent.</p> <p>When TYPE='H', FREQUENCYF contains the percentage of entries in the index that have a value that is in the range of the quantile that is indicated in QUANTILENO.</p>	G
QUANTILENO	SMALLINT NOT NULL WITH DEFAULT -1	QUANTILENO contains an ordinary sequence number of a quantile in the whole consecutive value range, from low to high.	G
LOWVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', LOWVALUE contains the lower bound for the quantile that is in QUANTILENO. LOWVALUE is not used if TYPE does not equal 'H'.	G
HIGHVALUE	VARCHAR(2000) NOT NULL WITH DEFAULT FOR BIT DATA	When TYPE='H', HIGHVALUE contains the upper bound for the quantile that is in QUANTILENO. HIGHVALUE is not used if TYPE does not equal 'H'.	G

SYSIBM.SYSLOBSTATS table

The SYSIBM.SYSLOBSTATS table contains one row for each LOB table space.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL	Timestamp of RUNSTATS statistics update.	G
AVGSIZE	INTEGER NOT NULL	Average size of a LOB, measured in bytes, in the LOB table space.	S
FREESPACE	INTEGER NOT NULL	Number of kilobytes of available space in the LOB table space.	S
ORGRATIO	DECIMAL(5,2) NOT NULL	The percentage of organization in the LOB table space. A value of '100' indicates perfect organization of the LOB table space. A value of '1' indicates that the LOB table space is disorganized. A value of '0' indicates that the LOB table space is totally disorganized.	S
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the LOB table space named in NAME.	G
NAME	VARCHAR(24) NOT NULL	Name of the LOB table space.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSLOBSTATS_HIST table

The SYSIBM.SYSLOBSTATS_HIST table contains rows from SYSIBM.SYSLOBSTATS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL	Timestamp of RUNSTATS statistics update.	G
FREESPACE	INTEGER NOT NULL	Number of pages of free space in the LOB table space.	S
ORGRATIO	DECIMAL(5,2) NOT NULL	The percentage of organization in the LOB table space. A value of '100' indicates perfect organization of the LOB table space. A value of '1' indicates that the LOB table space is disorganized. A value of '0' indicates that the LOB table space is totally disorganized.	S
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the LOB table space named in NAME.	G
NAME	VARCHAR(24) NOT NULL	Name of the LOB table space.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSOBJROLEDEP table

The SYSIBM.SYSOBJROLEDEP table lists the dependent objects for each role.

Column name	Data type	Description	Use
DEFINER	VARCHAR(128) NOT NULL	The authorization ID or role that created the object.	G
DEFINERTYPE	CHAR(1) NOT NULL	The type of definer: L Role blank Authorization ID	G
ROLENAME	VARCHAR(128) NOT NULL	Name of the role on which there is a dependency.	G
DSCHEMA	VARCHAR(128) NOT NULL	Name of the schema of the dependent object.	G
DNAME	VARCHAR(762) NOT NULL	Name of the dependent object.	G
DTYPE	CHAR(1) NOT NULL	The type of the dependent object in DNAME: A Alias B Trigger D Database E Distinct type F User-defined function I Index J JAR file K Package L Role M Materialized query table N Trusted context O Stored procedure P Plan Q Sequence R Table space S Storage group T Table V View	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSPACKAGE table

The SYSIBM.SYSPACKAGE table contains a row for every package.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Name of the package collection. For a trigger package, it is the schema name of the trigger.	G
NAME	VARCHAR(128) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	Consistency token for the package. For a package derived from a DB2 DBRM, this is either: <ul style="list-style-type: none"> The "level" as specified by the LEVEL option when the package's program was precompiled The timestamp indicating when the package's program was precompiled, in an internal format. 	S
OWNER	VARCHAR(128) NOT NULL	Authorization ID of the package owner. For a trigger package, the value is the authorization ID of the owner of the trigger, which is set to the current authorization ID (the plan or package owner for static CREATE TRIGGER statement; the CURRENT SQLID for a dynamic CREATE TRIGGER statement).	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the creator of the package version. For a trigger package, the value is determined differently. For dynamic SQL, it is the primary authorization ID of the user who issued the CREATE TRIGGER statement. For static SQL, it is the authorization ID of the plan or package owner.	G
TIMESTAMP	TIMESTAMP NOT NULL	Timestamp indicating when the package was created.	G
BINDTIME	TIMESTAMP NOT NULL	Timestamp indicating when the package was last bound.	G
QUALIFIER	VARCHAR(128) NOT NULL	Implicit qualifier for the unqualified table, view, index, and alias names in the static SQL statements of the package.	G
PKSIZE	INTEGER NOT NULL	Size of the base section ³⁹ of the package, in bytes.	G
AVGSIZE	INTEGER NOT NULL	Average size, in bytes, of those sections ³⁹ of the plan that contain SQL statements processed at bind time.	G
SYSENTRIES	SMALLINT NOT NULL	Number of enabled or disabled entries for this package in SYSIBM.SYSPKSYSTEM. A value of 0 if all types of connections are enabled.	G

39. Packages are divided into *sections*. The base section of the package must be in the EDM pool during the entire time the package is executing. Other sections of the package, corresponding roughly to sets of related SQL statements, are brought into the pool as needed.

Column name	Data type	Description	Use
VALID	CHAR(1) NOT NULL	Whether the package is valid:	G
		A An ALTER statement changed the description of the table or base table of a view referred to by the package. For a CREATE INDEX statement involving data sharing, VALID is also marked as "A". The changes do not invalidate the package.	
		H An ALTER TABLE statement changed the description of the table or base table of a view referred to by the package. For releases of DB2 prior to Version 5, the change invalidates the package.	
		N No Y Yes	
OPERATIVE	CHAR(1) NOT NULL	Whether the package can be allocated:	G
		N No; an explicit BIND or REBIND is required before the package can be allocated.	
		Y Yes	
VALIDATE	CHAR(1) NOT NULL	Whether validity checking can be deferred until run time:	G
		B All checking must be performed at bind time.	
		R Validation is done at run time for tables, views, and privileges that do not exist at bind time.	
ISOLATION	CHAR(1) NOT NULL	Isolation level when the package was last bound or rebound	G
		R RR (repeatable read)	
		S CS (cursor stability)	
		T RS (read stability)	
		U UR (uncommitted read)	
		blank Not specified, and therefore at the level specified for the plan executing the package	
RELEASE	CHAR(1) NOT NULL	The value used for RELEASE when the package was last bound or rebound:	G
		C Value used was COMMIT.	
		D Value used was DEALLOCATE.	
		I The local package is inheriting the value from the plan	
		blank Not specified, and therefore the value specified for the plan executing the package.	
EXPLAIN	CHAR(1) NOT NULL	EXPLAIN option specified for the package; that is, whether information on the package's statements was added to the owner of the PLAN_TABLE table:	G
		N No	
		Y Yes	
QUOTE	CHAR(1) NOT NULL	SQL string delimiter for SQL statements in the package:	G
		N Apostrophe	
		Y Quotation mark	
COMMA	CHAR(1) NOT NULL	Decimal point representation for SQL statements in package:	G
		N Period	
		Y Comma	

Column name	Data type	Description	Use
HOSTLANG	CHAR(1) NOT NULL	Host language for the package's DBRM: B Assembler language C OS/VS COBOL D C F Fortran P PL/I 2 VS COBOL II or IBM COBOL Release 1 (formerly called COBOL/370) 3 IBM COBOL (Release 2 or subsequent releases) 4 C++ blank For remotely bound packages, or trigger packages (TYPE='T')	G
CHARSET	CHAR(1) NOT NULL	Indicates whether the system CCSID for SBCS data was 290 (Katakana) when the program was precompiled: K Yes A No	G
MIXED	CHAR(1) NOT NULL	Indicates if mixed data was in effect when the package's program was precompiled (for more on when mixed data is in effect, see "Character strings" on page 73): N No Y Yes	G
DEC31	CHAR(1) NOT NULL	Indicates whether DEC31 was in effect when the package's program was precompiled (for more on when DEC31 is in effect, see "Arithmetic with two decimal operands" on page 183): N No Y Yes	G
DEFERPREP	CHAR(1) NOT NULL	Indicates the CURRENTDATA option when the package was bound or rebound: A Data currency is required for all cursors. Inhibit blocking for all cursors. B Data currency is not required for ambiguous cursors. C Data currency is required for ambiguous cursors. blank The package was created before the CURRENTDATA option was available.	G
SQLERROR	CHAR(1) NOT NULL	Indicates the SQLERROR option on the most recent subcommand that bound or rebound the package: C CONTINUE N NOPACKAGE	G
REMOTE	CHAR(1) NOT NULL	Source of the package: C Package was created by BIND COPY. D Package was created by BIND COPY with the OPTIONS(COMMAND) option. K The package was copied from a package that was originally bound on behalf of a remote requester. L The package was copied with the OPTIONS(COMMAND) option from a package that was originally bound on behalf of a remote requester. N Package was locally bound from a DBRM. Y Package was bound on behalf of a remote requester.	G

Column name	Data type	Description	Use
PCTIMESTAMP	TIMESTAMP NOT NULL	Date and time the application program was precompiled, or '0001-01-01-00.00.00.000000' if the LEVEL precompiler option was used, or if the package came from a non-DB2 location.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
VERSION	VARCHAR(122) NOT NULL	Version identifier for the package. The value is blank for a trigger package (TYPE='T') .	G
PDSNAME	VARCHAR(132) NOT NULL	For a locally bound package, the name of the PDS (library) in which the package's DBRM is a member. For a locally copied package, the value in SYSPACKAGE.PDSNAME for the source package. Otherwise, the product signature of the bind requester followed by one of the following: <ul style="list-style-type: none"> • The requester's location name if the product is DB2 • Otherwise, the requester's LU name enclosed in angle brackets; for example, "<LUSQLDS>". 	G
DEGREE	CHAR(3) NOT NULL WITH DEFAULT	The DEGREE option used when the package was last bound: ANY DEGREE(ANY) 1 or blank DEGREE(1). Blank if the package was migrated.	G
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the most recent bind. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment when the bind was performed.	G

Column name	Data type	Description	Use
DYNAMICRULES	CHAR(1) NOT NULL WITH DEFAULT	<p>The DYNAMICRULES option used when the package was last bound:</p> <p>B BIND. Dynamic SQL statements are executed with DYNAMICRULES bind behavior.</p> <p>D DEFINEBIND. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES define behavior.</p> <p>When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES bind behavior.</p> <p>E DEFINERUN. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES define behavior.</p> <p>When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES run behavior.</p> <p>H INVOKEBIND. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES invoke behavior.</p> <p>When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES bind behavior.</p> <p>I INVOKERUN. When the package is run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES invoke behavior.</p> <p>When the package is not run under an active stored procedure or user-defined function, dynamic SQL statements in the package are executed with DYNAMICRULES run behavior.</p> <p>R RUN. Dynamic SQL statements are executed with DYNAMICRULES run behavior.</p> <p>blank DYNAMICRULES is not specified for the package. The package uses the DYNAMICRULES value of the plan to which the package is appended at execution time.</p> <p>For a description of the DYNAMICRULES behaviors, see “Authorization IDs and dynamic SQL” on page 64.</p>	G

Column name	Data type	Description	Use
REOPTVAR	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether the access path is determined again at execution time using input variable values: A Bind option REOPT(AUTO) indicates that the access path is determined multiple times at execution time depending on the parameter value. N Bind option REOPT(NONE) indicates that the access path is determined at bind time. Y Bind option REOPT(ALWAYS) indicates that the access path is determined at execution time for SQL statements with variable values. 1 Bind option REOPT(ONCE) indicates that the access path is determined only once at execution time, using the first set of input variable values, regardless of how many times the same statement is executed.	G
DEFERPREPARE	CHAR(1) NOT NULL WITH DEFAULT	Whether PREPARE processing is deferred until OPEN is executed: N Bind option NODEFER(PREPARE) indicates that PREPARE processing is not deferred until OPEN is executed. Y Bind option DEFER(PREPARE) indicates that PREPARE processing is deferred until OPEN is executed. blank Bind option not specified for the package. It is inherited from the plan.	G
KEEPDYNAMIC	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether prepared dynamic statements are to be purged at each commit point: N The bind option is KEEPDYNAMIC(NO). Prepared dynamic SQL statements are destroyed at commit. Y The bind option is KEEPDYNAMIC(YES). Prepared dynamic SQL statements are kept past commit.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL WITH DEFAULT	SQL path specified on the BIND or REBIND command that bound the package. The path is used to resolve unqualified data type, function, and stored procedure names used in certain contexts. If the PATH bind option was not specified, the value in the column is a zero length string; however, DB2 uses a default SQL path of: SYSIBM, SYSFUN, SYSPROC, <i>package qualifier</i> .	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of package. Identifies how the package is created: N CREATE PROCEDURE or ALTER PROCEDURE statement, or BIND PACKAGE DEPLOY command created the package, and this package is a native SQL routine package. T CREATE TRIGGER statement created the package, and the package is a trigger package. blank BIND PACKAGE command created the package.	G
DBPROTOCOL	CHAR(1) NOT NULL WITH DEFAULT 'P'	Whether remote access for SQL with three-part names is implemented with DRDA or DB2 private protocol access: D DRDA P DB2 private protocol	G

Column name	Data type	Description	Use
FUNCTIONTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the function was resolved. Set by the BIND and REBIND commands, but not by AUTOBIND.	G
OPTHINT	VARCHAR(128) NOT NULL WITH DEFAULT	Value of the OPTHINT bind option. Identifies rows in the owner.PLAN_TABLE to be used as input to DB2. Contains blanks if no rows in the owner.PLAN_TABLE are to be used as input.	G
ENCODING_CCSID	INTEGER NOT NULL WITH DEFAULT	The CCSID corresponding to the encoding scheme or CCSID as specified for the bind option ENCODING. The Encoding Scheme specified on the bind command: ccsid The specified or derived CCSID. 0 The default CCSID as specified on panel DSNTIPF at installation time. Used when the package was bound prior to Version 7.	G
IMMEDWRITE	CHAR(1) NOT NULL WITH DEFAULT	Indicates when writes of updated group buffer pool dependent pages are to be done. This option is only applicable for data sharing environments. N Bind option IMMEDWRITE(NO) indicates normal write activity is done. Y Bind option IMMEDWRITE(YES) indicates that immediate writes are done for updated group buffer pool dependent pages. 1 Bind option IMMEDWRITE(PH1) indicates that updated group buffer pool dependent pages are written at or before phase 1 commit. blank A migrated package.	G
RELBOUND	CHAR(1) NOT NULL WITH DEFAULT	The release when the package was bound or rebound. blank Bound prior to Version 7 For all other values, see Release dependency indicators	G
	CHAR(1)	Not used.	N
REMARKS	VARCHAR(550) NOT NULL WITH DEFAULT	A character string provided by the user with the COMMENT statement.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner blank Authorization ID L Role	G
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The ROUNDING option used when the package was last bound: C ROUND_CEILING D ROUND_DOWN F ROUND_FLOOR G ROUND_HALF_DOWN E ROUND_HALF_EVEN H ROUND_HALF_UP U ROUND_UP blank The package created in a DB2 release prior to Version 9.	G

Column name	Data type	Description	Use
DISTRIBUTE	CHAR(1) NOT NULL WITH DEFAULT 'N'	<p>Determines if DB2 should gather location names from SQL statements, and create remote packages for the user (This only has effect during local bind):</p> <p>A DB2 will collect remote location names from SQL statements during local bind, and automatically create remote packages at those sites. The site names are gathered from object names in static SQL statements and literals on CONNECT statements. The sites at which the package is remotely bound can be determined by the location (BTYPPE='X') records in SYSIBM.SYSPACKDEP for this package.</p> <p>L DB2 will automatically create remote packages at the sites specified in the list of location-names. The sites at which the package is remotely bound can be determined by the location (BTYPPE='X') records in SYSIBM.SYSPACKDEP for this package.</p>	G
	DATE NOT NULL WITH DEFAULT	Not used	N

SYSIBM.SYSPACKAUTH table

The SYSIBM.SYSPACKAUTH table records the privileges that are held by users over packages.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privilege. Could also be PUBLIC or PUBLIC followed by an asterisk ⁴⁰ .	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privileges, the name of a plan that uses the privileges or PUBLIC for a grant to PUBLIC.	G
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Collection name for the package or packages on which the privilege was granted.	G
NAME	VARCHAR(128) NOT NULL	Name of the package on which the privileges are held. An asterisk (*) if the privileges are held on all packages in a collection.	G
	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	Not used	N
TIMESTAMP	TIMESTAMP NOT NULL	Timestamp indicating when the privilege was granted.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: blank An authorization ID L Role P An application plan	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. blank Not applicable A PACKADM (on collection *) C DBCTRL D DBADM L SYSCTRL M DBMAINT P PACKADM (on a specific collection) S SYSADM	G
BINDAUTH	CHAR(1) NOT NULL	Whether GRANTEE can use the BIND and REBIND subcommands on the package: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G

40. PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see *DB2 Administration Guide*.

Column name	Data type	Description	Use
COPYAUTH	CHAR(1) NOT NULL	Whether GRANTEE can COPY the package: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
EXECUTEAUTH	CHAR(1) NOT NULL	Whether GRANTEE can execute the package: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSPACKDEP table

The SYSIBM.SYSPACKDEP table records the dependencies of packages on local tables, views, synonyms, table spaces, indexes, aliases, functions, and stored procedures.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	The name of an object that a package depends on.	G
BQUALIFIER	VARCHAR(128) NOT NULL	The value of the column depends on the type of object: <ul style="list-style-type: none"> • If BNAME identifies a table space (BTYPE is R), the value is the name of its database. • If BNAME identifies user-defined function, a cast function, a stored procedure, or a sequence (BTYPE is F, O, or Q), the value is the schema name. • If BNAME identifies a role, the value is blank. • Otherwise, the value is the schema of BNAME. 	G
BTYPE	CHAR(1) NOT NULL	Type of object identified by BNAME and BQUALIFIER: <ul style="list-style-type: none"> A Alias E INSTEAD OF trigger F User-defined function or cast function G Global temporary table I Index M Materialized query table O Stored procedure P Partitioned table space if it is defined as LARGE or with the DSSIZE parm Q Sequence object R Table space S Synonym T Table V View 	G
DLOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
DCOLLID	VARCHAR(128) NOT NULL	Name of the package collection.	G
DNAME	VARCHAR(128) NOT NULL	Name of the package.	G
DCONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	Consistency token for the package. This is either: <ul style="list-style-type: none"> • The "level" as specified by the LEVEL option when the package's program was precompiled • The timestamp indicating when the package's program was precompiled, in an internal format. 	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Owner of the package:	G

Column name	Data type	Description	Use
DTYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of package: N Native SQL routine package O Original copy of a package P Previous copy of a package T Trigger package blank Not a trigger package or a native SQL routine package	G
DOWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner of the package: blank Authorization ID L Role	G

SYSIBM.SYSPACKLIST table

The SYSIBM.SYSPACKLIST table contains one or more rows for every local application plan bound with a package list. Each row represents a unique entry in the plan's package list.

Column name	Data type	Description	Use
PLANNAME	VARCHAR(24) NOT NULL	Name of the application plan.	G
SEQNO	SMALLINT NOT NULL	Sequence number of the entry in the package list.	G
LOCATION	VARCHAR(128) NOT NULL	Location of the package. Blank if this is local. An asterisk (*) indicates location to be determined at run time.	G
COLLID	VARCHAR(128) NOT NULL	Collection name for the package. An asterisk (*) indicates that the collection name is determined at run time.	G
NAME	VARCHAR(128) NOT NULL	Name of the package. An asterisk (*) indicates an entire collection.	G
TIMESTAMP	TIMESTAMP NOT NULL	Timestamp indicating when the row was created.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSPACKSTMT table

The SYSIBM.SYSPACKSTMT table contains one or more rows for each statement in a package.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Name of the package collection.	G
NAME	VARCHAR(128) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	Consistency token for the package. This is either: <ul style="list-style-type: none"> The "level" as specified by the LEVEL option when the package's program was precompiled The timestamp indicating when the package's program was precompiled, in an internal format 	S
SEQNO	INTEGER NOT NULL	Sequence number of the row with respect to a statement in the package ⁴¹ . The numbering starts with 0.	G
STMTNO	SMALLINT NOT NULL	The statement number of the statement in the source program. A statement number greater than 32767 is stored as zero ⁴¹ or as a negative number ⁴² . If the value is zero, see STMTNOI for the statement number.	G
SECTNO	SMALLINT NOT NULL	The section number of the statement. ⁴²	G
BINDERROR	CHAR(1) NOT NULL	Whether an SQL error was detected at bind time: N No Y Yes	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
VERSION	VARCHAR(122) NOT NULL	Version identifier for the package.	G
STMT	VARCHAR(3500) NOT NULL WITH DEFAULT FOR BIT DATA	All or a portion of the text for the SQL statement that the row represents.	S

41. Rows in which the value of SEQNO, STMTNO, and SECTNO are zero are for internal use.

42. To convert a negative STMTNO to a meaningful statement number that corresponds to your precompile output, add 65536 to it. For example, -26472 is equivalent to +39064 (-26472 + 65536).

Column name	Data type	Description	Use
ISOLATION	CHAR(1) NOT NULL WITH DEFAULT	<p>Isolation level for the SQL statement:</p> <p>R RR (repeatable read)</p> <p>T RS (read stability)</p> <p>S CS (cursor stability)</p> <p>U UR (uncommitted read)</p> <p>L RS isolation, with a <i>lock-clause</i></p> <p>X RR isolation, with a <i>lock-clause</i></p> <p>blank The WITH clause was not specified on this statement. The isolation level is recorded in SYSPACKAGE.ISOLATION and in SYSPLAN.ISOLATION.</p>	G
STATUS	CHAR(1) NOT NULL WITH DEFAULT	<p>Status of binding the statement:</p> <p>A Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using defaults for input variables during access path selection.</p> <p>B Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using values for input variables during access path selection.</p> <p>C Compiled - statement was bound successfully using defaults for input variables during access path selection.</p> <p>D Distributed - statement references a remote object using a three-part name. DB2 will implicitly use DRDA access either because the DBPROTOCOL bind option was not specified (defaults to DRDA), or the bind option DBPROTOCOL(DRDA) was explicitly specified. This option allows the use of three-part names with DRDA access but it requires that the package be bound at the target remote site.</p> <p>E Explain - statement is an SQL EXPLAIN statement. The explain is done at bind time using defaults for input variables during access path selection.</p> <p>F Parsed - statement did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using values for input variables during access path selection.</p> <p>G Compiled - statement bound successfully, but REOPT is specified. The statement will be rebound at execution time using values for input variables during access path selection.</p> <p>H Parsed - statement is either a data definition statement or a statement that did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using defaults for input variables during access path selection. Data manipulation statements use defaults for input variables during access path selection.</p> <p>I Indefinite - statement is dynamic. The statement will be bound at execution time using defaults for input variables during access path selection.</p>	S

Column name	Data type	Description	Use
STATUS (cont.)		<p>J Indefinite - statement is dynamic. The statement will be bound at execution time using values for input variables during access path selection.</p> <p>K Control - CALL statement.</p> <p>L Bad - the statement has some allowable error. The bind continues but the statement cannot be executed.</p> <p>M Parsed - statement references a table that is qualified with SESSION and was not bound because the table reference could be for a declared temporary table that will not be defined until the package or plan is run. The SQL statement will be rebound at execution time using values for input variables during access path selection.</p> <p>blank The statement is non-executable, or was bound in a DB2 release prior to Version 5.</p>	
ACCESSPATH	CHAR(1) NOT NULL WITH DEFAULT	<p>For static statements, indicates if the access path for the statement is based on user-specified optimization hints. A value of 'H' indicates that optimization hints were used. A blank value indicates that the access path was determined without the use of optimization hints, or that there is no access path associated with the statement.</p> <p>For dynamic statements, the value is blank.</p>	G
STMTNOI	INTEGER NOT NULL WITH DEFAULT	If the value of STMTNO is zero, the column contains the statement number of the statement in the source program. If both STMTNO and STMTNOI are zero, the statement number is greater than 32767.	G
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement.	G
EXPLAINABLE	CHAR(1) NOT NULL WITH DEFAULT	<p>Contains one of the following values:</p> <p>Y Indicates that the SQL statement can be used with the EXPLAIN function and might have rows describing its access path in the owner.PLAN_TABLE.</p> <p>N Indicates that the SQL statement does not have any rows describing its access path in the owner.PLAN_TABLE.</p> <p>blank Indicates that the SQL statement was bound prior to Version 7.</p>	G
QUERYNO	INTEGER NOT NULL WITH DEFAULT -1	The query number of the SQL statement in the source program. SQL statements bound prior to Version 7 have a default value of -1. Statements bound in Version 7 or later use the value specified on the QUERYNO clause on SELECT, UPDATE, INSERT, DELETE, EXPLAIN, DECLARE CURSOR, or REFRESH TABLE statements. If the QUERYNO clause is not specified, the query number is set to the statement number.	G

SYSIBM.SYSPARMS table

The SYSIBM.SYSPARMS table contains a row for each parameter of a routine or multiple rows for table parameters (one for each column of the table).

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the routine.	G
NAME	VARCHAR(128) NOT NULL	Name of the routine.	G
SPECIFICNAME	VARCHAR(128) NOT NULL	Specific name of the routine.	G
ROUTINETYPE	CHAR(1) NOT NULL	Type of routine: F User-defined function or cast function P Stored procedure	G
CAST_FUNCTION	CHAR(1) NOT NULL	Whether the routine is a cast function: N Not a cast function Y A cast function The only way to get a value of Y is if a user creates a distinct type when DB2 implicitly generates cast functions for the distinct type.	G
PARMNAME	VARCHAR(128) NOT NULL	Name of the parameter. For a table parameter, the parameter name in the row corresponding to the first column of the table is the parameter name specified on CREATE; an empty string or blanks are stored for the parameter name for the rows corresponding to the remaining columns.	G
ROUTINEID	INTEGER NOT NULL	Internal identifier of the routine.	S

Column name	Data type	Description	Use
ROWTYPE	CHAR(1) NOT NULL	<p>The following values indicate the type of parameter described by this row:</p> <p>P Input parameter.</p> <p>O Output parameter; not applicable for functions</p> <p>B Both an input and an output parameter; not applicable for functions</p> <p>R Result before casting; not applicable for stored procedures</p> <p>C Result after casting; not applicable for stored procedures</p> <p>S Input parameter of the underlying built-in source function. For a sourced function and a given ORDINAL value:</p> <ul style="list-style-type: none"> The row with ROWTYPE = P describes the input parameter of the user-defined function (identified by ROUTINEID). The row with ROWTYPE = S describes the corresponding input parameter of the built-in function that is the underlying source function (identified by the SOURCESHEMA and SOURCESPECIFIC values). <p>A value of 'X' indicates that the row is not used to describe a particular parameter of the routine. Instead, for a routine that was created prior to Version 9, the row is used to record a CCSID for the encoding scheme specified in a PARAMETER CCSID clause, or a DATATYPEID for the representation of the variable length character string parameters of a LANGUAGE C routine, as specified in a PARAMETER VARCHAR clause. For routines created with Version 8 (new function mode) or later releases, the CCSID is recorded in the PARAMETER_CCSID column of SYSROUTINES. For routines created with Version 9 or later releases, the DATATYPEID information to support PARAMETER VARCHAR is recorded in the PARAMETER_VARCHARFORM column of SYSIBM.SYSROUTINES.</p>	G
ORDINAL	SMALLINT NOT NULL	<p>If ROWTYPE is B, O, P, or S, the value is the ordinal number of the parameter within the routine signature.</p> <p>If ROWTYPE is C or R, the value depends on the type of function:</p> <ul style="list-style-type: none"> For a scalar function, the value is 0. For a table function, the value is the ordinal number of the column of the output table. <p>If ROWTYPE is X, the value is 0.</p>	G
TYPESHEMA	VARCHAR(128) NOT NULL	Schema of the data type of the parameter.	G
TYPENAME	VARCHAR(128) NOT NULL	Name of the data type of the parameter.	G

Column name	Data type	Description	Use
DATATYPEID	INTEGER NOT NULL	For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type. When ROWTYPE is X and ORDINAL is 0, a non-zero DATATYPEID indicates that actual representation, for a LANGUAGE C routine, of any varying length string parameters that appear in the routine's parameter list of in the RETURNS clause.	S
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, 0. For a distinct type, the internal ID of the built-in data type upon which the distinct type is based.	S
LOCATOR	CHAR(1) NOT NULL	Indicates whether a locator to a value, instead of the actual value, is to be passed or returned when the routine is called: N The actual value is to be passed. Y A locator to a value is to be passed	G
TABLE	CHAR(1) NOT NULL	The data type of a column for a table parameter: N This is not a table parameter. Y This is a table parameter.	G
TABLE_COLNO	SMALLINT NOT NULL	For table parameters, the column number of the table. Otherwise, the value is 0.	G
LENGTH	INTEGER NOT NULL	Length attribute of the parameter or result; If the parameter or result length is determined during function resolution, the length attribute can also be 0. In the case of a decimal parameter or result this is the precision.	G
SCALE	SMALLINT NOT NULL	Scale of the parameter or result. If the parameter or result scale is determined during function resolution, the scale can also be 0.	G
SUBTYPE	CHAR(1) NOT NULL	If the data type is a distinct type, the subtype of the distinct type, which is based on the subtype of its source type: B The subtype is FOR BIT DATA. S The subtype is FOR SBCS DATA. M The subtype is FOR MIXED DATA. blank The source type is not a character type.	G
CCSID	INTEGER NOT NULL	CCSID of the data type for character, graphic, date, time, and timestamp data types. When ROWTYPE is X and ORDINAL is 0, the CCSID column is the CCSID for all character and graphic string parameters.	G
CAST_FUNCTION_ID	INTEGER NOT NULL	Internal function ID of the function used to cast the argument, if this function is sourced on another function, or result. Otherwise, the value is 0. Not applicable for stored procedures.	S
ENCODING_SCHEME	CHAR(1) NOT NULL	Encoding scheme of the parameter: A ASCII E EBCDIC U UNICODE blank The source type is not a character, graphic, or datetime type.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

Column name	Data type	Description	Use
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the routine. The column is a zero-length string if the value of ORIGIN is not 'T' or if the rows were created prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G

SYSIBM.SYSPKSYSTEM table

The SYSIBM.SYSPKSYSTEM table contains zero or more rows for every package. Each row for a given package represents one or more connections to an environment in which the package could be executed.

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL	Always contains blanks	S
COLLID	VARCHAR(128) NOT NULL	Name of the package collection.	G
NAME	VARCHAR(128) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	Consistency token for the package. This is either: <ul style="list-style-type: none"> The "level" as specified by the LEVEL option when the package's program was precompiled The timestamp indicating when the package's program was precompiled, in an internal format. 	S
SYSTEM	VARCHAR(24) NOT NULL	Environment. Values can be: BATCH TSO batch CICS Customer Information Control System DB2CALL DB2 call attachment facility DLIBATCH DLI batch support facility IMSBMP IMS BMP region IMSMPP IMS MPP and IFP region REMOTE remote server	G
ENABLE	CHAR(1) NOT NULL	Indicates whether the connections represented by the row are enabled or disabled: N Disabled Y Enabled	G
CNAME	VARCHAR(60) NOT NULL	Identifies the connection or connections to which the row applies. Interpretation depends on the environment specified by SYSTEM. Values can be: <ul style="list-style-type: none"> Blank if SYSTEM=BATCH or SYSTEM=DB2CALL The LU name for a database server if SYSTEM=REMOTE Either the requester's location (if the product is DB2) or the requester's LU name enclosed in angle brackets if SYSTEM=REMOTE. The name of a single connection if SYSTEM has any other value. CNAME can also be blank when SYSTEM is not equal to BATCH or DB2CALL. When this is so, the row applies to all servers or connections for the indicated environment.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSPLAN table

The SYSIBM.SYSPLAN table contains one row for each application plan.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the application plan.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the application plan.	G
	CHAR(6) NOT NULL	Not used	N
VALIDATE	CHAR(1) NOT NULL	Whether validity checking can be deferred until run time: B All checking must be performed during BIND. R Validation is done at run time for tables, views, and privileges that do not exist at bind time.	G
ISOLATION	CHAR(1) NOT NULL	Isolation level for the plan: R RR (repeatable read) T RS (read stability) S CS (cursor stability) U UR (uncommitted read)	G
VALID	CHAR(1) NOT NULL	Whether the application plan is valid: A An ALTER TABLE statement changed the description of the table or base table of a view that is referred to by the application plan. For a CREATE INDEX statement involving data sharing, VALID is also marked as "A". H An ALTER TABLE statement changed the description of the table or base table of a view that is referred to by the application plan. N No Y Yes	G
OPERATIVE	CHAR(1) NOT NULL	Whether the application plan can be allocated: N No; an explicit BIND or REBIND is required before the plan can be allocated Y Yes	G
	CHAR(8) NOT NULL	Not used	N
PLSIZE	INTEGER NOT NULL	Size of the base section ⁴³ of the plan, in bytes.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
AVGSIZE	INTEGER NOT NULL	Average size, in bytes, of those sections ⁴³ of the plan that contain SQL statements processed at bind time.	G

⁴³ Plans are divided into *sections*. The base section of the plan must be in the EDM pool during the entire time the application program is executing. Other sections of the plan, corresponding roughly to sets of related SQL statements, are brought into the pool as needed.

Column name	Data type	Description	Use
ACQUIRE	CHAR(1) NOT NULL	When resources are acquired: A At allocation U At first use	G
RELEASE	CHAR(1) NOT NULL	When resources are released: C At commit D At deallocation	G
	CHAR(1) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
EXPLAIN	CHAR(1) NOT NULL	EXPLAIN option specified for the plan; that is, whether information on the plan's statements was added to the owner's PLAN_TABLE table: N No Y Yes	G
EXPREDICATE	CHAR(1) NOT NULL	Indicates the CURRENTDATA option when the plan was bound or rebound: B Data currency is not required for ambiguous cursors. Allow blocking for ambiguous cursors. C Data currency is required for ambiguous cursors. Inhibit blocking for ambiguous cursors. N Blocking is inhibited for ambiguous cursors, but the plan was created before the CURRENTDATA option was available.	G
BOUNDDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the binder of the plan.	G
QUALIFIER	VARCHAR(128) NOT NULL WITH DEFAULT	Implicit qualifier for the unqualified table, view, index, and alias names in the static SQL statements of the plan.	G
CACHESIZE	SMALLINT NOT NULL WITH DEFAULT	Size, in bytes, of the cache to be acquired for the plan. A value of zero indicates that no cache is used.	G
PLENTRIES	SMALLINT NOT NULL WITH DEFAULT	Number of package list entries for the plan. The negative of that number if there are rows for the plan in SYSIBM.SYSPACKLIST but the plan was bound in a prior release after fall back.	G
DEFERPREP	CHAR(1) NOT NULL WITH DEFAULT	Whether the package was last bound with the DEFER(PREPARE) option: N No Y Yes	G

Column name	Data type	Description	Use
CURRENTSERVER	VARCHAR(128) NOT NULL WITH DEFAULT	Location name specified with the CURRENTSERVER option when the plan was last bound. Blank if none was specified, implying that the first server is the local DB2 subsystem.	G
SYSENTRIES	SMALLINT NOT NULL WITH DEFAULT	Number of rows associated with the plan in SYSIBM.SYSPLSYSTEM. The negative of that number if such rows exist but the plan was bound in a prior release after fall back. A negative value or zero means that all connections are enabled.	G
DEGREE	CHAR(3) NOT NULL WITH DEFAULT	The DEGREE option used when the plan was last bound: ANY DEGREE(ANY) 1 or blank DEGREE(1). Blank if the plan was migrated.	G
SQLRULES	CHAR(1) NOT NULL WITH DEFAULT	The SQLRULES option used when the plan was last bound: D or blank SQLRULES(DB2) S SQLRULES(STD) blank A migrated plan	G
DISCONNECT	CHAR(1) NOT NULL WITH DEFAULT	The DISCONNECT option used when the plan was last bound: E or blank DISCONNECT(EXPLICIT) A DISCONNECT(AUTOMATIC) C DISCONNECT(CONDITIONAL) blank A migrated plan	G
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the most recent bind. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment when the bind was performed.	G
DYNAMICRULES	CHAR(1) NOT NULL WITH DEFAULT	The DYNAMICRULES option used when the plan was last bound: B BIND. Dynamic SQL statements are executed with DYNAMICRULES bind behavior. blank RUN. Dynamic SQL statements in the plan are executed with DYNAMICRULES run behavior.	G
BOUNDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the plan was bound.	G

Column name	Data type	Description	Use
REOPTVAR	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether the access path is determined again at execution time using input variable values: A Bind option REOPT(AUTO) indicates that the access path is determined multiple times at execution time depending on the parameter value. N Bind option REOPT(NONE) indicates that the access path is determined at bind time. Y Bind option REOPT(ALWAYS) indicates that the access path is determined at execution time for SQL statements with variable values. 1 Bind option REOPT(ONCE) indicates that the access path is determined only once at execution time, using the first set of input variable values, regardless of how many times the same statement is executed.	G
KEEPDYNAMIC	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether prepared dynamic statements are to be purged at each commit point: N The bind option is KEEPDYNAMIC(NO). Prepared dynamic SQL statements are destroyed at commit or rollback. Y The bind option is KEEPDYNAMIC(YES). Prepared dynamic SQL statements are kept past commit or rollback.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL WITH DEFAULT	SQL path specified on the BIND or REBIND command that bound the plan. The path is used to resolve unqualified data type, function, and stored procedure names used in certain contexts. If the PATH bind option was not specified, the value in the column is a zero length string; however, DB2 uses a default SQL path of: SYSIBM, SYSFUN, SYSPROC, <i>plan qualifier</i> .	G
DBPROTOCOL	CHAR(1) NOT NULL WITH DEFAULT 'P'	Whether remote access for SQL with three-part names is implemented with DRDA or DB2 private protocol access: D DRDA P DB2 private protocol	G
FUNCTIONTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the function was resolved. Set by the BIND and REBIND commands, but not by AUTOBIND.	G
OPTHINT	VARCHAR(128) NOT NULL WITH DEFAULT	Value of the OPTHINT bind option. Identifies rows in the owner.PLAN_TABLE to be used as input to DB2. Contains blanks if no rows in the owner.PLAN_TABLE are to be used as input.	G
ENCODING_CCSID	INTEGER NOT NULL WITH DEFAULT	The CCSID corresponding to the encoding scheme or CCSID as specified for the bind option ENCODING. The Encoding Scheme specified on the bind command: ccsid The specified or derived CCSID. 0 The default CCSID as specified on panel DSNTIPF at installation time. Used when the plan was bound prior to Version 7	G

Column name	Data type	Description	Use
IMMEDWRITE	CHAR(1) NOT NULL WITH DEFAULT	Indicates when writes of updated group buffer pool dependent pages are to be done. This option is only applicable for data sharing environments. N Bind option IMMEDIATEWRITE(NO) indicates normal write activity is done. Y Bind option IMMEDIATEWRITE(YES) indicates that immediate writes are done for updated group buffer pool dependent pages. 1 Bind option IMMEDIATEWRITE(PH1) indicates that updated group buffer pool dependent pages are written at or before phase 1 commit. blank A migrated package.	G
RELBOUND	CHAR(1) NOT NULL WITH DEFAULT	The release when the package was bound or rebound. blank Bound prior to Version 7 K Bound on Version 7 L Bound on Version 8	G
	CHAR(1)	Not used.	N
REMARKS	VARCHAR(762) NOT NULL WITH DEFAULT	A character string provided by the user with the COMMENT statement.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G
ROUNDING	CHAR(1) NOT NULL WITH DEFAULT	The ROUNDING option used when the plan was last bound: C ROUND_CEILING D ROUND_DOWN F ROUND_FLOOR G ROUND_HALF_DOWN E ROUND_HALF_EVEN H ROUND_HALF_UP U ROUND_UP blank The plan was created in a DB2 release prior to Version 9.	G
	DATE NOT NULL WITH DEFAULT		N

SYSIBM.SYSPLANAUTH table

The SYSIBM.SYSPLANAUTH table records the privileges that are held by users over application plans.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privileges. Could also be PUBLIC for a grant to PUBLIC.	G
NAME	VARCHAR(24) NOT NULL	Name of the application plan on which the privileges are held.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. blank Not applicable C DBCTRL D DBADM L SYSCTRL M DBMAINT S SYSADM	G
BINDAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the BIND, REBIND, or FREE subcommands against the plan: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
EXECUTEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can run application programs that use the application plan: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G

Column name	Data type	Description	Use
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: blank Authorization ID L Role	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSPLANDEP table

The SYSIBM.SYSPLANDEP table records the dependencies of plans on tables, views, aliases, synonyms, table spaces, indexes, functions, and stored procedures.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	The name of an object the plan depends on.	G
BCREATOR	VARCHAR(128) NOT NULL	If BNAME is a table space, its database. Otherwise, the schema of BNAME. If BNAME is a role, the value is blank.	G
BTYPE	CHAR(1) NOT NULL	Type of object identified by BNAME: A Alias E INSTEAD OF trigger F User-defined function or cast function G Global temporary table I Index M Materialized query table O Stored procedure P Partitioned table space if it is defined as LARGE or with the DSSIZE parm Q Sequence object R Table space S Synonym T Table V View	G
DNAME	VARCHAR(24) NOT NULL	Name of the plan.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSPLSYSTEM table

The SYSIBM.SYSPLSYSTEM table contains zero or more rows for every plan. Each row for a given plan represents one or more connections to an environment in which the plan could be used.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the plan.	G
SYSTEM	VARCHAR(24) NOT NULL	Environment. Values can be: BATCH TSO batch DB2CALL DB2 call attachment facility CICS Customer Information Control System DLIBATCH DLI batch support facility IMSBMP IMS BMP region IMSMPP IMS MPP or IFP region	G
ENABLE	CHAR(1) NOT NULL	Indicates whether the connections represented by the row are enabled or disabled: N Disabled Y Enabled	G
CNAME	VARCHAR(60) NOT NULL	Identifies the connection or connections to which the row applies. Interpretation depends on the environment specified by SYSTEM. Values can be: <ul style="list-style-type: none"> Blank if SYSTEM=BATCH or SYSTEM=DB2CALL The name of a single connection if SYSTEM has any other value <p>CNAME can also be blank when SYSTEM is not equal to BATCH or DB2CALL. When this is so, the row applies to all connections for the indicated environment.</p>	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSRELS table

The SYSIBM.SYSRELS table contains one row for every referential constraint.

Column name	Data type	Description	Use
CREATOR	VARCHAR(128) NOT NULL	The schema of the dependent table of the referential constraint.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the dependent table of the referential constraint.	G
RELNAME	VARCHAR(128) NOT NULL	Constraint name.	G
REFTBNAME	VARCHAR(128) NOT NULL	Name of the parent table of the referential constraint.	G
REFTBCREATOR	VARCHAR(128) NOT NULL	The schema of the parent table.	G
COLCOUNT	SMALLINT NOT NULL	Number of columns in the foreign key.	G
DELETERULE	CHAR(1) NOT NULL	Type of delete rule for the referential constraint: A NO ACTION C CASCADE N SET NULL R RESTRICT	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
RELOBID1	SMALLINT NOT NULL WITH DEFAULT	Internal identifier of the constraint with respect to the database that contains the parent table.	S
RELOBID2	SMALLINT NOT NULL WITH DEFAULT	Internal identifier of the constraint with respect to the database that contains the dependent table.	S
TIMESTAMP	TIMESTAMP NOT NULL WITH DEFAULT	Date and time the constraint was defined. If the constraint is between catalog tables prior to DB2 Version 2 Release 3, the value is '1985-04-01-00.00.00.000000.'	G
IXOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	The schema of unique non-primary index used for the parent key. '99999999' if the enforcing index has been dropped. Blank if the enforcing index is a primary index.	G
IXNAME	VARCHAR(128) NOT NULL WITH DEFAULT	Name of unique non-primary index used for a parent key. '99999999' if the enforcing index has been dropped. Blank if the enforcing index is a primary index.	G

Column name	Data type	Description	Use
ENFORCED	CHAR(1) NOT NULL WITH DEFAULT 'Y'	Enforced by the system or not: Y Enforced by the system N Not enforced by the system (trusted)	G
CHECKEXISTING- DATA	CHAR(1) NOT NULL WITH DEFAULT	Option for checking existing data: I Immediately check existing data. If ENFORCED = 'Y', this column will have a value of 'I'. N Never check existing data. If ENFORCED = 'N', this column will have a value of 'N'.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSRESAUTH table

The SYSIBM.SYSRESAUTH table records CREATE IN and PACKADM ON privileges for collections; USAGE privileges for distinct types; and USE privileges for buffer pools, storage groups, and table spaces.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privilege.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G
QUALIFIER	VARCHAR(128) NOT NULL	Qualifier of the table space (the database name) if the privilege is for a table space (OBTYP= 'R'). The schema name of the distinct type if the privilege is for a distinct type (OBTYP= 'D'). The schema name of the JAR file if the privilege is for a JAR file (OBTYP= 'J'). The value is PACKADM if the privilege is for a collection (OBTYP= 'C') and the authority held is PACKADM. Otherwise, the value is blank.	G
NAME	VARCHAR(128) NOT NULL	Name of the buffer pool, collection, DB2 storage group, distinct type, or table space. Could also be ALL when USE OF ALL BUFFERPOOLS is granted.	G
	CHAR(1) NOT NULL	Internal use only	I
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. blank Not applicable A PACKADM (on collection *) C DBCTRL D DBADM L SYSCTRL M DBMAINT P PACKADM (on a specific collection) S SYSADM	G
OBTYP	CHAR(1) NOT NULL	Type of object: B Buffer pool C Collection D Distinct type R Table space S Storage group J JAR file (Java archive file)	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N

Column name	Data type	Description	Use
USEAUTH	CHAR(1) NOT NULL	Whether the privilege is held with the GRANT option: G Privilege is held with the GRANT option Y Privilege is held without the GRANT option The authority held is PACKADM when the OBTYPE is C (a collection) and QUALIFIER is PACKADM. The authority held is CREATE IN when the OBTYPE is C and QUALIFIER is blank.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
GRANTEETYPE 	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: blank Authorization ID L Role	G
GRANTORTYPE 	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSROLES table

The SYSIBM.SYSROLES table contains one row for each role.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	The name of the role.	G
DEFINER	VARCHAR(128) NOT NULL	The authorization ID or role that defined this role listed in the NAME column.	G
DEFINERTYPE	CHAR(1) NOT NULL	The type of definer: L Role blank Authorization ID	G
CREATEDTS	TIMESTAMP NOT NULL	The time when the role is created.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the role. See Release dependency indicators for the values.	G
REMARKS	VARCHAR(762) NOT NULL	A character string that is provided using the COMMENT statement.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSROUTINEAUTH table

The SYSIBM.SYSROUTINEAUTH table records the privileges that are held by users on routines. (A routine can be a user-defined function, cast function, or stored procedure.)

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privilege.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user who holds the privilege or the name of a plan or package that uses the privilege. Can also be PUBLIC for a grant to PUBLIC.	G
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine	G
SPECIFICNAME	VARCHAR(128) NOT NULL	Specific name of the routine. An asterisk (*) if the privilege is held on all routines in the schema.	G
GRANTEDTS	TIMESTAMP NOT NULL	Time when the GRANT statement was executed.	G
ROUTINETYPE	CHAR(1) NOT NULL	Type of routine: F User-defined function or cast function P Stored procedure	G
GRANTEEType	CHAR(1) NOT NULL	Type of grantee: blank An authorization ID L Role P An application plan or package. The grantee is a package if COLLID is not blank. R Internal use only	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. This field is also used to indicate that the privilege was held on all schemas by the grantor. blank Not applicable 1 Grantor had privilege on schema.* at time of grant L SYSCTRL S SYSADM	G
EXECUTEAUTH	CHAR(1) NOT NULL	Whether GRANTEE can execute the routine: Y Privilege is held without GRANT option. G Privilege is held with GRANT option.	G
COLLID	VARCHAR(128) NOT NULL	If the GRANTEE is a package, its collection name. Otherwise, the value is blank.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	If the GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, the value is blank.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSROUTINES table

The SYSIBM.SYSROUTINES table contains a row for every routine. (A routine can be a user-defined function, cast function, or stored procedure.)

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the routine.	G
NAME	VARCHAR(128) NOT NULL	Name of the routine.	G
ROUTINETYPE	CHAR(1) NOT NULL	Type of routine: F User-defined function or cast function P Stored procedure	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the routine.	G
SPECIFICNAME	VARCHAR(128) NOT NULL	Specific name of the routine.	G
ROUTINEID	INTEGER NOT NULL	Internal identifier of the routine.	S
RETURN_TYPE	INTEGER NOT NULL	Internal identifier of the result data type of the function. The column contains a -2 if the function is a table function.	S
ORIGIN	CHAR(1) NOT NULL	Origin of the routine: E External routine or external SQL procedure N Native SQL procedure Q SQL function S System-generated function U Sourced on user-defined function or built-in function	G
FUNCTION_TYPE	CHAR(1) NOT NULL	Type of function: C Aggregate function S Scalar function T Table function blank For a stored procedure (ROUTINETYPE = 'P')	G
PARAM_COUNT	SMALLINT NOT NULL	Number of parameters for the routine.	G

Column name	Data type	Description	Use
LANGUAGE	VARCHAR(24) NOT NULL	Implementation language of the routine: <ul style="list-style-type: none"> • ASSEMBLE • C • COBOL • COMPJAVA • JAVA • PLI • REXX • SQL <p>The value is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E' or not 'Q'.</p>	G
COLLID	VARCHAR(128) NOT NULL	Name of the package collection to be used when the routine is executed. A blank value indicates the package collection is the same as the package collection of the program that invoked the routine.	G
SOURCESHEMA	VARCHAR(128) NOT NULL	If ORIGIN is 'U' and ROUTINETYPE is 'F', the schema of the source user-defined function ('SYSIBM' for a source built-in function). Otherwise, the value is blank.	G
SOURCESPECIFIC	VARCHAR(128) NOT NULL	If ORIGIN is 'U' and ROUTINETYPE is 'F', the specific name of the source user-defined function or source built-in function name. Otherwise, the value is blank.	G
DETERMINISTIC	CHAR(1) NOT NULL	The deterministic option of an external function or a stored procedure: N Indeterminate (results might differ with a given set of input values). Y Deterministic (results are consistent). blank ROUTINETYPE='F' and ORIGIN is not 'E' or not 'Q' (the routine is a function, but not an external function or an SQL function).	G
EXTERNAL_ACTION	CHAR(1) NOT NULL	The external action option of an external function or SQL function: N Function has no side effects. E Function has external side effects so that the number of invocations is important. blank ORIGIN is not 'E' or 'Q' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
NULL_CALL	CHAR(1) NOT NULL	The CALLED ON NOT NULL INPUT option of an external function or stored procedure: N The routine is not called if any parameter has a NULL value. Y The routine is called if any parameter has a NULL value. blank ROUTINETYPE='F' and ORIGIN is not 'E' (the routine is a function, but not an external function).	G
CAST_FUNCTION	CHAR(1) NOT NULL	Whether the routine is a cast function: N The routine is not a cast function. Y The routine is a cast function. blank ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P'). A cast function is generated by DB2 for a CREATE TYPE statement.	G

Column name	Data type	Description	Use
SCRATCHPAD	CHAR(1) NOT NULL	The SCRATCHPAD option of an external function: N This function does not have a SCRATCHPAD. Y This function has a SCRATCHPAD. blank ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
SCRATCHPAD_LENGTH	INTEGER NOT NULL	Length of the scratchpad if the ORIGIN is 'E' for the function (ROUTINETYPE='F') and NO SCRATCHPAD is not specified. Otherwise, the value is 0.	G
FINAL_CALL	CHAR(1) NOT NULL	The FINAL CALL option of an external function: N A final call will not be made to the function. Y A final call will be made to the function. blank ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
PARALLEL	CHAR(1) NOT NULL	The PARALLEL option of an external function: A This function can be invoked by parallel tasks. D This function cannot be invoked by parallel tasks. blank ORIGIN is not 'E' for the function (ROUTINETYPE='F'), or it is a stored procedure (ROUTINETYPE='P').	G
PARAMETER_STYLE	CHAR(1) NOT NULL	The PARAMETER STYLE option of an external function or stored procedure: D DB2SQL. All parameters are passed to the external function or stored procedure according to the DB2SQL standard convention. G GENERAL. All parameters are passed to the stored procedure according to the GENERAL standard convention. N GENERAL CALL WITH NULLS. All parameters are passed to the stored procedure according to the GENERAL WITH NULLS convention. J JAVA. All parameters are passed to the function or procedure according to the conventions for JAVA and SQLJ specifications. blank The column is blank if the ORIGIN is not 'E' or if LANGUAGE is SQL.	G
FENCED	CHAR(1) NOT NULL	Y Indicates that this routine runs separately from the DB2 address space in a WLM managed DB2 address space. All user-defined routines that are not marked with Y in this column run in the DB2 address space. blank ORIGIN is 'Q' or ORIGIN is 'N'.	G
SQL_DATA_ACCESS	CHAR(1) NOT NULL	The SQL statements that are allowed in an external function, SQL function, or stored procedure: C CONTAINS SQL - Only SQL that does not read or modify data is allowed. M MODIFIES SQL DATA - All SQL is allowed, including SQL that reads or modifies data. N NO SQL - SQL is not allowed. R READS SQL DATA - Only SQL that reads data is allowed. blank Not applicable.	G

Column name	Data type	Description	Use
DBINFO	CHAR(1) NOT NULL	The DBINFO option of an external function or stored procedure: N No, the DBINFO parameter will not be passed to the external function or stored procedure. Y Yes, the DBINFO parameter will be passed to the external function or stored procedure. blank ORIGIN is not 'E'.	G
STAYRESIDENT	CHAR(1) NOT NULL	The STAYRESIDENT option of the routine, which determines whether the routine is to be deleted from memory when the routine ends. N The load module is to be deleted from memory after the routine terminates. Y The load module is to remain resident in memory after the routine terminates. blank ORIGIN is not 'E'.	G
ASUTIME	INTEGER NOT NULL	Number of CPU service units permitted for any single invocation of this routine. If ASUTIME is zero, the number of CPU service units is unlimited. The value is 0 if ROUTINETYPE = 'F' and ORIGIN is not 'E'. If a routine consumes more CPU service units than the ASUTIME value allows, DB2 cancels the routine.	G
WLM_ENVIRONMENT	VARCHAR(96) NOT NULL	Name of the WLM environment to be used to run this routine. When ORIGIN = 'N', this is the name of the WLM ENVIRONMENT FOR DEBUG MODE that is to be used when debugging a native SQL procedure. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'. If the ROUTINETYPE = 'P', the value might be blank. If this value is blank the stored procedure cannot be run.	G
WLM_ENV_FOR_NESTED	CHAR(1) NOT NULL	For nested routine calls, indicates whether the address space of the calling stored procedure or user-defined function is used to run the nested stored procedure or user-defined function: N The nested stored procedure or user-defined function runs in an address space other than the specified WLM environment if the calling stored procedure or user-defined function is not running in the specified WLM environment. 'WLM ENVIRONMENT name' was specified. Y The nested stored procedure or user-defined function runs in the environment used by the calling stored procedure or user-defined function. 'WLM ENVIRONMENT(name,*)' was specified. blank WLM_ENVIRONMENT is blank. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
PROGRAM_TYPE	CHAR(1) NOT NULL	Indicates whether the routine runs as a Language Environment main routine or a subroutine: M The routine runs as a main routine. S The routine runs as a subroutine. blank ORIGIN is not 'E'.	G

Column name	Data type	Description	Use
EXTERNAL_SECURITY	CHAR(1) NOT NULL	Specifies the authorization ID to be used if the routine accesses resources protected by an external security product: D DB2 - The authorization ID associated with the WLM-established stored procedure address space. U SESSION_USER - The authorization ID of the SQL user that invoked the routine. C DEFINER - The authorization ID of the owner of the routine. blank ORIGIN is not 'E'.	G
COMMIT_ON_RETURN	CHAR(1) NOT NULL	If ROUTINETYPE = 'P', whether the transaction is always to be committed immediately on successful return (non-negative SQLCODE) from this stored procedure: N The unit of work is to continue. Y The unit of work is to be committed immediately. If ROUTINETYPE = 'F', the value is blank.	G
RESULT_SETS	SMALLINT NOT NULL	If ROUTINETYPE = 'P', the maximum number of ad hoc result sets that this stored procedure can return. If no ad hoc result sets exist or ROUTINETYPE = 'F', the value is zero.	G
LOBCOLUMNS	SMALLINT NOT NULL	If ORIGIN = 'E' or 'Q', the number of LOB columns found in the parameter list for this user-defined function. If no LOB columns are found in the parameter list or ORIGIN is not 'E' or not 'Q', the value is 0.	I
CREATEDTS	TIMESTAMP NOT NULL	Time when the CREATE statement was executed for this routine.	G
ALTEREDTS	TIMESTAMP NOT NULL	Time when the last ALTER statement was executed for this routine.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
PARM1	SMALLINT NOT NULL	Internal use only	I
PARM2	SMALLINT NOT NULL	Internal use only	I
PARM3	SMALLINT NOT NULL	Internal use only	I
PARM4	SMALLINT NOT NULL	Internal use only	I

Column name	Data type	Description	Use
PARM5	SMALLINT NOT NULL	Internal use only	I
PARM6	SMALLINT NOT NULL	Internal use only	I
PARM7	SMALLINT NOT NULL	Internal use only	I
PARM8	SMALLINT NOT NULL	Internal use only	I
PARM9	SMALLINT NOT NULL	Internal use only	I
PARM10	SMALLINT NOT NULL	Internal use only	I
PARM11	SMALLINT NOT NULL	Internal use only	I
PARM12	SMALLINT NOT NULL	Internal use only	I
PARM13	SMALLINT NOT NULL	Internal use only	I
PARM14	SMALLINT NOT NULL	Internal use only	I
PARM15	SMALLINT NOT NULL	Internal use only	I
PARM16	SMALLINT NOT NULL	Internal use only	I
PARM17	SMALLINT NOT NULL	Internal use only	I
PARM18	SMALLINT NOT NULL	Internal use only	I
PARM19	SMALLINT NOT NULL	Internal use only	I
PARM20	SMALLINT NOT NULL	Internal use only	I

Column name	Data type	Description	Use
PARM21	SMALLINT NOT NULL	Internal use only	I
PARM22	SMALLINT NOT NULL	Internal use only	I
PARM23	SMALLINT NOT NULL	Internal use only	I
PARM24	SMALLINT NOT NULL	Internal use only	I
PARM25	SMALLINT NOT NULL	Internal use only	I
PARM26	SMALLINT NOT NULL	Internal use only	I
PARM27	SMALLINT NOT NULL	Internal use only	I
PARM28	SMALLINT NOT NULL	Internal use only	I
PARM29	SMALLINT NOT NULL	Internal use only	I
PARM30	SMALLINT NOT NULL	Internal use only	I
IOS_PER_INVOC	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of I/Os that required to execute the routine. The value is -1 if the estimated number is not known.	S
INSTS_PER_INVOC	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of machine instructions that required to execute the routine. The value is -1 if the estimated number is not known.	S
INITIAL_IOS	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of I/O's that are performed the first time or the last time the routine is invoked. The value is -1 if the estimated number is not known.	S
INITIAL_INSTS	FLOAT NOT NULL WITH DEFAULT -1	Estimated number of machine instructions that are performed the first time or the last time the routine is invoked. The value is -1 if the estimated number is not known.	S
CARDINALITY	FLOAT NOT NULL WITH DEFAULT -1	The predicted cardinality of the routine, -1 to trigger the use of the default value (10,000).	S

Column name	Data type	Description	Use
RESULT_COLS	SMALLINT NOT NULL DEFAULT 1	For a table function, the number of columns in the result table. Otherwise, the value is 1.	S
EXTERNAL_NAME	VARCHAR(762) NOT NULL	The path/module/function that DB2 should load to execute the routine. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
PARM_SIGNATURE	VARCHAR(150) NOT NULL WITH DEFAULT FOR BIT DATA	Internal use only	I
RUNOPTS	VARCHAR(762) NOT NULL	The Language Environment run-time options to be used for this routine. An empty string indicates that the installation default Language Environment run-time options are to be used. The column is blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G
JAVA_SIGNATURE	VARCHAR(3072) NOT NULL WITH DEFAULT	The signature of the JAR file. blank When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
CLASS	VARCHAR(384) NOT NULL WITH DEFAULT	The class name contained in the JAR file. blank When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
JARSCHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	The schema of the JAR file. blank When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
JAR_ID	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the JAR file. blank When PARAMETER STYLE is not JAVA. The column is also blank if ROUTINETYPE = 'F' and ORIGIN is not 'E'.	G
SPECIAL_REGS	CHAR(1) NOT NULL WITH DEFAULT 'I'	The SPECIAL REGISTER option for a routine. I INHERIT SPECIAL REGISTERS D DEFAULT SPECIAL REGISTERS blank ROUTINETYPE = 'F' and ORIGIN is not 'E' or not 'Q'.	G
NUM_DEP_MQTS	SMALLINT NOT NULL WITH DEFAULT	Number of dependent materialized query tables. The value is 0 if the row does not describe a user-defined table function, or if no materialized query tables are defined on the table function.	G
MAX_FAILURE	SMALLINT NOT NULL WITH DEFAULT -1	Allowable failures for this routine (0-32767). If zero is specified, the routine will never be stopped. If no value is specified for this routine, the default will be -1 to indicate that the DB2 installation parameter (STORMXAB) will be used.	G

Column name	Data type	Description	Use
PARAMETER_CCSID	INTEGER NOT NULL WITH DEFAULT	<p>A CCSID that specifies how character, graphic, date, time, and timestamp data types for system generated parameters to the routine such as message tokens and DBINFO should be passed. The value is dependent on the encoding scheme specified implicitly or explicitly for the PARAMETER CCSID clause defined at the system for that encoding scheme. The following list describes the CCSID for each encoding scheme:</p> <p>ASCII If mixed data is allowed, this CCSID is for mixed ASCII data, SBCS data uses the corresponding SBCS CCSID, and graphic data uses the corresponding DBCS CCSID. Otherwise, this CCSID is for SBCS ASCII data.</p> <p>EBCDIC If mixed data is allowed, this CCSID is for mixed EBCDIC data, SBCS data uses the corresponding SBCS CCSID, and graphic data uses the corresponding DBCS CCSID. Otherwise, this is the CCSID for SBCS EBCDIC data.</p> <p>UNICODE This CCSID is for mixed data (1208).</p> <p>A value of zero means that the CCSIDs used are those CCSIDs for the encoding scheme of other string or datetime parameters in the parameter list or RETURNS clause CCSID clauses, or the value in the DEF ENCODING SCHEME on installation panel DSNTIPF.</p>	G
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	<p>The version identifier for a native SQL procedure (indicated by the value 'N' in the column ORIGIN).</p> <p>A zero length string for the rows that are created prior to Version 9 and for the rows in which the value of ORIGIN is not 'N'.</p>	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	The consistency token for the routine. The column is set to X'20' if the value of ORIGIN is not 'N'	G
ACTIVE	CHAR(1) NOT NULL WITH DEFAULT	<p>Identifies the active version of the routine:</p> <p>Y The routine is the active version.</p> <p>N The routine is not the active version.</p> <p>blank The value of ORIGIN is not 'N' or the row was created prior to Version 9.</p>	G

Column name	Data type	Description	Use
DEBUG_MODE	CHAR(1) NOT NULL WITH DEFAULT	Identifies whether or not this routine is enabled for debugging: 1 This routine is enabled for debugging and can be debugged in a client debug session using the DB2 Unified Debugger. 0 This routine is not enabled for debugging. N This routine can never be enabled for debugging. blank The LANGUAGE is not specified as JAVA, the value of ORIGIN is not 'N', or the row was created prior to Version 9.	G
TEXT_ENVID	INTEGER NOT NULL WITH DEFAULT	Internal identifier of the environment. The column is 0 if the value of ORIGIN is not 'N' or if the row was created prior to Version 9.	G
TEXT_ROWID	ROWID NOT NULL GENERATED ALWAYS	ID to support LOB columns for source text.	G
TEXT	CLOB(2M) NOT NULL WITH DEFAULT	The source text of the CREATE statement or the ALTER statement with the body for the routine. The column is a zero-length string if the value of ORIGIN is not 'N' or if the row was created prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G
PARAMETER_VARCHARFORM	INTEGER NOT NULL WITH DEFAULT	A non-zero value that indicates the actual representation, to a LANGUAGE C routine, of any varying length string parameter that appears in the parameter list or RETURNS clause for that routine.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
PACKAGEPATH	VARCHAR(4096)	The value of the PACKAGE PATH option of the CREATE FUNCTION, CREATE PROCEDURE, ALTER FUNCTION, or ALTER PROCEDURE statement that created or last changed the routine. PACKAGE PATH identifies the package path to use when the routine is executed. A blank value indicates the package path is the same as the package path of the program that invoked the routine.	G

SYSIBM.SYSROUTINESTEXT table

The SYSIBM.SYSROUTINESTEXT is an auxiliary table for the TEXT column of SYSIBM.SYSROUTINES and is required to hold the LOB data.

Column name	Data type	Description	Use
TEXT	CLOB(2M) NOT NULL WITH DEFAULT	The source text of the CREATE PROCEDURE statement for the routine. TEXT can also hold the source text of the ALTER PROCEDURE statement for the routine if the routine is a native SQL procedure and the SQL procedure body is included in the ALTER PROCEDURE statement.	G

SYSIBM.SYSROUTINES_OPTS table

The SYSIBM.SYSROUTINES_OPTS table Contains a row for each generated routine, such as one created by DB2 for z/OS Procedure Processor DSNTPSMP, that records the build options for the routine.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
ROUTINENAME	VARCHAR(128) NOT NULL	Name of the routine.	G
BUILDDATE	DATE NOT NULL WITH DEFAULT	Date the routine was built.	G
BUILDTIME	TIME NOT NULL WITH DEFAULT	Time the routine was built.	G
BUILDSTATUS	CHAR(1) NOT NULL WITH DEFAULT 'C'	Whether this version of the routine's options is the current version.	G
BUILDSHEMA	VARCHAR(128) NOT NULL	Schema name for BUILDNAME.	G
BUILDNAME	VARCHAR(128) NOT NULL	Procedure used to create the routine.	G
BUILDOWNER	VARCHAR(128) NOT NULL	Authorization ID used to create the routine.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
PRECOMPILE_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	Precompiler options used to build the routine.	G
COMPILE_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	Compiler options used to build the routine.	G
PRELINK_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	Prelink-edit options used to build the routine.	G

Column name	Data type	Description	Use
LINK_OPTS	VARCHAR(765) NOT NULL WITH DEFAULT	Link-edit options used to build the routine.	G
BIND_OPTS	VARCHAR(3072) NOT NULL WITH DEFAULT	Bind options used to build the routine.	G
SOURCEDSN	VARCHAR(765) NOT NULL WITH DEFAULT	Name of the source data set.	G
DEBUG_MODE	CHAR(1) NOT NULL	Debugging is on or off for this object. 0 Debugging is off. Default and value on migration are both 0. 1 Debugging is on.	G

SYSIBM.SYSROUTINES_SRC table

The SYSIBM.SYSROUTINES_SRC table contains source for generated routines, such as those created by DB2 for z/OS Procedure Processor DSNTPSMP.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the routine.	G
ROUTINENAME	VARCHAR(128) NOT NULL	Name of the routine.	G
BUILDDATE	DATE NOT NULL WITH DEFAULT	Date the routine was built.	G
BUILDTIME	TIME NOT NULL WITH DEFAULT	Time the routine was built.	G
BUILDSTATUS	CHAR(1) NOT NULL WITH DEFAULT 'C'	Whether this version of the routine's source is the current version.	G
SEQNO	INTEGER NOT NULL	Number of the source statement piece in CREATESTMT.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
CREATESTMT	VARCHAR(7500) NOT NULL	Routine source statement.	G

SYSIBM.SYSSCHEMAAUTH table

The SYSIBM.SYSSCHEMAAUTH table contains one or more rows for each user that is granted a privilege on a particular schema in the database.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges or SYSADM.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user or group who holds the privileges. Can also be PUBLIC for a grant to PUBLIC.	G
SCHEMANAME	VARCHAR(128) NOT NULL	Name of the schema or '*' for all schemas.	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. This field is also used to indicate that the privilege was held on all schemas by the grantor. 1 Grantor had privilege on all schemas at time of grant L SYSCtrl S SYSADM	G
CREATEINAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds CREATEIN privilege on the schema: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
ALTERINAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds ALTERIN privilege on the schema: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
DROPINAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds DROPIN privilege on the schema: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
GRANTEDTS	TIMESTAMP NOT NULL	Time when the GRANT statement was executed.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
GRANTEEType	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: blank Authorization ID L Role	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSSEQUENCEAUTH table

The SYSIBM.SYSSEQUENCEAUTH table records the privileges that are held by users over sequences.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user or group that holds the privileges or the name of an application plan or package that uses the privileges. PUBLIC for a grant to PUBLIC.	G
SCHEMA	VARCHAR(128) NOT NULL	Schema of the sequence.	G
NAME	VARCHAR(128) NOT NULL	Name of the sequence.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: blank An authorization ID. L Role P An application plan or package. The grantee is a package if COLLID is not blank. R Internal use only.	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor: blank Not applicable L SYSCTRL S SYSADM	G
ALTERAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds ALTER privilege on the sequence: blank Privilege is not held. G Privilege is held with the GRANT option. Y Privilege is held without the GRANT option.	G
USEAUTH	CHAR(1) NOT NULL	Indicates whether grantee holds USAGE privilege on the sequence: blank Privilege is not held. G Privilege is held with the GRANT option. Y Privilege is held without the GRANT option.	G
COLLID	VARCHAR(128) NOT NULL	If the GRANTEE is a package, its collection name. Otherwise, a string of length zero.	G
CONTOKEN	CHAR(8) NOT NULL FOR BIT DATA	If the GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, blank.	G
GRANTEDTS	TIMESTAMP NOT NULL	Time when the GRANT statement was executed.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

Column name	Data type	Description	Use
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSSEQUENCES table

The SYSIBM.SYSSEQUENCES table contains one row for each identity column or user-defined sequence.

Column name	Data type	Description	Use
SCHEMA	VARCHAR(128) NOT NULL	Schema of the sequence. For an identity column, the value of TBCREATOR from the SYSCOLUMNS entry for the column.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the sequence. For an identity column, the value of TBCREATOR from the SYSCOLUMNS entry for the column.	G
NAME	VARCHAR(128) NOT NULL	Name of the identity column or sequence. (The name for an identity is generated by DB2.)	G
SEQTYPE	CHAR(1) NOT NULL	Type of sequence object: I An identity column S A user-defined sequence X An implicitly created DOCID column for a base table that contains XML data.	G
SEQUENCEID	INTEGER NOT NULL	Internal identifier of the identity column or sequence.	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the sequence or identity column.	G
INCREMENT	DECIMAL(31,0) NOT NULL	Increment value (positive or negative, within INTEGER scope).	G
START	DECIMAL(31,0) NOT NULL	Start value.	G
MAXVALUE	DECIMAL(31,0) NOT NULL	Maximum value allowed for the identity column or sequence.	G
MINVALUE	DECIMAL(31,0) NOT NULL	Minimum value allowed for the identity column or sequence.	G
CYCLE	CHAR(1) NOT NULL	Whether cycling will occur when a boundary is reached: N No Y Yes, cycling will occur	G
CACHE	INTEGER NOT NULL	Number of sequence values to preallocate in memory for faster access. A value of 0 indicates that values are not to be preallocated.	G
ORDER	CHAR(1) NOT NULL	Whether the values must be generated in order: Y Yes N No	G
DATATYPEID	INTEGER NOT NULL	For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type.	S
SOURCETYPEID	INTEGER NOT NULL	For a built-in data type, 0. For a distinct type, the internal ID of the built-in data type upon which the distinct type is based.	S

Column name	Data type	Description	Use
CREATEDTS	TIMESTAMP NOT NULL	Timestamp when the identity column or sequence was created.	G
ALTEREDTS	TIMESTAMP NOT NULL	Timestamp when the last ALTER statement was executed for this identity column or sequence.	G
MAXASSIGNEDVAL	DECIMAL(31,0)	Last possible assigned value. Initialized to null when the object is created. Updated each time the next chunk of <i>n</i> values is cached, where <i>n</i> is the value for CACHE.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
REMARKS	VARCHAR(762) NOT NULL	Character string provided by user with the COMMENT statement. The value is blank for an identity column.	G
PRECISION	SMALLINT NOT NULL WITH DEFAULT	The precision defined for a sequence with a decimal or numeric type. The value is 5 for SMALLINT, 10 for INTEGER, or the actual precision specified by the user for the decimal data type. The value is 0 for rows created prior to Version 8.	G
RESTARTWITH 	DECIMAL(31,0) NULLABLE WITH DEFAULT	The RESTART WITH value specified for a sequence during ALTER or NULL. The value is NULL if no ALTER with RESTART WITH has happened.	G
OWNERTYPE 	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G
RELCREATED 	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSSEQUENCESDEP table

The SYSIBM.SYSSEQUENCESDEP table records the dependencies of identity columns and sequences.

Column name	Data type	Description	Use
BSEQUENCEID	INTEGER NOT NULL	Internal identifier of the identity column or sequence.	G
DCREATOR	VARCHAR(128) NOT NULL	The owner of the object that is dependent on this identity column or sequence.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DNAME	VARCHAR(128) NOT NULL	Name of the object that is dependent on this identity column or sequence.	G
DCOLNAME	VARCHAR(128) NOT NULL	Name of the identity column. Blank for SQL function rows.	G
DTYPE	CHAR(1) NOT NULL WITH DEFAULT 'I'	The type of object that is dependent on this sequence: F SQL function I Identity column X Implicit DOCID column that is created on a base table with XML blank Represents an identity column created prior to Version 8	G
BSHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	The schema name of the sequence, will be a string of length zero for an object created prior to Version 8.	G
BNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The sequence name (generated by DB2 for an identity column), will be a string of length zero for an object created prior to Version 8.	G
DSHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	The qualifier of the object that is dependent on this sequence, will be a string of length zero for an object created prior to Version 8.	G
DOWNER	VARCHAR(128) NOT NULL WITH DEFAULT	The owner of the object that is dependent on this sequence. This will be a string of length zero for an object that was created prior to Version 9.	G
DOWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	The type of owner: Blank An authorization ID L A role	G

SYSIBM.SYSSTMT table

The SYSIBM.SYSSTMT table contains one or more rows for each SQL statement of each DBRM.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the DBRM.	G
PLNAME	VARCHAR(24) NOT NULL	Name of the application plan.	G
PLCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the application plan.	G
SEQNO	SMALLINT NOT NULL	Sequence number of this row with respect to a statement of the DBRM. Rows in which the values of SEQNO, STMTNO, and SECTNO are zero are for internal use. The numbering starts with zero.	G
STMTNO	SMALLINT NOT NULL	The statement number of the statement in the source program. A statement number greater than 32767 is stored as zero. If the value is zero, see STMTNOI for the statement number. Rows in which the values of SEQNO, STMTNO, and SECTNO are zero are for internal use.	G
SECTNO	SMALLINT NOT NULL	The section number of the statement. Rows in which the values of SEQNO, STMTNO, and SECTNO are zero are for internal use.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
TEXT	VARCHAR(3800) NOT NULL WITH DEFAULT FOR BIT DATA	Text or portion of the text of the SQL statement.	S
ISOLATION	CHAR(1) NOT NULL WITH DEFAULT	Isolation level for the SQL statement: R RR (repeatable read) T RS (read stability) S CS (cursor stability) U UR (uncommitted read) L RS isolation, with a <i>lock-clause</i> X RR isolation, with a <i>lock-clause</i> blank The WITH clause was not specified on this statement. The isolation level is recorded in SYSPACKAGE.ISOLATION and in SYSPLAN.ISOLATION.	G

Column name	Data type	Description	Use
STATUS	CHAR(1) NOT NULL WITH DEFAULT	<p>Status of binding the statement:</p> <p>A Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using defaults for input variables during access path selection.</p> <p>B Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using values for input variables during access path selection.</p> <p>C Compiled - statement was bound successfully using defaults for input variables during access path selection.</p> <p>D Distributed - statement references a remote object using a three-part name. DB2 will implicitly use DRDA access either because the DBPROTOCOL bind option was not specified (defaults to DRDA), or the bind option DBPROTOCOL(DRDA) was explicitly specified. This option allows the use of three-part names with DRDA access but it requires that the package be bound at the target remote site.</p> <p>E Explain - statement is an SQL EXPLAIN statement. The explain is done at bind time using defaults for input variables during access path selection.</p> <p>F Parsed - statement did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using values for input variables during access path selection.</p> <p>G Compiled - statement bound successfully, but REOPT is specified. The statement will be rebound at execution time using values for input variables during access path selection.</p> <p>H Parsed - statement is either a data definition statement or a statement that did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using defaults for input variables during access path selection. Data manipulation statements use defaults for input variables during access path selection.</p> <p>I Indefinite - statement is dynamic. The statement will be bound at execution time using defaults for input variables during access path selection.</p>	S

Column name	Data type	Description	Use
STATUS		<p>J Indefinite - statement is dynamic. The statement will be bound at execution time using values for input variables during access path selection.</p> <p>K Control - CALL statement.</p> <p>L Bad - the statement has some allowable error. The bind continues but the statement cannot be executed.</p> <p>M Parsed - statement references a table that is qualified with SESSION and was not bound because the table reference could be for a declared temporary table that will not be defined until the package or plan is run. The SQL statement will be rebound at execution time using values for input variables during access path selection.</p> <p>blank The statement is non-executable, or was bound in a DB2 release prior to Version 5.</p>	
ACCESSPATH	CHAR(1) NOT NULL WITH DEFAULT	<p>For static statements, indicates if the access path for the statement is based on user-specified optimization hints. A value of 'H' indicates that optimization hints were used. A blank value indicates that the access path was determined without the use of optimization hints, or that there is no access path associated with the statement.</p> <p>For dynamic statements, the value is blank.</p>	G
STMTNOI	INTEGER NOT NULL WITH DEFAULT	If the value of STMTNOI is not zero, the column contains the statement number of the statement in the source program.	G
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement.	G
EXPLAINABLE	CHAR(1) NOT NULL WITH DEFAULT	<p>Contains one of the following values:</p> <p>Y Indicates that the SQL statement can be used with the EXPLAIN function and might have rows describing its access path in the owner.PLAN_TABLE.</p> <p>N Indicates that the SQL statement does not have any rows describing its access path in the owner.PLAN_TABLE.</p> <p>blank Indicates that the SQL statement was bound prior to Version 7.</p>	G
QUERYNO	INTEGER NOT NULL WITH DEFAULT -1	The query number of the SQL statement in the source program. SQL statements bound prior to Version 7 have a default value of -1. Statements bound in Version 7 or later use the value specified on the QUERYNO clause on SELECT, UPDATE, INSERT, DELETE, EXPLAIN, and DECLARE CURSOR statements. If the QUERYNO clause is not specified, the query number is set to the statement number.	G

Column name	Data type	Description	Use
PLCREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G

SYSIBM.SYSTOGROUP table

The SYSIBM.SYSTOGROUP table contains one row for each storage group.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the storage group.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the storage group.	G
VCATNAME	VARCHAR(24) NOT NULL	Name of the integrated catalog facility catalog.	G
	VARCHAR(24) NOT NULL	Not used	N
SPACE	INTEGER NOT NULL	Number of kilobytes of DASD storage allocated to the storage group as determined by the last execution of the STOSPACE utility.	G
	CHAR(5) NOT NULL	Not used	N
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the storage group.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If the STOSPACE utility was executed for the storage group, date and time when STOSPACE was last executed.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the storage group.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER STOGROUP statement was executed for the storage group. If no ALTER STOGROUP statement has been applied, ALTEREDTS has the value of CREATEDTS.	G
SPACEF	FLOAT NOT NULL WITH DEFAULT	Kilobytes of DASD storage for the storage group. The value is -1 if statistics have not been gathered. This is an updatable column.	G
DATACLAS	VARCHAR(24) NOT NULL	Name of the SMS data class. Blank if data class is not used.	G
MGMTCLAS	VARCHAR(24) NOT NULL	Name of the SMS management class. Blank if management class is not used.	G

Column name	Data type	Description	Use
STORCLAS	VARCHAR(24) NOT NULL	Name of the SMS storage class. Blank if storage class is not used.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSSTRINGS table

The SYSIBM.SYSSTRINGS table contains information about character conversion. Each row describes a conversion from one coded character set to another.

Also refer to z/OS C/C++ Programming Guide for information on the additional conversions that are supported.

Each row in the table must have a unique combination of values for its INCCSID, OUTCCSID, and IBMREQD columns. Rows for which the value of IBMREQD is N can be deleted, inserted, and updated subject to this uniqueness constraint and to the constraints imposed by a VALIDPROC defined on the table. An inserted row could have values for the INCCSID and OUTCCSID columns that match those of a row for which the value of IBMREQD is Y. DB2 then uses the information in the inserted row instead of the information in the IBM-supplied row. Rows for which the value of IBMREQD is Y cannot be deleted, inserted, or updated. For information about the use of inserted rows for character conversion, see *DB2 Installation Guide*.

DB2 has two methods for character conversions and applies them in the following order:

1. Conversions specified by the various combinations of the INCCSID and OUTCCSID columns in the SYSIBM.SYSSTRINGS catalog table.
2. Conversions provided by z/OS support for Unicode. For more information, see *z/OS Support for Unicode: Using Conversion Services*.

If neither of these methods can be used for a particular character conversion, DB2 returns an error.

Conversions with the following combinations of source CCSID and target CCSID will always use the conversion services that are provided by z/OS support for Unicode. Any rows in the SYSIBM.SYSSTRINGS table with the following CCSID combinations will be ignored:

- Either the source CCSID (INCCSID column) or the target CCSID (OUTCCSID column) is 1200 or 1208.
- The source CCSID (INCCSID column) is 17584 and the target CCSID (OUTCCSID column) is 16684.

Column name	Data type	Description	Use
INCCSID	INTEGER NOT NULL	The source CCSID for the character conversion represented by this row. The value of the source CCSID must be in the range of 1 to 65533 and must not be the same as the value for the OUTCCSID column.	G
OUTCCSID	INTEGER NOT NULL	The target CCSID for the character conversion represented by this row. The value of the target CCSID must be in the range of 1 to 65533 and must not be the same as the value for the INCCSID column.	G

Column name	Data type	Description	Use
TRANSTYPE	CHAR(2) NOT NULL	Indicates the nature of the conversion. Values can be: GG GRAPHIC to GRAPHIC MM EBCDIC MIXED to EBCDIC MIXED MS EBCDIC MIXED to SBCS PM ASCII MIXED to EBCDIC MIXED PS ASCII MIXED to SBCS SM SBCS to EBCDIC MIXED SS SBCS to SBCS MP EBCDIC MIXED to ASCII MIXED PP ASCII MIXED to ASCII MIXED SP SBCS to ASCII MIXED	G
ERRORBYTE	CHAR(1) FOR BIT DATA (Nulls are allowed)	The byte used in the conversion table as an error byte. Any non-null value that is specified for the ERRORBYTE column must not be the same as the value that is specified for the SUBBYTE column. Null indicates the absence of an error byte.	S
SUBBYTE	CHAR(1) FOR BIT DATA (Nulls are allowed)	The byte used in the conversion table as a substitution character. Any non-null value that is specified for the SUBBYTE column must not be the same as the value that is specified for the ERRORBYTE column. Null indicates the absence of a substitution character.	S
TRANSPROC	VARCHAR(24) NOT NULL WITH DEFAULT	The name of a module or blanks. A nonblank value must conform to the rules for z/OS program names. If IBMREQD is 'N', a nonblank value is the name of a conversion procedure provided by the user. The first five characters of the name of a user-provided conversion procedure must not be 'DSNXV'; these characters are used to distinguish user-provided conversion procedures from DB2 modules that contain DBCS conversion tables. If IBMREQD is 'Y', a nonblank value is the name of a DB2 module that contains DBCS conversion tables.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape.	G
		Value Meaning	
		B Version 1R3 dependency indicator, not from the machine-readable material (MRM) tape	
		C Version 2R1 dependency indicator, not from MRM tape	
		D Version 2R2 dependency indicator, not from MRM tape	
		E Version 2R3 dependency indicator, not from MRM tape	
		F Version 3R1 dependency indicator, not from MRM tape	
		G Version 4 dependency indicator, not from MRM tape	
		H Version 5 dependency indicator, not from MRM tape	
		I Version 6 dependency indicator, not from MRM tape	
		J Version 6 dependency indicator, not from MRM tape	
		K Version 7 dependency indicator, not from MRM tape	
		L Version 8 dependency indicator, not from MRM tape	
		M Version 9 dependency indicator, not from MRM tape	
		N Not from MRM tape, no dependency	
TRANSTAB	VARCHAR(256) FOR BIT DATA NOT NULL WITH DEFAULT	Either a 256-byte conversion table or an empty (0 length) string.	S

SYSIBM.SYSSYNONYMS table

The SYSIBM.SYSSYNONYMS table contains one row for each synonym of a table or view.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Synonym for the table or view.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the synonym.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table or view.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the synonym.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the synonym. The value is '0001-01.01.00.00.00.000000' for synonyms created in a DB2 release prior to Version 5.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSTABAUTH table

The SYSIBM.SYSTABAUTH table records the privileges that users hold on tables and views.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID or role of the user who granted the privileges. Could also be PUBLIC, or PUBLIC followed by an asterisk. ⁴⁴ .	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID or role of the user who holds the privileges or the name of an application plan or package that uses the privileges. PUBLIC for a grant to PUBLIC. PUBLIC followed by an asterisk for a grant to PUBLIC AT ALL LOCATIONS.	G
GRANTEETYPE	CHAR(1) NOT NULL	Type of grantee: blank An authorization ID L Role P An application plan or a package. The grantee is a package if COLLID is not blank.	G
DBNAME	VARCHAR(24) NOT NULL	If the privileges were received from a user with DBADM, DBCTRL, or DBMAINT authority, DBNAME is the name of the database on which the GRANTOR has that authority. Otherwise, DBNAME is blank.	G
SCREATOR	VARCHAR(128) NOT NULL	If the row of SYSIBM.SYSTABAUTH was created as a result of a CREATE VIEW statement, SCREATOR is the schema of a table or view referred to in the CREATE VIEW statement. Otherwise, SCREATOR is the same as TCREATOR.	G
STNAME	VARCHAR(128) NOT NULL	If the row of SYSIBM.SYSTABAUTH was created as a result of a CREATE TABLE statement or a materialized query table, STNAME is the name of a table or view referred to in the fullselect of the CREATE TABLE statement.	G
TCREATOR	VARCHAR(128) NOT NULL	The schema of the table or view.	G
TTNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. blank Not applicable C DBCTRL D DBADM L SYSCTRL M DBMAINT S SYSADM	G
	CHAR(12) NOT NULL	Internal use only	I

44. PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS.

Column name	Data type	Description	Use
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
UPDATECOLS	CHAR(1) NOT NULL	The value of this column is blank if the value of UPDATEAUTH applies uniformly to all columns of the table or view. The value is an asterisk (*) if the value of UPDATEAUTH applies to some columns but not to others. In this case, rows will exist in SYSIBM.SYSCOLAUTH with matching timestamps and PRIVILEGE = blank. These rows list the columns on which update privileges have been granted.	G
ALTERAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can alter the table: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
DELETEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can delete rows from the table or view: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
INDEXAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create indexes on the table: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
INSERTAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can insert rows into the table or view: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
SELECTAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can select rows from the table or view: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
UPDATEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can update rows of the table or view: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
	VARCHAR(128) NOT NULL WITH DEFAULT	Not used	N
	VARCHAR(128) NOT NULL WITH DEFAULT	Not used	N

Column name	Data type	Description	Use
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	If the GRANTEE is a package, its collection name. Otherwise, the value is blank.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT FOR BIT DATA	If the GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, the value is blank.	S
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
REFERENCESAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can create or drop referential constraints in which the table is a parent. blank Privilege is not held G Privilege held with the GRANT option Y Privilege held without the GRANT option	G
REFCOLS	CHAR(1) NOT NULL WITH DEFAULT	The value of this column is blank if the value of REFERENCESAUTH applies uniformly to all columns of the table. The value is an asterisk(*) if the value of REFERENCESAUTH applies to some columns but not to others. In this case, rows will exist in SYSIBM.SYSCOLAUTH with PRIVILEGE = R and matching timestamps that list the columns on which reference privileges have been granted.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G
TRIGGERAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can create triggers in which the table is named as the subject table: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G

SYSIBM.SYSTABCONST table

The SYSIBM.SYSTABCONST table contains one row for each unique constraint (primary key or unique key) created in DB2 Version 7 or later.

Column name	Data type	Description	Use
CONSTNAME	VARCHAR(128) NOT NULL	Name of the constraint.	G
TBCREATOR	VARCHAR(128) NOT NULL	The schema of the table on which the constraint is defined.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table on which the constraint is defined.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID under which the constraint was created.	G
TYPE	CHAR(1) NOT NULL	Type of constraint: P Primary key U Unique key	G
IXOWNER	VARCHAR(128) NOT NULL	The schema of the index enforcing the constraint or blank if index has not been created yet.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the index enforcing the constraint or blank if index has not been created yet.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the statement to create the constraint was executed.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
COLCOUNT	SMALLINT NOT NULL	Number of columns in the constraint.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSTABLEPART table

The SYSIBM.SYSTABLEPART table contains one row for each nonpartitioned table space and one row for each partition of a partitioned table space.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number; 0 if table space is not partitioned.	G
TSNAME	VARCHAR(24) NOT NULL	Name of the table space.	G
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the table space.	G
IXNAME	VARCHAR(128) NOT NULL	Name of the partitioning index. This column is blank unless this is a table that uses index-controlled partitioning.	G
IXCREATOR	VARCHAR(128) NOT NULL	The schema of the partitioning index. This column is blank unless this is a table that uses index-controlled partitioning.	G
PQTY	INTEGER NOT NULL	For user-managed data sets, the value is the primary space allocation in units of 4 KB storage blocks or -1. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike PQTY, however, PRIQTY asks for space in 1 KB units. A value of -1 indicates that either of the following cases is true: <ul style="list-style-type: none"> • PRIQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements. • -1 was the most recently specified value for PRIQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement. 	G
SQTY	SMALLINT NOT NULL	For user-managed data sets, the value is the secondary space allocation in units of 4 KB storage blocks or -1. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike SQTY, however, SECQTY asks for space in 1 KB units. A value of -1 indicates that either of the following cases is true: <ul style="list-style-type: none"> • SECQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements. • -1 was the most recently specified value for SECQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement. If the value does not fit into the column, the value of the column is 32767. See the description of column SECQTYI.	G
STORTYPE	CHAR(1) NOT NULL	Type of storage allocation: E Explicit (storage group not used) I Implicit (storage group used)	G

Column name	Data type	Description	Use
STORNAME	VARCHAR(128) NOT NULL	Name of storage group used for space allocation. Blank if storage group not used.	G
VCATNAME	VARCHAR(24) NOT NULL	Name of integrated catalog facility catalog used for space allocation.	G
CARD	INTEGER NOT NULL	Number of rows in the table space or partition or, if the table space is a LOB table space, the number of LOBs in the table space. The value is '2 147 483 647' if the number of rows is greater than or equal to '2 147 483 647'. The value is -1 if statistics have not been gathered.	G
FARINDREF	INTEGER NOT NULL	Number of rows that have been relocated far from their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
NEARINDREF	INTEGER NOT NULL	Number of rows that have been relocated near their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
PERCACTIVE	SMALLINT NOT NULL	Percentage of space occupied by rows of data from active tables. The value is -1 if statistics have not been gathered. The value is -2 if the table space is a LOB table space.	S
PERCDROP	SMALLINT NOT NULL	Percentage of space occupied by rows of dropped tables. The value is -1 if statistics have not been gathered. The value is 0 for segmented table spaces. Not applicable if the table is an auxiliary table.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
LIMITKEY	VARCHAR(765) NOT NULL	The high value of the partition in external format. If the table space was converted from index-controlled partitioning to table-controlled partitioning, the value is the highest possible value for an ascending key, or the lowest possible value for a descending key. If the table space is not partitioned, the value is an empty string.	G
FREEPAGE	SMALLINT NOT NULL	Number of pages loaded before a page is left as free space.	G
PCTFREE	SMALLINT NOT NULL	Percentage of each page left as free space.	G
CHECKFLAG	CHAR(1) NOT NULL WITH DEFAULT	blank The table space is not a partition, or does not contain rows that might violate referential constraints, check constraints, or both. C The table space partition is in a check pending status and there are rows in the table that can violate referential constraints, check constraints, or both.	G
	CHAR(4) NOT NULL WITH DEFAULT FOR BIT DATA	Not used	N

Column name	Data type	Description	Use
SPACE	INTEGER NOT NULL WITH DEFAULT	<p>Number of kilobytes of DASD storage allocated to the table space partition, as determined by the last execution of the STOSPACE utility or RUNSTATS utility.</p> <p>0 The STOSPACE or RUNSTATS utility has not been run.</p> <p>-1 The table space was defined with the DEFINE NO clause, which defers the physical creation of the data sets until data is first inserted into one of the partitions, and data has yet to be inserted.</p> <p>non-zero or non-negative value An auxiliary table in the LOB table space. The value is if The value is updated by STOSPACE if the table space is related to a storage group. The value is updated by RUNSTATS if the utility is executed as RUNSTATS TABLESPACE with UPDATE(ALL) or UPDATE(SPACE).</p>	G
COMPRESS	CHAR(1) NOT NULL WITH DEFAULT	<p>Indicates the following:</p> <ul style="list-style-type: none"> For a table space partition, whether the COMPRESS attribute for the partition is YES. For a nonpartitioned table space, whether the COMPRESS attribute is YES for the table space. <p>Values for the column can be:</p> <p>Y Compression is defined for the table space</p> <p>blank No compression</p>	G
PAGESAVE	SMALLINT NOT NULL WITH DEFAULT	<p>Percentage of pages saved in the table space or partition as a result of defining the table space with COMPRESS YES or other compression routines. For example, a value of 25 indicates a savings of 25 percent, so that the pages required are only 75 percent of what would be required without data compression. The calculation includes overhead bytes for each row, the bytes required for dictionary, and the bytes required for the current FREEPAGE and PCTFREE specification for the table space or partition. This calculation is based on an average row length, and the result varies depending on the actual lengths of the rows. The value is 0 if there are no savings from using data compression, or if statistics have not been gathered. The value can be negative, if for example, data compression causes an increase in the number of pages in the data set.</p>	S
STATTIME	TIMESTAMP NOT NULL WITH DEFAULT	<p>If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.0000000'.</p>	G
GBPCACHE	CHAR(1) NOT NULL WITH DEFAULT	<p>Group buffer pool cache option specified for this table space or table space partition.</p> <p>A Changed and unchanged pages are cached in the group buffer pool.</p> <p>N No data is cached in the group buffer pool.</p> <p>S Only changed system pages, such as space map pages that do not contain actual data values, are cached in the group buffer pool.</p> <p>blank Only changed pages are cached in the group buffer pool.</p>	G

Column name	Data type	Description	Use
CHECKRID5B	CHAR(5) NOT NULL WITH DEFAULT FOR BIT DATA	Blank if the table or partition is not in a check pending status (CHECKFLAG is blank), or if the table space is not partitioned. Otherwise, the RID of the first row of the table space partition that can violate referential constraints, check constraints, or both; or the value is X'0000000000', indicating that any row can violate referential constraints.	S
TRACKMOD	CHAR(1) NOT NULL WITH DEFAULT	Whether to track the page modifications in the space map pages: N No blank Yes	G
EPOCH	INTEGER NOT NULL WITH DEFAULT	A number that increments whenever a utility operation that changes the location of rows in a table occurs.	G
SECQTYI	INTEGER NOT NULL WITH DEFAULT	Secondary space allocation in units of 4KB storage. For user-managed data sets, the value is the secondary space allocation in units of 4KB blocks.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of rows in the table space or partition, or if the table space is a LOB table space, the number of LOBs in the table space. The value is -1 if statistics have not been gathered.	G
IPREFIX	CHAR(1) NOT NULL WITH DEFAULT 'I'	The first character of the instance qualifier for the data set name for the table space or partition. 'I' or 'J' are the only valid characters for this field. The default is 'I'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER TABLESPACE statement was executed for the table space or partition. If no ALTER TABLESPACE statement has been applied, the value is '0001-01-01.00.00.00.000000'.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. The value might be non-zero for an auxiliary table in the LOB table space. This is an updatable column.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Number of data sets. The value is -1 if statistics have not been gathered. This is an updatable column.	G
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is -1 if statistics have not been gathered. This is an updatable column. This value is only for the last DSNUM for the object.	G
LOGICAL_PART	SMALLINT NOT NULL WITH DEFAULT	The logical partition (logical ascending or descending order) for table spaces created with either table-controlled partitioning or index-controlled partitioning. The physical partition number is kept in column PART and is zero for partitioned table spaces created prior to Version 8 and for nonpartitioned table spaces.	G

Column name	Data type	Description	Use
LIMITKEY_ INTERNAL	VARCHAR(512) NOT NULL WITH DEFAULT FOR BIT DATA	The highest value of the limit key of the partition in an internal format. If the uses index-controlled partitioning instead of table-controlled partitioning or the table is not partitioned, the value is an empty string. If the table space was converted from index-controlled partitioning to table-controlled partitioning, the value is the highest possible value for an ascending key, or the lowest possible value for a descending key. If any column of the key has a field procedure, the internal format is the encoded form of the value.	G
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number of the oldest format of data in the table part and any image copies at the part level.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the partition was created.	G
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space or part. If the table space or part is compressed, the value is the compressed row length. If the table space or part is not compressed, the value is the uncompressed row length. The value is -1 if statistics have not been gathered.	G
FORMAT	CHAR(1) NOT NULL WITH DEFAULT	Indicates the format of the rows in the table space or partition: R Indicates reordered row format blank Indicates basic row format or a LOB tablespace	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
REORG_LR_TS	TIMESTAMP NOT NULL WITH DEFAULT	The time when the REORG or LOAD REPLACE utility last occurred. The default value is '0001-01-01.00.00.00.000000'.	G

SYSIBM.SYSTABLEPART_HIST table

The SYSIBM.SYSTABLEPART_HIST table contains rows from SYSTABLEPART.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number. 0 if table space is not partitioned.	G
TSNAME	VARCHAR(24) NOT NULL	Name of the table space.	G
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the table space.	G
PQTY	INTEGER NOT NULL	<p>For user-managed data sets, the value is the primary space allocation in units of 4 KB storage blocks or -1.</p> <p>For user-specified values of PRIQTY other than -1, the value is set to the primary space allocation only if RUNSTATS TABLESPACE with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. PQTY is based on a value of PRIQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike PQTY, however, PRIQTY asks for space in 1 KB units.</p> <p>A value of -1 indicates that either of the following cases is true:</p> <ul style="list-style-type: none">• PRIQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements.• -1 was the most recently specified value for PRIQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement. <p>If a storage group is not used, the value is 0.</p>	G

Column name	Data type	Description	Use
SECQTYI	INTEGER NOT NULL	<p>For user-managed data sets, the value is the secondary space allocation in units of 4 KB storage blocks or -1.</p> <p>For user-specified values of SECQTY other than -1, the value is set to the secondary space allocation only if RUNSTATS TABLESPACE with UPDATE(ALL) or UPDATE(SPACE) is executed; otherwise, the value is zero. SQTY is based on a value of SECQTY in the appropriate CREATE or ALTER TABLESPACE statement. Unlike SQTY, however, SECQTY asks for space in 1 KB units.</p> <p>A value of -1 indicates that either of the following cases is true:</p> <ul style="list-style-type: none"> SECQTY was not specified for a CREATE TABLESPACE statement or for any subsequent ALTER TABLESPACE statements. -1 was the most recently specified value for SECQTY, either on the CREATE TABLESPACE statement or a subsequent ALTER TABLESPACE statement. <p>If a storage group is not used, the value is 0.</p>	G
FARINDREF	INTEGER NOT NULL WITH DEFAULT -1	Number of rows that have been relocated far from their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
NEARINDREF	INTEGER NOT NULL WITH DEFAULT -1	Number of rows that have been relocated near their original page. The value is -1 if statistics have not been gathered. Not applicable if the table space is a LOB table space.	S
PERCACTIVE	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of space occupied by rows of data from active tables. The value is -1 if statistics have not been gathered. The value is -2 if the table space is a LOB table space.	S
PERCDROP	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of space occupied by rows of dropped tables. The value is -1 if statistics have not been gathered. The value is 0 for segmented table spaces. Not applicable if the table is an auxiliary table.	S
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of kilobytes of DASD storage allocated to the table space partition. The value is -1 if statistics have not been gathered.	G

Column name	Data type	Description	Use
PAGESAVE	SMALLINT NOT NULL	<p>Percentage of pages saved in the table space or partition as a result of defining the table space with COMPRESS YES or other compression routines. For example, a value of 25 indicates a savings of 25 percent, so that the pages required are only 75 percent of what would be required without data compression.</p> <p>The calculation includes overhead bytes for each row, the bytes required for dictionary, and the bytes required for the current FREEPAGE and PCTFREE specification for the table space or partition. This calculation is based on an average row length, and the result varies depending on the actual lengths of the rows.</p> <p>The value is 0 if there are no savings from using data compression, or if statistics have not been gathered. The value can be negative, if for example, data compression causes an increase in the number of pages in the data set.</p>	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of rows in the table space or partition, or if the table space is a LOB table space, the number of LOBS in the table space. The value is '-1' if statistics have not been gathered.	S
EXTENTS	INTEGER NOT NULL WITH DEFAULT -1	Number of data set extents. The value is '-1' if statistics have not been gathered. This value is only for the last DSNUM for the object.	G
DSNUM	INTEGER NOT NULL WITH DEFAULT -1	Data set number within the table space. For partitioned table spaces, this value corresponds to the partition number for a single partition copy, or 0 for a copy of an entire partitioned table space or index space. The value is '-1' if statistics have not been gathered.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space or part. If the table space or part is compressed, the value is the compressed row length. If the table space or part is not compressed, the value is the uncompressed row length. The value is '-1' if statistics have not been gathered.	G

SYSIBM.SYSTABLES table

The SYSIBM.SYSTABLES table contains one row for each table, view, or alias.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the table, view, or alias.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the table, view, or alias.	G
TYPE	CHAR(1) NOT NULL	Type of object: A Alias C Clone table G Created global temporary table M Materialized query table P Implicit table created for XML columns T Table V View X Auxiliary table	G
DBNAME	VARCHAR(24) NOT NULL	For a table, or a view of tables, the name of the database that contains the table space named in TSNAME. For a created temporary table, an alias, or a view of a view, the value is DSNDB06.	G
TSNAME	VARCHAR(24) NOT NULL	For a table, or a view of one table, the name of the table space that contains the table. For a view of more than one table, the name of a table space that contains one of the tables. For a created temporary table, the value is SYSPKAGE. Although SYSPKAGE is used as the value, created temporary tables are not stored in the SYSPKAGE table space. For a view of a view, the value is SYSVIEWS. For an alias, it is SYSDBAUT.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database; 0 if the row describes a view, alias, or created temporary table. Non-zero if the view has an INSTEAD OF trigger defined.	S
OBID	SMALLINT NOT NULL	Internal identifier of the table; 0 if the row describes a view, an alias, or a created temporary table. Non-zero if the view has an INSTEAD OF trigger defined.	S
COLCOUNT	SMALLINT NOT NULL	Number of columns in the table or view. The value is 0 if the row describes an alias.	G
EDPROC	VARCHAR(24) NOT NULL	Name of the edit procedure; blank if the row describes a view or alias or a table without an edit procedure.	G
VALPROC	VARCHAR(24) NOT NULL	Name of the validation procedure; blank if the row describes a view or alias or a table without a validation procedure.	G
CLUSTERTYPE	CHAR(1) NOT NULL	Whether RESTRICT ON DROP applies: blank No Y Yes. Neither the table nor any table space or database that contains the table can be dropped.	G
	INTEGER NOT NULL	Not used	N

Column name	Data type	Description	Use
	INTEGER NOT NULL	Not used	N
NPAGES	INTEGER NOT NULL	Total number of pages on which rows of the table appear. The value is -1 if statistics have not been gathered, or the row describes a view, an alias, a created temporary table, or an auxiliary table. This is an updatable column.	S
PCTPAGES	SMALLINT NOT NULL	Percentage of active table space pages that contain rows of the table. A page is termed active if it is formatted for rows, regardless of whether it contains any. If the table space is segmented, the percentage is based on the number of active pages in the set of segments assigned to the table. The value is -1 if statistics have not been gathered, or the row describes a view, alias, created temporary table, or auxiliary table. This is an updatable column.	S
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G
PARENTS	SMALLINT NOT NULL	Number of relationships in which the table is a dependent. The value is 0 if the row describes a view, an alias, a created temporary table, or a materialized query table.	G
CHILDREN	SMALLINT NOT NULL	Number of relationships in which the table is a parent. The value is 0 if the row describes a view, an alias, a created temporary table, or a materialized query table.	G
KEYCOLUMNS	SMALLINT NOT NULL	Number of columns in the primary key of the table. The value is 0 if the row describes a view, an alias, or a created temporary table.	G
RECLENGTH	SMALLINT NOT NULL	<p>For user tables, the maximum length of any record in the table. Length is $8+N+L$, where:</p> <ul style="list-style-type: none"> The number 8 accounts for the header (6 bytes) and the ID map entry (2 bytes). N is 10 if the table has an edit procedure, or 0 otherwise. L is the sum of the maximum column lengths. In determining the maximum length of a column, take into account whether the column allows nulls and the data type of the column. If the column can contain nulls and is not a LOB or ROWID column, add 1 byte for a null indicator. Use 4 bytes for the length of a LOB column and 19 bytes for the length of a ROWID column. If the column has a varying-length data type (for example, VARCHAR, CLOB, or BLOB), add 2 bytes for a length indicator. For more information on column lengths, see “Data types” on page 69. <p>The value is 0 if the row describes a view, alias, or auxiliary table. For maximum row and record sizes, see Maximum record size.</p>	G

Column name	Data type	Description	Use
STATUS	CHAR(1) NOT NULL	Indicates the status of the table definition: I The definition of the table is incomplete. The TABLESTATUS column indicates the reason for the table definition being incomplete. R An error occurred when an attempt was made to regenerate the internal representation of the view. X The table has a unique constraint (primary key or unique key) and the table definition is complete. blank The table has no unique constraint (primary key or unique key), the table is a catalog table, or the row describes a view or alias. The definition of the table, view, or alias is complete.	G
KEYOBID	SMALLINT NOT NULL	Internal DB2 identifier of the index that enforces uniqueness of the primary key of the table; 0 if not applicable.	S
LABEL	VARCHAR(90) NOT NULL	The label as given by a LABEL statement; otherwise, the value is an empty string.	G
CHECKFLAG	CHAR(1) NOT NULL WITH DEFAULT	C The table space that contains the table is in a check pending status. One of the following conditions is true: <ul style="list-style-type: none"> There are rows in the table that violate referential constraints, check constraints, or both The table is a materialized query table that might contain inconsistent data blank Indicates the following: <ul style="list-style-type: none"> The table contains no rows that violate referential constraints, check constraints, or both The table is a materialized query table that contains consistent data The row describes a view, an alias, or a temporary table 	G
I CHECKRID	CHAR(4) NOT NULL WITH DEFAULT FOR BIT DATA	A value of 'FFFFFF00' in this column indicates that the edit procedure on this table is defined without row attribute sensitivity. Any other value indicates that the edit procedure is defined with row attribute sensitivity.	G
AUDITING	CHAR(1) NOT NULL WITH DEFAULT	Value of the audit option: A AUDIT ALL C AUDIT CHANGE blank AUDIT NONE, or the row describes a view, an alias, or a created temporary table.	G
CREATEDBY	VARCHAR(128) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the table, view, or alias.	G

Column name	Data type	Description	Use
LOCATION	VARCHAR(128) NOT NULL WITH DEFAULT	Location name of the object of an alias, or the name of the accelerator server that is used for a materialized query table. Blank for a table, a view, for an alias that was not defined with a three-part object name, or a materialized query table that is not accelerated.	G
TBCREATOR	VARCHAR(128) NOT NULL WITH DEFAULT	<ul style="list-style-type: none"> For an alias, the schema of the referred to table or view For a base table that is involved in a clone relationship, the name of the creator of the clone table For a clone table that is involved in a clone relationship, the name of the creator of the base table For a view, the name of the underlying table. Otherwise, TBCREATOR is blank 	G
TBNAME	VARCHAR(128) NOT NULL WITH DEFAULT	<ul style="list-style-type: none"> For an alias, the name for the referred to table or view For a base table that is involved in a clone relationship, the name of the clone table For a clone table that is involved in a clone relationship, the name of the base table For a view, the name of the underlying table. Otherwise, TBNAME is blank 	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the table, view, or alias	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	For a table, the time when the latest ALTER TABLE statement was applied. If no ALTER TABLE statement has been applied, or if the row is for an alias, ALTEREDTS has the value of CREATEDTS. For a view, the time when the last ALTER VIEW REGENERATE statement was applied.	G
DATA_CAPTURE	CHAR(1) NOT NULL WITH DEFAULT	Records the value of the DATA CAPTURE option for a table: blank No Y Yes For a created temporary table, DATA_CAPTURE is always blank.	G
RBA1	CHAR(6) NOT NULL WITH DEFAULT FOR BIT DATA	The log RBA when the table was created. Otherwise, RBA1 is X'000000000000', indicating that the log RBA is not known, or that the object is a view, an alias, or a created temporary table. In a data sharing environment, RBA1 is the LRSN (Log Record Sequence Number) value.	S
RBA2	CHAR(6) NOT NULL WITH DEFAULT FOR BIT DATA	The log RBA when the table was last altered. Otherwise, RBA2 is X'000000000000' indicating that the log RBA is not known, or that the object is a view, an alias, or a created temporary table. RBA1 will equal RBA2 if the table has not been altered. In a data sharing environment, RBA2 is the LRSN (Log Record Sequence Number) value.	S
PCTROWCOMP	SMALLINT NOT NULL WITH DEFAULT	Percentage of rows compressed within the total number of active rows in the table. This includes any row in a table space that is defined with COMPRESS YES. The value is -1 if statistics have not been gathered, or the row describes a view, alias, created temporary table, or auxiliary table. This is an updatable column.	S

Column name	Data type	Description	Use
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. For a created temporary table, the value of STATSTIME is always the default value. This is an updatable column.	G
CHECKS	SMALLINT NOT NULL WITH DEFAULT	Number of check constraints defined on the table. The value is 0 if the row describes a view, an alias, a created temporary table, a materialized query table, or if no constraints are defined on the table.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Total number of rows in the table or total number of LOBs in an auxiliary table. The value is -1 if statistics have not been gathered or the row describes a view, alias, or created temporary table. This is an updatable column.	S
CHECKRID5B	CHAR(5) NOT NULL WITH DEFAULT FOR BIT DATA	Blank if the table or partition is not in a check pending status (CHECKFLAG is blank), if the table space is not partitioned, or if the table is a created temporary table. Otherwise, the RID of the first row of the table space partition that can violate referential constraints, check constraints, or both; or the value is X'0000000000', indicating that any row can violate referential constraints.	S
ENCODING_SCHEME	CHAR(1) NOT NULL WITH DEFAULT 'E'	Encoding scheme for tables, views, and local aliases: E EBCDIC A ASCII M Multiple CCSID set or multiple encoding schemes U UNICODE blank For remote aliases The value is 'E' for tables in non work file databases and blank for tables in work file databases created prior to Version 5 or the default database, DSNDB04.	G
TABLESTATUS	VARCHAR(30) NOT NULL WITH DEFAULT	Not applicable for tables that were created prior to DB2 for z/OS Version 5. Indicates the reason for an incomplete table definition: L Definition is incomplete because an auxiliary table or auxiliary index has not been defined for a LOB column. P Definition is incomplete because the table lacks a primary index. R Definition is incomplete because the table lacks a required index on a row ID. U Definition is incomplete because the table lacks a required index on a unique key. V An error occurred when an attempt was made to regenerate the internal representation of the view. blank Definition is complete.	G
NPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Number of pages used by the table. The value is -1 if statistics have not been gathered or this is an auxiliary table. This is an updatable column.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. The value might be non-zero for an auxiliary table in the LOB table space. This is an updatable column.	G

Column name	Data type	Description	Use
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space. If the table space is compressed, the value is the compressed row length. If the table space is not compressed, the value is the uncompressed row length. The value is -1 if statistics have not been gathered.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
NUM_DEP_MQTS	SMALLINT NOT NULL WITH DEFAULT	Number of dependent materialized query tables. The value is zero if the row describes an alias or a created temporary table, or if no materialized query tables are defined on the table.	G
VERSION	SMALLINT NOT NULL WITH DEFAULT	The version of the data row format for this table. <ul style="list-style-type: none"> A value of zero indicates that a version-creating alter operation has never occurred against this table. A value of -1 indicates that the view has been regenerated because a column of the base table has been altered. A value of 800 indicates that a successful CREATE VIEW or ALTER VIEW statement has occurred against this table in Version 8 or later. 	G
PARTKEYCOLNUM	SMALLINT NOT NULL WITH DEFAULT	The number of columns in the partitioning key. This value is zero for tables that do not have partitioning or use index-controlled partitioning. The value is non-zero for tables that use table-controlled partitioning.	G
SPLIT_ROWS	CHAR(1) NOT NULL WITH DEFAULT	Value is blank, except for VOLATILE tables, which will have 'Y' in the field to indicate to DB2 to use index access on this table whenever possible.	G
SECURITY_LABEL	CHAR(1) NOT NULL	This column is only meaningful if the TYPE column is a T (for table) or M (for materialized query table). The value indicates whether the table has multilevel security: Blank The table does not have multilevel security. R The table has multilevel security with row granularity.	G
OWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the table, view, or alias, blank for tables, views or aliases that were created in a DB2 release prior to Version 9.	G
APPEND	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the APPEND option is specified for the table. Y The APPEND option is specified. N The APPEND option is not specified.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G

SYSIBM.SYSTABLESPACE table

The SYSIBM.SYSTABLESPACE table contains one row for each table space.

Column name	Data type	Description	Use
NAME	VARCHAR(24) NOT NULL	Name of the table space.	G
CREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the table space.	G
DBNAME	VARCHAR(24) NOT NULL	Name of the database that contains the table space.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database which contains the table space.	S
OBID	SMALLINT NOT NULL	Internal identifier of the table space file descriptor.	S
PSID	SMALLINT NOT NULL	Internal identifier of the table space page set descriptor.	S
BPOOL	CHAR(8) NOT NULL	Name of the buffer pool used for the table space.	G
PARTITIONS	SMALLINT NOT NULL	Number of partitions of the table space; 0 if the table space is not partitioned.	G
LOCKRULE	CHAR(1) NOT NULL	Lock size of the table space: A Any L Large object (LOB) P Page R Row S Table space T Table X implicitly created XML table space	G
PGSIZE	SMALLINT NOT NULL	Size of pages in the table space in kilobytes.	G
ERASERULE	CHAR(1) NOT NULL	Whether the data sets are to be erased when dropped. The value is meaningless if the table space is partitioned. N No erase Y Erase	G
STATUS	CHAR(1) NOT NULL	Availability status of the table space: A Available C Definition is incomplete because the table space does not use table-controlled partitioning and a partitioning index has not been created. P Table space is in a check pending status. S Table space is in a check pending status with the scope less than the entire table space. T Definition is incomplete because no table has been created.	G

Column name	Data type	Description	Use
IMPLICIT	CHAR(1)	Whether the table space was created implicitly:	G
	NOT NULL	N No Y Yes	
NTABLES	SMALLINT	Number of tables defined in the table space.	G
	NOT NULL		
NACTIVE	INTEGER	Number of active pages in the table space. A page is termed active if it is formatted for rows, even if it currently contains none. The value is 0 if statistics have not been gathered. This is an updatable column.	S
	NOT NULL		
	VARCHAR(24)	Not used	N
	NOT NULL		
CLOSERULE	CHAR(1)	Whether the data sets are candidates for closure when the limit on the number of open data sets is reached.	G
	NOT NULL	N No Y Yes	
SPACE	INTEGER	Number of kilobytes of DASD storage allocated to the table space, as determined by the last execution of the STOSPACE utility. The value is 0 if the table space is not related to a storage group, or if STOSPACE has not been run. If the table space is partitioned, the value is the total kilobytes of DASD storage allocated to all partitions that are storage group defined.	G
	NOT NULL		
IBMREQD	CHAR(1)	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
	NOT NULL		
	VARCHAR(54)		
	VARCHAR(24)	Internal use only	I
	NOT NULL		
SEGSIZE	SMALLINT	Number of pages in each segment of a segmented table space. The value is 0 if the table space is not segmented.	G
	NOT NULL WITH DEFAULT		
CREATEDBY	VARCHAR(128)	Primary authorization ID of the user who created the table space.	G
	NOT NULL WITH DEFAULT		
STATSTIME	TIMESTAMP	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. This is an updatable column.	G
	NOT NULL WITH DEFAULT		

Column name	Data type	Description	Use
LOCKMAX	INTEGER	<p>The maximum number of locks per user to acquire for the table or table space before escalating to the next locking level.</p> <p>0 Lock escalation does not occur.</p> <p><i>n</i> <i>n</i>, where <i>n</i> > 0, is the maximum number of locks (row, page, or LOB locks for the table or table space) an application process can acquire before lock escalation occurs.</p> <p>-1 Represents LOCKMAX SYSTEM. The value of field LOCKS PER TABLE(SPACE) on installation panel DSNTIPJ determines lock escalation. If the value of the field is 0, lock escalation does not occur. If the value is <i>n</i>, where <i>n</i> > 0, lock escalation occurs as it does for LOCKMAX <i>n</i>.</p>	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT	<p>The type of table space:</p> <p>blank The table space was created without the LOB or MEMBER CLUSTER options. If the DSSIZE column is zero, the table space is not greater than 64 gigabytes.</p> <p>I The table space was defined with the MEMBER CLUSTER option and is not greater than 64 gigabytes.</p> <p>G The table space was defined with the MAXPARTITIONS option (a partitioned-by-growth table space) with the underlying structure of a universal table space.</p> <p>K The table space was defined with the MEMBER CLUSTER option and can be greater than 64 gigabytes.</p> <p>L The table space can be greater than 64 gigabytes.</p> <p>O The table space was defined with the LOB option (the table space is a LOB table space).</p> <p>P Implicit table space created for XML columns.</p> <p>R Range-partitioned universal table space.</p>	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the table space. If the table space was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER TABLESPACE statement was executed for the table space. If no ALTER TABLESPACE statement has been applied, ALTEREDTS has the value of CREATEDTS. If the index was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ENCODING_SCHEME	CHAR(1) NOT NULL WITH DEFAULT 'E'	<p>Default encoding scheme for the table space:</p> <p>E EBCDIC</p> <p>A ASCII</p> <p>U UNICODE</p> <p>blank For table spaces in a work file database or a TEMP database (a database that was created AS TEMP, which is for declared temporary tables.)</p> <p>The value is 'E' for tables in non work file databases and blank for tables in work file databases created prior to Version 5 or the default database, DSNDB04.</p>	G

Column name	Data type	Description	Use
SBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default SBCS CCSID for the table space. For a table space in a work file database, a TEMP database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
DBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default DBCS CCSID for the table space. For a table space in a work file database, a TEMP database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
MIXED_CCSID	INTEGER NOT NULL WITH DEFAULT	Default mixed CCSID for the table space. For a table space in a work file database, a TEMP database, or a database created in a DB2 release prior to Version 5, the value is 0.	G
MAXROWS	SMALLINT NOT NULL DEFAULT 255	The maximum number of rows that DB2 will place on a data page. The default value is 255. For a LOB table space, the value is 0 to indicate that the column is not applicable.	G
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
LOG	CHAR(1) NOT NULL WITH DEFAULT 'Y'	Whether the changes to a table space are to be logged. N This table space has the NOT LOGGED attribute. Undo and redo logging for the table space and all indexes for tables in the table space is suppressed. Logging is also suppressed for the auxiliary indexes for all auxiliary tables associated with tables in the table space. Y This table space has the LOGGED attribute. Normal logging is associated with modifications to this table space, all indexes for tables in this table space, and all auxiliary indexes for all auxiliary tables associated with tables in the table space. X This LOB or XML table space has the NOT LOGGED attribute. Undo and redo logging for the table space is suppressed. Also, the logging attribute for this LOB or XML table space is linked to the logging attribute of the associated base table space and might not be able to be altered independently. If the logging attribute of the base table space is altered to LOGGED, the logging attribute of the LOB or XML table space will also be altered to LOGGED.	G
NACTIVEF	FLOAT NOT NULL WITH DEFAULT -1	Number of active pages in the table space. A page is termed active if it is formatted for rows, even if it currently contains none. The value is -1 if statistics have not been gathered. This is an updatable column.	S
DSSIZE	INTEGER NOT NULL WITH DEFAULT	Maximum size of a data set in kilobytes.	G

Column name	Data type	Description	Use
OLDEST_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number of the oldest format of data in the table space and any image copies.	G
CURRENT_VERSION	SMALLINT NOT NULL WITH DEFAULT	The version number describing the newest format of data in the table space. A zero indicates that the table space has never had versioning. After the version number reaches the maximum value, the number wraps back to one.	G
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average length of rows for the tables in the table space or part. If the table space or part is compressed, the value is the compressed row length. If the table space or part is not compressed, the value is the uncompressed row length. The value is -1 if statistics have not been gathered.	G
SPACEF	FLOAT NOT NULL WITH DEFAULT	Kilobytes of DASD storage for the storage group. The value is -1 if statistics have not been gathered. This is an updatable column.	G
CREATORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
INSTANCE	SMALLINT NOT NULL WITH DEFAULT	INSTANCE indicates the data set instance number of the current base object (table and index).	G
CLONE	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether the table space contains any objects that are involved in a clone relationship: Y Table space contains objects that are involved in a clone relationship N Table space does not contain any objects that are involved in a clone relationship	G
MAXPARTITIONS	SMALLINT NOT NULL WITH DEFAULT	Identifies the maximum number of partitions to which the table space can grow. 0 if the table space is not a partition-by-growth table space.	G

SYSIBM.SYSTABLESPACESTATS table

The SYSIBM.SYSTABLESPACESTATS table contains real time statistics for table spaces.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
UPDATESTATTIME	TIMESTAMP NOT NULL WITH DEFAULT	The timestamp that the row in the TABLESPACESTATS table is inserted or updated.	G
NACTIVE	INTEGER	The number of active pages in the table space or partition.	G
NPAGES	INTEGER	The number of distinct pages with active rows in the partition or table space. This is an updatable column.	G
EXTENTS	SMALLINT	The number of extents in the table space. For multi-piece table spaces, this value is the number of extents for the last data set. For a data set that is stripped across multiple volumes, the value is the number of logical extents. A null value indicates the number of extents is unknown.	G
LOADRLASTTIME	TIMESTAMP	The timestamp that the LOAD REPLACE utility was last run on the table space or partition. A null value indicates that the LOAD REPLACE utility has never been run on the table space or partition or that the timestamp is unknown.	G
REORGLASTTIME	TIMESTAMP	The timestamp the REORG utility was last run on the table space or partition, or when the REORG utility has not been run, the time when the table space or partition was created. A null value indicates that the timestamp is unknown.	G
REORGINSERTS	INTEGER	The number of records or LOBs that have been inserted into the table space or partition or loaded into the table space or partition using the LOAD utility specified without the REPLACE option since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of inserted records or LOBs is unknown.	G
REORGDELETES	INTEGER	The number of records or LOBs that have been deleted from the table space or partition since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of deleted records or LOBs is unknown.	G
REORGUPDATES	INTEGER	The number of rows that have been updated in the table space or partition since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of updated rows is unknown.	G

Column name	Data type	Description	Use
REORGUNCLUSTINS	INTEGER	The number of records that were inserted that are not well-clustered with respect to the clustering index since the last REORG or LOAD REPLACE , or since the object was created. A record is well-clustered if the record is inserted into a page that is within 16 pages of the ideal candidate page. The clustering index determines the ideal candidate page. A null value indicates that the number of not well clustered pages is unknown.	G
REORGDISORGLOB	INTEGER	The number of LOBs that were inserted that are not perfectly chunked since the last REORG or LOAD REPLACE, or since the object was created. A LOB is perfectly chunked if the allocated pages are in the minimum number of chunks. A null value indicates that the number of not perfectly chunked LOBs is unknown.	G
REORGMASDELETE	INTEGER	The number of mass deletes from a segmented or LOB table space, or the number of dropped tables from a segmented table space since the last time the REORG or LOAD REPLACE utilities were run, or since the object was created. A null value indicates that the number of mass deletes is unknown.	G
REORGNEARINDREF	INTEGER	The number of overflow records that are created and relocated near the pointer record since the last time the REORG and LOAD REPLACE utilities were run, or since the object was created. For non-segmented table spaces, a page is near the present page if the two page numbers differ by 16 or less. For segmented table spaces, a page is near the present page if the two page numbers differ by SEGSIZE*2 or less. A null value indicates that the number of overflow records that are near the pointer record is unknown.	G
REORGFARINDREF	INTEGER	The number of overflow records that are created and relocated far from the pointer record since the last time the REORG and LOAD REPLACE utilities were run, or since the object was created. For non-segmented table spaces, a page is far from the present page if the two page numbers differ by more than 16. For segmented table spaces, a page is far from the present page if the two page numbers differ by at least (SEGSIZE*2)+1. A null value indicates that the number of overflow records that are near the pointer record is unknown.	G
STATSLASTTIME	TIMESTAMP	The timestamp of the last time that the RUNSTATS utility is run on the table space or partition.	G

Column name	Data type	Description	Use
STATSINSERTS	INTEGER	The number of records or LOBs that have been inserted into the table space or partition or loaded into the table space or partition using the LOAD utility specified without the REPLACE option since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of inserted records or LOBs is unknown.	G
STATSDELETES	INTEGER	The number of records or LOBs that have been deleted from the table space or partition since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of deleted records or LOBs is unknown.	G
STATSUPDATES	INTEGER	The number of rows that have been updated in the table space or partition since the last time that the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of updated rows is unknown.	G
STATSMASSDELETE	INTEGER	The number of mass deletes from a segmented or LOB table space, or the number of tables that are dropped from a segmented table space, since the last time the RUNSTATS utility was run, or since the object was created. A null value indicates that the number of mass deletes is unknown.	G
COPYLASTTIME	TIMESTAMP	The timestamp of the last full or incremental image copy of the table space or partition, or since the object was created. A null value indicates that the COPY utility has never been run on the table space or partition. A null value can also indicate that the timestamp of the last image copy is unknown.	G
COPYUPDATED-PAGES	INTEGER	The number of distinct types that have been updated since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of updated pages is unknown.	G
COPYCHANGES	INTEGER	The number of insert, update, and delete operations, or the number of records loaded, since the last time that the COPY utility was run, or since the object was created. A null value indicates that the number of insert, update, and delete operations or the number of records loaded is unknown.	G
COPYUPDATELRSN	CHAR(6) FOR BIT DATA	The LRSN or RBA of the first update that occurs after the last time the COPY utility was run, or since the object was created. A null value indicates that the LRSN or RBA is unknown.	G

Column name	Data type	Description	Use
COPYUPDATETIME	TIMESTAMP	The timestamp of the first update that occurs after the last time that the COPY utility was run, or since the object was created. A null value indicates that the timestamp is unknown.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DBID	SMALLINT NOT NULL	The internal identifier of the database. This column is used to map a DBID to its statistics.	G
PSID	SMALLINT NOT NULL	The internal identifier of the table space page set descriptor. This column is used to map a PSID to its statistics.	G
PARTITION	SMALLINT NOT NULL	The data set number within the table space. This column is used to map a data set number in a table space to its statistics. For partitioned table spaces, this value corresponds to the partition number for a single partition. For non-partitioned table spaces, this value is 0.	G
INSTANCE	SMALLINT NOT NULL WITH DEFAULT 1	Indicates if the object is associated with data set instance 1 or 2. This is an updatable column.	G
SPACE	BIGINT	The amount of space, in KB, that is allocated to the table space or partition. For multi-piece, linear page sets, this value is the amount of space in all data sets. A null value indicates the amount of space is unknown.	G
TOTALROWS	BIGINT	The number of rows or LOBs that are in the table space or partition.	G
DATASIZE	BIGINT	The total number of bytes that row data occupy in the data rows or LOB rows. This is an updatable column.	G
UNCOMPRESSED-DATASIZE	BIGINT	This column is not used. The value is always set to 0.	G
DBNAME	VARCHAR(24) NOT NULL	The name of the database. This column is used to map a database to its statistics.	G
NAME	VARCHAR(24) NOT NULL	The name of the table space. This column is used to map a table space to its statistics.	G

SYSIBM.SYSTABLES_HIST table

The SYSIBM.SYSTABLES_HIST table contains rows from SYSTABLES.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the table, view, or alias.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the table, view, or alias.	G
DBNAME	VARCHAR(24) NOT NULL	For a table, or a view of tables, the name of the database that contains the table space named in TSNAME. For a temporary table, an alias, or a view of a view, the value is DSNDB06.	G
TSNAME	VARCHAR(24) NOT NULL	For a table, or a view of one table, the name of the table space that contains the table. For a view of more than one table, the name of a table space that contains one of the tables. For a temporary table, the value is SYSPKAGE. For a view of a view, the value is SYSVIEWS. For an alias, it is SYSDBAUT.	G
COLCOUNT	SMALLINT NOT NULL	Number of columns in the table or view. The value is 0 if the row describes an alias.	G
PCTPAGES	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of active table space pages that contain rows of the table. A page is termed active if it is formatted for rows, regardless of whether it contains any. If the table space is segmented, the percentage is based on the number of active pages in the set of segments assigned to the table. The value is -1 if statistics have not been gathered, or the row describes a view, alias, temporary table, or auxiliary table.	S
PCTROWCOMP	SMALLINT NOT NULL WITH DEFAULT -1	Percentage of rows compressed within the total number of active rows in the table. This includes any row in a table space that is defined with COMPRESS YES. The value is -1 if statistics have not been gathered, or the row describes a view, alias, temporary table, or auxiliary table.	G
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. For a temporary table, the value of STATSTIME is always the default value.	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of rows in the table or total number of LOBs in an auxiliary table. The value is -1 if statistics have not been gathered or the row describes a view, alias, or temporary table.	S
NPAGESF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of pages on which rows of the partition appear. The value is -1 if statistics have not been gathered.	S

Column name	Data type	Description	Use
AVGROWLEN	INTEGER NOT NULL WITH DEFAULT -1	Average row length of the table specified in the table space. The value is -1 if statistics have not been gathered.	G
SPACEF	FLOAT(8) NOT NULL WITH DEFAULT -1	Kilobytes of DASD storage. The value is -1 if statistics have not been gathered. This is an updatable column.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSTABSTATS table

The SYSIBM.SYSTABSTATS table contains one row for each partition of a partitioned table space.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
CARD	INTEGER NOT NULL	Total number of rows in the partition.	S
NPAGES	INTEGER NOT NULL	Total number of pages on which rows of the partition appear.	S
PCTPAGES	SMALLINT NOT NULL	Percentage of total active pages in the partition that contain rows of the table.	S
NACTIVE	INTEGER NOT NULL	Number of active pages in the partition.	S
PCTROWCOMP	SMALLINT NOT NULL	Percentage of rows compressed within the total number of active rows in the partition. This includes any row in a table space that is defined with COMPRESS YES.	S
STATTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
DBNAME	VARCHAR(24) NOT NULL	Database that contains the table space named in TSNAME.	G
TSNAME	VARCHAR(24) NOT NULL	Table space that contains the table.	G
PARTITION	SMALLINT NOT NULL	Partition number of the table space that contains the table.	G
OWNER	VARCHAR(128) NOT NULL	The schema of the table.	G
NAME	VARCHAR(128) NOT NULL	Name of the table.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Total number of rows in the partition.	S

SYSIBM.SYSTABSTATS_HIST table

The SYSIBM.SYSTABSTATS_HIST table contains rows from SYSTABSTATS.

Rows are added or changed in this table when RUNSTATS collects history statistics. Rows in this table can also be inserted, updated, and deleted.

Column name	Data type	Description	Use
NPAGES	INTEGER NOT NULL	Total number of pages on which rows of the partition appear.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
DBNAME	VARCHAR(24) NOT NULL	Database that contains the table space named in TSNAME.	G
TSNAME	VARCHAR(24) NOT NULL	Table space that contains the table.	G
PARTITION	SMALLINT NOT NULL	Partition number of the table space that contains the table.	G
I OWNER	VARCHAR(128) NOT NULL	The schema of the table.	G
NAME	VARCHAR(128) NOT NULL	Name of the table.	G
CARDF	FLOAT(8) NOT NULL WITH DEFAULT -1	Total number of rows in the partition. The value is -1 if statistics have not been gathered.	S
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.SYSTRIGGERS table

The SYSIBM.SYSTRIGGERS table contains one row for each trigger.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the trigger and trigger package.	G
SCHEMA	VARCHAR(128) NOT NULL	Schema of the trigger. This implicit or explicit qualifier for the trigger name is also used for the collection ID of the trigger package.	G
SEQNO	SMALLINT NOT NULL	Sequence number of this row; the first portion of the trigger definition is in row 1, and successive rows have increasing SEQNO values.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database for the trigger.	G
OBID	SMALLINT NOT NULL	Internal identifier of the trigger.	G
OWNER	VARCHAR(128) NOT NULL	Owner of the trigger.	G
CREATEDBY	VARCHAR(128) NOT NULL	Primary authorization ID of the user who created the trigger.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the table or view.	G
TBOWNER	VARCHAR(128) NOT NULL	Qualifier of the name of the table or view to which this trigger applies.	G
TRIGTIME	CHAR(1) NOT NULL	Time when triggered actions are applied to the base table, relative to the event that activated the trigger: A Trigger is applied after the event. B Trigger is applied before the event. I Trigger is applied instead of the event	G
TRIGEVENT	CHAR(1) NOT NULL	Operation that activates the trigger: I Insert D Delete U Update	G
GRANULARITY	CHAR(1) NOT NULL	Trigger is executed once per: S Statement R Row	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the CREATE statement was executed for this trigger. The time value is used in resolving functions, distinct types, and stored procedures. It is also used to order the execution of multiple triggers.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

Column name	Data type	Description	Use
TEXT	VARCHAR(6000) NOT NULL	Full text of the CREATE TRIGGER statement.	G
REMARKS	VARCHAR(762) NOT NULL	A character string provided by the user with the COMMENT statement.	G
TRIGNAME	VARCHAR(128) NOT NULL	Unused	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of creator: blank Authorization ID L Role	G
ENVID	INTEGER NOT NULL WITH DEFAULT	Internal environment identifier.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSUSERAUTH table

The SYSIBM.SYSUSERAUTH table records the system privileges that are held by users.

Column name	Data type	Description	Use
GRANTOR	VARCHAR(128) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	VARCHAR(128) NOT NULL	Authorization ID of the user that holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G
	CHAR(12) NOT NULL	Internal use only	I
	CHAR(6) NOT NULL	Not used	N
	CHAR(8) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
AUTHHOWGOT	CHAR(1) NOT NULL	Authorization level of the user from whom the privileges were received. This authorization level is not necessarily the highest authorization level of the grantor. blank Not applicable C DBCTRL D DBADM L SYSCTRL M DBMAINT O SYSOPR S SYSADM	G
	CHAR(1) NOT NULL	Not used	N
BINDADDAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the BIND subcommand with the ADD option: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
BSDSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the RECOVER BSDS command: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
CREATEDBAAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create databases and automatically receive DBADM authority over the new databases: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
CREATEDBCAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can execute the CREATE DATABASE statement to create new databases and automatically receive DBCTRL authority over the new databases: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
CREATESGAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can execute the CREATE STOGROUP statement to create new storage groups: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
DISPLAYAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the DISPLAY commands: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
RECOVERAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the RECOVER INDOUBT command: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
STOPALLAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the STOP command: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
STOSPACEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the STOSPACE utility: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
SYSADMAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has system administration authority: blank Privilege is not held G Privilege was granted with the GRANT option Y Privilege was granted without the GRANT option GRANTEE has the privilege with the GRANT option for a value of either Y or G.	G
SYSOPRAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has system operator authority: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
TRACEAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the START TRACE and STOP TRACE commands: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
MON1AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can obtain IFC serviceability data: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G

Column name	Data type	Description	Use
MON2AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can obtain IFC data: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
CREATEALIASAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can execute the CREATE ALIAS statement: blank Privilege is not held G Privilege held with the GRANT option Y Privilege held without the GRANT option	G
SYSCTRLAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has SYSCTRL authority: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option GRANTEE has the privilege with the GRANT option for a value of either Y or G.	G
BINDAGENTAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has BINDAGENT privilege: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
ARCHIVEAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE is privileged to use the ARCHIVE LOG command: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
	CHAR(1) NOT NULL WITH DEFAULT	Not used	N
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed. The value is '1985-04-01.00.00.00.000000' for the one installation row.	G
CREATETMTAB-AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has CREATETMTABAUTH privilege: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
GRANTEETYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantee: blank Authorization ID L Role	G
GRANTORTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of grantor: blank Authorization ID L Role	G
DEBUGSESSION-AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE has DEBUGSESSION privilege: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G

SYSIBM.SYSVIEWDEP table

The SYSIBM.SYSVIEWDEP table records the dependencies of views on tables, functions, and other views.

Column name	Data type	Description	Use
BNAME	VARCHAR(128) NOT NULL	Name of the object on which the view is dependent. If the object type is a function (BTYP= 'F'), the name is the specific name of the function.	G
BCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of BNAME. For functions, it is the schema name of the BNAME.	G
BTYP	CHAR(1) NOT NULL	Type of object: F Function M Materialized query table T Table V View	G
DNAME	VARCHAR(128) NOT NULL	Name of the view.	G
DCREATOR	VARCHAR(128) NOT NULL	The schema of the view.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
BSHEMA	VARCHAR(128) NOT NULL WITH DEFAULT	Schema of BNAME.	G
DTYP	CHAR(1) NOT NULL	Type of table: F SQL function M Materialized query table V View	G
OWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the view, blank for views that were created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G

SYSIBM.SYSVIEWS table

The SYSIBM.SYSVIEWS table contains one or more rows for each view, materialized query table, or user-defined SQL function.

Column name	Data type	Description	Use
NAME	VARCHAR(128) NOT NULL	Name of the object.	G
CREATOR	VARCHAR(128) NOT NULL	The schema of the object.	G
SEQNO	SMALLINT NOT NULL	Sequence number of this row; the first portion of the view is on row one and successive rows have increasing values of SEQNO.	G
CHECK	CHAR(1) NOT NULL	Whether the WITH CHECK OPTION clause was specified in the CREATE VIEW statement: N No C Yes with the <i>cascaded</i> semantic Y Yes with the <i>local</i> semantic The value is N if the view has no WHERE clause, or the object is not a view.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
TEXT	VARCHAR(1500) NOT NULL	Text or portion of the text of the statement that was used to create the object.	G
PATHSCHEMAS	VARCHAR(2048) NOT NULL WITH DEFAULT	SQL path at the time the object was defined. The path is used to resolve unqualified data type and function names used in the object definition.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G
TYPE	CHAR(1) NOT NULL	Type of table: F SQL function M Materialized query table V View	G
REFRESH	CHAR(1) NOT NULL WITH DEFAULT	Refresh mode: D A materialized query table with a deferred refresh mode blank Not a materialized query table	G
ENABLE	CHAR(1) NOT NULL WITH DEFAULT	Indicates whether query optimization is enabled: Y Enabled N Disabled blank Not a materialized query table	G
MAINTENANCE	CHAR(1) NOT NULL WITH DEFAULT	Maintenance mode: S For a REFRESH = 'D', a materialized query table that is maintained by the system. U For a REFRESH = 'D', a materialized query table that is maintained by the user. blank Not a materialized query table.	G

Column name	Data type	Description	Use
REFRESH_TIME	TIMESTAMP NOT NULL WITH DEFAULT	For REFRESH = 'D' and MAINTENANCE = 'S', the timestamp of the REFRESH TABLE statement that last refreshed the data. Otherwise, this is the default timestamp ('0001-01-01.00.00.00.000000').	G
ISOLATION	CHAR(1) NOT NULL WITH DEFAULT	Isolation level when the materialized query table is created or altered from a base table: R RR (repeatable read) S CS (cursor stability) T RS (read stability) U UR (uncommitted read) blank Not a materialized query table	G
SIGNATURE	VARCHAR(1024) NOT NULL WITH DEFAULT FOR BIT DATA	Contains an internal description. Used for materialized query tables.	G
APP_ENCODING_ CCSID	INTEGER NOT NULL WITH DEFAULT	CCSID of the current application encoding scheme at the time the object was created. For objects created prior to Version 8 of DB2, the value is 0.	G
OWNER	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID of the owner of the view, blank for views that were created in a DB2 release prior to Version 9.	G
OWNERTYPE	CHAR(1) NOT NULL WITH DEFAULT	Indicates the type of owner: blank Authorization ID L Role	G
ENVID	INTEGER NOT NULL WITH DEFAULT	Internal environment identifier.	G

SYSIBM.SYSVOLUMES table

The SYSIBM.SYSVOLUMES table contains one row for each volume of each storage group.

Column name	Data type	Description	Use
SGNAME	VARCHAR(128) NOT NULL	Name of the storage group.	G
SGCREATOR	VARCHAR(128) NOT NULL	Authorization ID of the owner of the storage group.	G
VOLID	VARCHAR(18) NOT NULL	Serial number of the volume or * if SMS-managed.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. Blank if created prior to Version 9. See Release dependency indicators for all other values.	G

SYSIBM.SYSXMLRELS table

The SYSIBM.SYSXMLRELS table contains one row for each XML table that is created for an XML column.

Column name	Data type	Description	Use
TBOWNER	VARCHAR(128) NOT NULL	Schema or qualifier of the base table.	G
TBNAME	VARCHAR(128) NOT NULL	Name of the base table.	G
COLNAME	VARCHAR(128) NOT NULL	Name of the XML column in the base table.	G
XMLTBOWNER	VARCHAR(128) NOT NULL	Schema or qualifier of the XML table.	G
XMLTBNAME	VARCHAR(128) NOT NULL	Name of the XML table.	G
XMLRELOBID	INTEGER NOT NULL	Internal identifier of the relationship between the base table and the XML table.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G
CREATEDTS	TIMESTAMP NOT NULL	Time when the XML table was created.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.SYSXMLSTRINGS table

Each row of the SYSIBM.SYSXMLSTRINGS table contains a single string and its unique ID that are used to condense XML data. The string can be an element name, attribute name, name space prefix, or a namespace URI.

Column name	Data type	Description	Use
STRINGID	INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY	Unique ID for the string.	G
STRING	VARCHAR(1000) NOT NULL	The string data.	G
IBMREQD	CHAR(1) NOT NULL	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.USERNAMES table

Each row in the SYSIBM.USERNAMES table is used to carry out one outbound ID translation or inbound ID translation and “come from” checking.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
TYPE	CHAR(1) NOT NULL	How the row is to be used: I For inbound translation and “come from” checking. O For outbound translation. S For outbound system AUTHID to establish a trusted connection.	G
AUTHID	VARCHAR(128) NOT NULL WITH DEFAULT	Authorization ID to be translated. Applies to any authorization ID if blank.	G
LINKNAME	VARCHAR(24) NOT NULL	Identifies the VTAM or TCP/IP network locations associated with this row. A blank value in this column indicates this name translation rule applies to any TCP/IP or SNA partner. If a non-blank LINKNAME is specified, one or both of the following statements must be true: <ul style="list-style-type: none"> A row exists in SYSIBM.LUNAMES whose LUNAME matches the value specified in the SYSIBM.USERNAMES LINKNAME column. This row specifies the VTAM site associated with this name translation rule. A row exists in SYSIBM.IPNAMES whose LINKNAME matches the value specified in the SYSIBM.USERNAMES LINKNAME column. This row specifies the TCP/IP host associated with this name translation rule. Inbound name translation and “come from” checking are not performed for TCP/IP clients.	G
NEWAUTHID	VARCHAR(128) NOT NULL WITH DEFAULT	Translated value of AUTHID. Blank specifies no translation. NEWAUTHID can be stored as encrypted data by calling the DSNLEUSR stored procedure. To send the encrypted value of AUTHID across a network, one of the encryption security options in the SYSIBM.IPNAMES table should be specified.	G
PASSWORD	VARCHAR(24) NOT NULL WITH DEFAULT	Password to accompany an outbound request, if passwords are not encrypted by RACF. If passwords are encrypted, or the row is for inbound requests, the column is not used. PASSWORD can be stored as encrypted data by calling the DSNLEUSR stored procedure. To send the encrypted value of PASSWORD across a network, one of the encryption security options in the SYSIBM.IPNAMES table should be specified.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	G

SYSIBM.XSRCOMPONENT table

The SYSIBM.XSRCOMPONENT table is an auxiliary table for the BLOB column COMPONENT in SYSIBM.SYSXSROBJECTCOMPONENTS. It is in LOB table space SYSXSRA3.

Column name	Data type	Description	Use
COMPONENT	BLOB(30M)	Contents of the XML schema document	G

SYSIBM.XSROBJECTS table

The SYSIBM.XSROBJECTS table contains one row for each registered XML schema.

Rows in this table can only be changed using static SQL statements issued by the DB2-supplied XSR stored procedures.

Column name	Data type	Description	Use
XSROBJECTID	INTEGER NOT NULL	Internal identifier of the XML schema. XSROBJECTID is generated as an identity column.	G
XSROBJECTSCHEMA	VARCHAR(128) NOT NULL	Qualifier of the XML schema name. This is always set to 'SYSXSR'.	G
XSROBJECTNAME	VARCHAR(128) NOT NULL	Name of the XML schema.	G
TARGETNAMESPACE	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the target namespace URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS	G
SCHEMALOCATION	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the schema location URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ID that is used to support BLOB data type values.	G
GRAMMAR	BLOB(250M)	The internal binary representation of the XML schema.	G
PROPERTIES	BLOB(5M)	Additional property information of the entire XML schema.	G
CREATEDBY	VARCHAR(128) NOT NULL	Authorization ID under which the XML schema was created.	G
CREATEDTS	TIMESTAMP NOT NULL	The time that the DB2-supplied stored procedure XSR_REGISTER was executed for the XML schema.	G
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Registration status of the XML schema: C Complete I Incomplete T Temporary	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G
DECOMPOSITION	CHAR(1)	Indicates the decomposition status of the XSR object: Y Enabled N Not enabled X Inoperative	G

Column name	Data type	Description	Use
DECOMPOSITION_	VARCHAR(128)	Indicates the version of the XDB map that is used for	G
VERSION		decomposition.	
REMARKS	VARCHAR(762)	Character string that contains comments about this XML	G
		schema.	

SYSIBM.XSROBJECTCOMPONENTS table

The SYSIBM.XSROBJECTCOMPONENTS table contains one row for each component (document) in an XML schema.

Rows in this table can only be changed using static SQL statements issued by the DB2-supplied XSR stored procedures.

Column name	Data type	Description	Use
XSRCOMPONENTID	INTEGER NOT NULL	Internal identifier of the XML schema document. XSRCOMPONENTID is generated as an identity column.	G
TARGETNAMESPACE	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the target namespace URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
SCHEMALOCATION	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the schema location URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
ROWID	ROWID NOT NULL GENERATED ALWAYS	The ID that is used to support BLOB data type values.	G
COMPONENT	BLOB(30M) NOT NULL	Contents of the XML schema document.	G
PROPERTIES	BLOB(5M)	If available, additional property information of the XML schema document	G
CREATEDTS	TIMESTAMP NOT NULL	The time that the XML schema document was registered.	G
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Registration status of the XML schema: C Complete I Incomplete	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.XSROBJECTGRAMMAR table

SYSIBM.XSROBJECTGRAMMAR is an auxiliary table for the BLOB column GRAMMAR in SYSIBM.SYSXSROBJECTS. It is in LOB table space SYSXSRA1.

Column name	Data type	Description	Use
GRAMMAR	BLOB(250M)	Internal binary representation of the XML schema	G

SYSIBM.XSROBJECTHIERARCHIES table

The SYSIBM.XSROBJECTHIERARCHIES table contains one row for each component (document) in an XML schema to record the XML schema document hierarchy relationship.

Rows in this table can only be changed using static SQL statements issued by the DB2-supplied XSR stored procedures.

Column name	Data type	Description	Use
XSROBJECTID	INTEGER	Internal identifier of the XML schema.	G
XSRCOMPONENTID	INTEGER	Internal identifier of the XML schema document.	G
HTYPE	CHAR(1)	Hierarchy type: D Document P Primary document	G
TARGETNAMESPACE	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the target namespace URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
SCHEMALOCATION	INTEGER	The value of the STRINGID column in SYSIBM.SYSXMLSTRINGS when the schema location URI of the primary XML schema document is stored in SYSIBM.SYSXMLSTRINGS.	G
RELCREATED	CHAR(1) NOT NULL	The release of DB2 that is used to create the object. See Release dependency indicators for the values.	G

SYSIBM.XSROBJECTPROPERTY table

SYSIBM.XSROBJECTPROPERTY is an auxiliary table for the BLOB column PROPERTIES in SYSIBM.SYSXSROBJECTS. It is in LOB table space SYSXSRA2.

Column name	Data type	Description	Use
PROPERTIES	BLOB(5M)	Contents of the additional property information of the entire XML schema.	G

SYSIBM.XSRPROPERTY table

The SYSIBM.XSRPROPERTY table is an auxiliary table for the BLOB column COMPONENT in SYSIBM.SYSXSROBJECTCOMPONENTS. It is in LOB table space SYSXSRA3.

Column name	Data type	Description	Use
COMPONENT	BLOB(5M)	Contents of the additional property information of the XML schema document.	G

EXPLAIN tables

EXPLAIN tables contain information about SQL statements and functions that run on DB2 for z/OS.

You can create and maintain a set of EXPLAIN tables to capture and analyze information about the performance of SQL statements and functions that run on DB2 for z/OS. Each row in an EXPLAIN table describes some aspect of a step in the execution of a query or subquery in an explainable statement. The column values for the row identify, among other things, the query or subquery, the tables and other objects involved, the methods used to carry out each step, and cost information about those methods. DB2 creates EXPLAIN output and populates EXPLAIN tables in the following situations:

- When an EXPLAIN statement is executed.
- At BIND or REBIND with the EXPLAIN(YES) option. Rows are added for every explainable statement in the plan or package being bound. For a plan, these do not include statements in the packages that can be used with the plan. For either a package or plan, they do not include explainable statements within EXPLAIN statements nor do they include explainable statements that refer to declared temporary tables, which are incrementally bound at run time.
- When the DSNAXP stored procedure executes successfully.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

PLAN_TABLE

The plan table, PLAN_TABLE, contains information about access paths that is collected from the results of EXPLAIN statements.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Optional PLAN_TABLE formats

A PLAN_TABLE instance can have a format with fewer columns than those shown in the sample CREATE TABLE statement. However instances of PLAN_TABLE must have one of the following formats:

- All the columns up to and including REMARKS (COLCOUNT = 25)
- All the columns up to and including MIXOPSEQ (COLCOUNT = 28)

- All the columns up to and including COLLID (COLCOUNT = 30)
- All the columns up to and including JOIN_PGROUP_ID (COLCOUNT = 34)
- All the columns up to and including IBM_SERVICE_DATA (COLCOUNT = 43)
- All the columns up to and including BIND_TIME (COLCOUNT = 46)
- All the columns up to and including PRIMARY_ACCESTYPE (COLCOUNT = 49)
- All the columns up to and including TABLE_TYPE (COLCOUNT = 51)
- All the columns up to and including STMTTOKEN (COLCOUNT = 58)
- All the columns shown in the CREATE statement (COLCOUNT=59)

Whichever set of columns you choose, the columns must appear in the order in which the CREATE statement indicates. You can add columns to an existing plan table with the ALTER TABLE statement only if the modified table satisfies one of the allowed sets of columns. For example, you cannot add column PREFETCH by itself, but must add columns PREFETCH, COLUMN_FN_EVAL, and MIXOPSEQ. If you add any NOT NULL columns, give them the NOT NULL WITH DEFAULT attribute.

When using the 58-column format for the plan table, remember that many columns (PROGNAME, COLLID, CREATOR, TNAME, ACCESSCREATOR, ACCESSNAME, CORRELATION_NAME) must be defined as VARCHAR(128). In previous releases of DB2, these columns were smaller.

Missing columns are ignored when rows are added to a plan table.

Column descriptions

PSPI Your subsystem or data sharing group can contain more than one of these tables, including a table with the qualifier SYSIBM, a table with the qualifier DB2OSCA, and additional tables that are qualified by user IDs.

The following table shows the descriptions of the columns in PLAN_TABLE.

Table 153. Descriptions of columns in PLAN_TABLE

Column name	Data Type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
QBLOCKNO	SMALLINT NOT NULL	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
PLANNO	SMALLINT NOT NULL	The number of the step in which the query that is indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.
METHOD	SMALLINT NOT NULL	<p>A number that indicates the join method that is used for the step:</p> <p>0 The table in this step is the first table that is accessed, a continuation of a previous table that was accessed, or a table that is not used.</p> <p>1 A nested loop join is used. For each row of the current composite table, matching rows of a new table are found and joined.</p> <p>2 A merge scan join is used. The current composite table and the new table are scanned in the order of the join columns, and matching rows are joined.</p> <p>3 Sorts are needed by ORDER BY, GROUP BY, SELECT DISTINCT, UNION, INTERSECT, EXCEPT, a quantified predicate, or an IN predicate. This step does not access a new table.</p> <p>4 A hybrid join was used. The current composite table is scanned in the order of the join-column rows of the new table. The new table is accessed using list prefetch.</p>
CREATOR	VARCHAR(128) NOT NULL	The creator of the new table that is accessed in this step, blank if METHOD is 3.

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
TNAME	VARCHAR(128) NOT NULL	<p>The name of one of the following objects:</p> <ul style="list-style-type: none"> • Materialized query table • Created or declared temporary table • Accelerated query table • Materialized view • materialized table expression <p>The value is blank if METHOD is 3. The column can also contain the name of a table in the form DSNWFQB(<i>qblockno</i>). DSNWFQB(<i>qblockno</i>) is used to represent the intermediate result of a UNION ALL, INTERSECT ALL, EXCEPT ALL, or an outer join that is materialized. If a view is merged, the name of the view does not appear. DSN_DIM_TBLX(<i>qblockno</i>) is used to represent the work file of a star join dimension table.</p>
TABNO	SMALLINT NOT NULL	Values are for IBM use only.
ACCESSTYPE	CHAR(2) NOT NULL	<p>The method of accessing the new table:</p> <p>A Accelerated query table access.</p> <p>DI By an intersection of multiple DOCID lists to return the final DOCID list</p> <p>DU By a union of multiple DOCID lists to return the final DOCID list</p> <p>DX By an XML index scan on the index that is named in ACCESSNAME to return a DOCID list</p> <p>E By direct row access using a row change timestamp column.</p> <p>I By an index (identified in ACCESSCREATOR and ACCESSNAME)</p> <p>I1 By a one-fetch index scan</p> <p>M By a multiple index scan (followed by MX, MI, or MU)</p> <p>MI By an intersection of multiple indexes</p> <p>MU By a union of multiple indexes</p> <p>MX By an index scan on the index named in ACCESSNAME. When the access method MX follows the access method DX, DI, or DU, the table is accessed by the DOCID index by using the DOCID list that is returned by DX, DI, or DU.</p> <p>N</p> <ul style="list-style-type: none"> • By an index scan when the matching predicate contains the IN keyword <p>O By a work file scan, as a result of a subquery.</p> <p>P By a dynamic pair-wise index scan</p> <p>R By a table space scan</p> <p>RW By a work file scan of the result of a materialized user-defined table function</p> <p>V By buffers for an INSERT statement within a SELECT</p> <p>blank Not applicable to the current row</p>
MATCHCOLS	SMALLINT NOT NULL	For ACCESSTYPE I, I1, N, MX, or DX, the number of index keys that are used in an index scan; otherwise, 0.

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
ACCESSCREATOR	VARCHAR(128) NOT NULL	For ACCESTYPE I, I1, N, MX, or DX, the creator of the index; otherwise, blank.
ACCESSNAME	VARCHAR(128) NOT NULL	For ACCESTYPE I, I1, N, MX, or DX, the name of the index; for ACCESTYPE P, DSNPJW(<i>mioxseqno</i>) is the starting pair-wise join leg in MIXOPSEQ; otherwise, blank.
INDEXONLY	CHAR(1) NOT NULL	Indication of whether access to an index alone is enough to perform the step, or Indication of whether data too must be accessed. Y Yes N No
SORTN_UNIQ	CHAR(1) NOT NULL	Indication of whether the new table is sorted to remove duplicate rows. Y Yes N No
SORTN_JOIN	CHAR(1) NOT NULL	Indication of whether the new table is sorted for join method 2 or 4. Y Yes N No
SORTN_ORDERBY	CHAR(1) NOT NULL	Indication of whether the new table is sorted for ORDER BY. Y Yes N No
SORTN_GROUPBY	CHAR(1) NOT NULL	Indication of whether the new table is sorted for GROUP BY. Y Yes N No
SORTC_UNIQ	CHAR(1) NOT NULL	Indication of whether the composite table is sorted to remove duplicate rows. Y Yes N No
SORTC_JOIN	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for join method 1, 2 or 4. Y Yes N No
SORTC_ORDERBY	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for an ORDER BY clause or a quantified predicate. Y Yes N No

Table 153. Descriptions of columns in *PLAN_TABLE* (continued)

Column name	Data Type	Description
SORTC_GROUPBY	CHAR(1) NOT NULL	Indication of whether the composite table is sorted for a GROUP BY clause.
		Y Yes
		N No
TSLOCKMOD	CHAR(3) NOT NULL	An indication of the mode of lock that is acquired on either the new table, or its table space or table space partitions. If the isolation can be determined at bind time, the values are:
		IS Intent share lock
		IX Intent exclusive lock
		S Share lock
		U Update lock
		X Exclusive lock
		SIX Share with intent exclusive lock
		N UR isolation; no lock
		If the isolation level cannot be determined at bind time, the lock mode is determined by the isolation level at run time is shown by the following values.
		NS For UR isolation, no lock; for CS, RS, or RR, an S lock.
		NIS For UR isolation, no lock; for CS, RS, or RR, an IS lock.
		NSS For UR isolation, no lock; for CS or RS, an IS lock; for RR, an S lock.
		SS For UR, CS, or RS isolation, an IS lock; for RR, an S lock.
		The data in this column is right justified. For example, IX appears as a blank, followed by I, followed by X. If the column contains a blank, then no lock is acquired.
		If the access method in the ACCESTYPE column is DX, DI, or DU, no latches are acquired on the XML index page and no lock is acquired on the new base table data page or row, nor on the XML table and the corresponding table spaces. The value of TSLOCKMODE is a blank in this case.
TIMESTAMP	CHAR(16) NOT NULL	Usually, the time at which the row is processed, to the last .01 second. If necessary, DB2 adds .01 second to the value to ensure that rows for two successive queries have different values.
REMARKS	VARCHAR(762) NOT NULL	A field into which you can insert any character string of 762 or fewer characters.
PREFETCH	CHAR(1) NOT NULL	Indication of whether data pages are to be read in advance by prefetch:
		D Optimizer expects dynamic prefetch
		S Pure sequential prefetch
		L Prefetch through a page list
		blank Unknown or no prefetch

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
COLUMN_FN_EVAL	CHAR(1) NOT NULL	When an SQL aggregate function is evaluated: R While the data is being read from the table or index S While performing a sort to satisfy a GROUP BY clause blank After data retrieval and after any sorts
MIXOPSEQ	SMALLINT NOT NULL	The sequence number of a step in a multiple index operation. 1, 2, ... n For the steps of the multiple index procedure (ACCESSTYPE is MX, MI, MU, DX, DI, or DU). 0 For any other rows.
VERSION	VARCHAR(122) NOT NULL	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable.
COLLID	VARCHAR(128) NOT NULL	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSNODYNAMICSQLCACHE indicates that the row is for a cached statement.
ACCESS_DEGREE	SMALLINT	The number of parallel tasks or operations that are activated by a query. This value is determined at bind time; the actual number of parallel operations that are used at execution time could be different. This column contains 0 if a host variable is used. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
ACCESS_PGROUP_ID	SMALLINT	The identifier of the parallel group for accessing the new table. A parallel group is a set of consecutive operations, executed in parallel, that have the same number of parallel tasks. This value is determined at bind time; it could change at execution time. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
JOIN_DEGREE	SMALLINT	The number of parallel operations or tasks that are used in joining the composite table with the new table. This value is determined at bind time and can be 0 if a host variable is used. The actual number of parallel operations or tasks used at execution time could be different. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
JOIN_PGROUP_ID	SMALLINT	The identifier of the parallel group for joining the composite table with the new table. This value is determined at bind time; it could change at execution time. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
SORTC_PGROUP_ID	SMALLINT	The parallel group identifier for the parallel sort of the composite table. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
SORTN_PGROUP_ID	SMALLINT	The parallel group identifier for the parallel sort of the new table. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
PARALLELISM_ MODE	CHAR(1)	The kind of parallelism, if any, that is used at bind time: C Query CP parallelism I Query I/O parallelism X Sysplex query parallelism This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
MERGE_ JOIN_ COLS	SMALLINT	The number of columns that are joined during a merge scan join (Method=2). This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.
CORRELATION_ NAME 	VARCHAR(128)	The correlation name of a table or view that is specified in the statement. If no correlation name exists, then the column is null. This column contains the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, it can contain null if the method that it refers to does not apply.

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
PAGE_RANGE	CHAR(1)	Indication of whether the table qualifies for page range screening, so that plans scan only the partitions that are needed. Y Yes blank No
JOIN_TYPE	CHAR(1)	The type of join: F FULL OUTER JOIN L LEFT OUTER JOIN P Pair-wise join S Star join blank INNER JOIN or no join RIGHT OUTER JOIN converts to a LEFT OUTER JOIN when you use it, so that JOIN_TYPE contains L.
GROUP_MEMBER	VARCHAR(128)	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
IBM_SERVICE_DATA	VARCHAR(254) FOR BIT DATA	This column contains values that are for IBM use only.
WHEN_OPTIMIZE	CHAR(1) NOT NULL WITH DEFAULT	When the access path was determined: blank At bind time, using a default filter factor for any host variables, parameter markers, or special registers. B At bind time, using a default filter factor for any host variables, parameter markers, or special registers; however, the statement is re-optimized at run time using input variable values for input host variables, parameter markers, or special registers. The bind option REOPT(ALWAYS), REOPT(AUTO), or REOPT(ONCE) must be specified for reoptimization to occur. R At run time, using input variables for any host variables, parameter markers, or special registers. The bind option REOPT(ALWAYS), REOPT(AUTO), or REOPT(ONCE) must be specified for this to occur.

Table 153. Descriptions of columns in *PLAN_TABLE* (continued)

Column name	Data Type	Description
QBLOCK_TYPE	CHAR(6) NOT NULL WITH DEFAULT	<p>For each query block, an indication of the type of SQL operation that is performed. For the outermost query, this column identifies the statement type. Possible values include:</p> <p>SELECT SELECT</p> <p>INSERT INSERT</p> <p>UPDATE UPDATE</p> <p>MERGE MERGE</p> <p>DELETE DELETE</p> <p>SELUPD SELECT with FOR UPDATE OF</p> <p>DELCUR DELETE WHERE CURRENT OF CURSOR</p> <p>UPDCUR UPDATE WHERE CURRENT OF CURSOR</p> <p>CORSUB Correlated subselect or fullselect</p> <p>TRUNCA TRUNCATE</p> <p>NCOSUB Noncorrelated subselect or fullselect</p> <p>TABLEX Table expression</p> <p>TRIGGR WHEN clause on CREATE TRIGGER</p> <p>UNION UNION</p> <p>UNIONA UNION ALL</p> <p>INTERS INTERSECT</p> <p>INTERA INTERSECT ALL</p> <p>EXCEPT EXCEPT</p> <p>EXCEPTA EXCEPT ALL</p>

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
BIND_TIME	TIMESTAMP NOT NULL WITH DEFAULT	<p>The time when the EXPLAIN information was captured:</p> <p>All cached statements When the statement entered the cache, in the form of a full-precision timestamp value.</p> <p>Non-cached static statements When the statement was bound, in the form of a full precision timestamp value.</p> <p>Non-cached dynamic statements When EXPLAIN was executed, in the form of a value equivalent to the value of the TIMESTAMP column appended by 4 zeros.</p>
OPTHINT	VARCHAR(128) NOT NULL WITH DEFAULT	A string that you use to identify this row as an optimization hint for DB2. DB2 uses this row as input when choosing an access path.
HINT_USED	VARCHAR(128) NOT NULL WITH DEFAULT	If DB2 used one of your optimization hints, it puts the identifier for that hint (the value in OPTHINT) in this column.
PRIMARY_ACCESTYPE	CHAR(1) NOT NULL WITH DEFAULT	<p>Indicates Indication of whether direct row access is attempted first:</p> <p>D DB2 tries to use direct row access with a rowid column. If DB2 cannot use direct row access with a rowid column at run time, it uses the access path that is described in the ACCESTYPE column of PLAN_TABLE.</p> <p>T The base table or result file is materialized into a work file, and the work file is accessed via sparse index access. If a base table is involved, then ACCESTYPE indicates how the base table is accessed.</p> <p>blank DB2 does not try to use direct row access by using a rowid column or sparse index access for a work file. The value of the ACCESTYPE column of PLAN_TABLE provides information on the method of accessing the table.</p>
PARENT_QBLOCKNO	SMALLINT NOT NULL	A number that indicates the QBLOCKNO of the parent query block.

Table 153. Descriptions of columns in PLAN_TABLE (continued)

Column name	Data Type	Description
TABLE_TYPE	CHAR(1)	<p>The type of new table:</p> <p>B Buffers for SELECT from INSERT, SELECT from UPDATE, SELECT from MERGE, or SELECT from DELETE statement.</p> <p>C Common table expression</p> <p>F Table function</p> <p>M Materialized query table</p> <p>Q Temporary intermediate result table (not materialized). For the name of a view or nested table expression, a value of Q indicates that the materialization was virtual and not actual. Materialization can be virtual when the view or nested table expression definition contains a UNION ALL that is not distributed.</p> <p>R Recursive common table expression</p> <p>S Subquery (correlated or non-correlated)</p> <p>T Table</p> <p>W Work file</p> <p>The value of the column is null if the query uses GROUP BY, ORDER BY, or DISTINCT, which requires an implicit sort.</p>
TABLE_ENCODE	CHAR(1) NOT NULL WITH DEFAULT	<p>The encoding scheme of the table. The possible values are:</p> <p>A ASCII</p> <p>E EBCDIC</p> <p>U Unicode</p> <p>M The table contains multiple CCSID sets</p>
TABLE_SCCSID	SMALLINT NOT NULL WITH DEFAULT	The SBCS CCSID value of the table. If column TABLE_ENCODE is M, the value is 0.
TABLE_MCCSID	SMALLINT NOT NULL WITH DEFAULT	The mixed CCSID value of the table. If the value of the TABLE_ENCODE column is M, the value is 0. If MIXED=NO in the DSNHDECP module, the value is -2.
TABLE_DCCSID	SMALLINT NOT NULL WITH DEFAULT	The DBCS CCSID value of the table. If the value of the TABLE_ENCODE column is M, the value is 0. If MIXED=NO in the DSNHDECP module, the value is -2.
ROUTINE_ID	INTEGER NOT NULL WITH DEFAULT	The values in this column are for IBM use only.
CTEREF	SMALLINT NOT NULL WITH DEFAULT	If the referenced table is a common table expression, the value is the top-level query block number.
STMTTOKEN	VARCHAR(240)	User-specified statement token.

Table 153. Descriptions of columns in *PLAN_TABLE* (continued)

Column name	Data Type	Description
PARENT_PLANNO	SMALLINT NOT NULL	Corresponds to the plan number in the parent query block where a correlated subquery is invoked. Or, for non-correlated subqueries, corresponds to the plan number in the parent query block that represents the work file for the subquery.



DSN_DETCOST_TABLE

The detailed cost table, DSN_DETCOST_TABLE, contains information about detailed cost estimation of the mini-plans in a query.

PSPI

When a query block is accelerated, a single row is added to DSN_DETCOST_TABLE. In rows for acceleration query blocks only the COMPCARD and COMPCOST columns contain meaningful information.

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_DETCOST_TABLE.

Table 154. DSN_DETCOST_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements</p> <p>The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements</p> <p>DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
PLANNO	SMALLINT NOT NULL	The plan number, a number used to identify each mini-plan with a query block.
OPENIO	FLOAT(4) NOT NULL	The Do-at-open IO cost for non-correlated subquery.

Table 154. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
OPENCPU	FLOAT(4) NOT NULL	The Do-at-open CPU cost for non-correlated subquery.
OPENCOST	FLOAT(4) NOT NULL	The Do-at-open total cost for non-correlated subquery.
DMIO	FLOAT(4) NOT NULL	IBM internal use only.
DMCPU	FLOAT(4) NOT NULL	IBM internal use only.
DMTOT	FLOAT(4) NOT NULL	IBM internal use only.
SUBQIO	FLOAT(4) NOT NULL	IBM internal use only.
SUBQCOST	FLOAT(4) NOT NULL	IBM internal use only.
BASEIO	FLOAT(4) NOT NULL	IBM internal use only.
BASECPU	FLOAT(4) NOT NULL	IBM internal use only.
BASETOT	FLOAT(4) NOT NULL	IBM internal use only.
ONECOMPROWS	FLOAT(4) NOT NULL	The number of rows qualified after applying local predicates.
IMLEAF	FLOAT(4) NOT NULL	The number of index leaf pages scanned by Data Manager.
IMIO	FLOAT(4) NOT NULL	IBM internal use only.
IMPREFH	CHAR(2) NOT NULL	IBM internal use only.
IMMPRED	INTEGER NOT NULL	IBM internal use only.
IMFF	FLOAT(4) NOT NULL	The filter factor of matching predicates only.
IMSRPRED	INTEGER NOT NULL	IBM internal use only.
IMFFADJ	FLOAT(4) NOT NULL	The filter factor of matching and screening predicates.
IMSCANCST	FLOAT(4) NOT NULL	IBM internal use only.
IMROWCST	FLOAT(4) NOT NULL	IBM internal use only.
IMPAGECST	FLOAT(4) NOT NULL	IBM internal use only.
IMRIDSORT	FLOAT(4) NOT NULL	IBM internal use only.
IMMERGCST	FLOAT(4) NOT NULL	IBM internal use only.
IMCPU	FLOAT(4) NOT NULL	IBM internal use only.
IMTOT	FLOAT(4) NOT NULL	IBM internal use only.
IMSEQNO	SMALLINT NOT NULL	IBM internal use only.
DMPEREFH	FLOAT(4) NOT NULL	IBM internal use only.
DMCLUDIO	FLOAT(4) NOT NULL	IBM internal use only.
DMPREDS	INTEGER NOT NULL	IBM internal use only.
DMSROWS	FLOAT(4) NOT NULL	IBM internal use only.
DMSCANCST	FLOAT(4) NOT NULL	IBM internal use only.
DMCOLS	FLOAT(4) NOT NULL	The number of data manager columns.
DMROWS	FLOAT(4) NOT NULL	The number of data manager rows returned (after all stage 1 predicates are applied).
RDSROWCST	FLOAT(4) NOT NULL	IBM internal use only.

Table 154. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
DMPAGECST	FLOAT(4) NOT NULL	IBM internal use only.
DMDATAIO	FLOAT(4) NOT NULL	IBM internal use only.
DMDATAIO	FLOAT(4) NOT NULL	IBM internal use only.
DMDATACPU	FLOAT(4) NOT NULL	IBM internal use only.
DMDATACPU	FLOAT(4) NOT NULL	IBM internal use only.
RDSROW	FLOAT(4) NOT NULL	The number of RDS rows returned (after all stage 1 and stage 2 predicates are applied).
SNCOLS	SMALLINT NOT NULL	The number of columns as sort input for new table.
SNROWS	FLOAT(4) NOT NULL	The number of rows as sort input for new table.
SNRECSZ	INTEGER NOT NULL	The record size for new table.
SNPAGES	FLOAT(4) NOT NULL	The page size for new table.
SNRUNS	FLOAT(4) NOT NULL	The number of runs generated for sort of new table.
SNMERGES	FLOAT(4) NOT NULL	The number of merges needed during sort.
SNIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
SNCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
SNCOST	FLOAT(4) NOT NULL	IBM internal use only.
SNSCANIO	FLOAT(4) NOT NULL	IBM internal use only.
SNSCANCPU	FLOAT(4) NOT NULL	IBM internal use only.
SNCCOLS	FLOAT(4) NOT NULL	The number of columns as sort input for Composite table.
SCROWS	FLOAT(4) NOT NULL	The number of rows as sort input for Composite Table.
SCRECSZ	FLOAT(4) NOT NULL	The record size for Composite table.
SCPAGES	FLOAT(4) NOT NULL	The page size for Composite table.
SCRUNS	FLOAT(4) NOT NULL	The number of runs generated during sort of composite.
SCMERGES	FLOAT(4) NOT NULL	The number of merges needed during sort of composite.
SCIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
SCCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
SCCOST	FLOAT(4) NOT NULL	IBM internal use only.
SCSCANIO	FLOAT(4) NOT NULL	IBM internal use only.
SCSCANCPU	FLOAT(4) NOT NULL	IBM internal use only.
SCSCANCOST	FLOAT(4) NOT NULL	IBM internal use only.
COMPCARD	FLOAT(4) NOT NULL	The total composite cardinality.
COMPIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
COMPCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
COMPCOST	FLOAT(4) NOT NULL	The total cost.

Table 154. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
JOINCOLS	SMALLINT NOT NULL	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The EXPLAIN timestamp.
COSTBLK	INTEGER NOT NULL	IBM internal use only.
COSTSTOR	INTEGER NOT NULL	IBM internal use only.
MPBLK	INTEGER NOT NULL	IBM internal use only.
MPSTOR	INTEGER NOT NULL	IBM internal use only.
COMPOSITES	INTEGER NOT NULL	IBM internal use only.
CLIPPED	INTEGER NOT NULL	IBM internal use only.
TABREF	VARCHAR(64) NOT NULL	IBM internal use only.
MAX_COMPOSITES	INTEGER NOT NULL	IBM internal use only.
MAX_STOR	INTEGER NOT NULL	IBM internal use only.
MAX_CPU	INTEGER NOT NULL	IBM internal use only.
MAX_ELAP	INTEGER NOT NULL	IBM internal use only.
TBL_JOINED_THRESH	INTEGER NOT NULL	IBM internal use only.
STOR_USED	INTEGER NOT NULL	IBM internal use only.
CPU_USED	INTEGER NOT NULL	IBM internal use only.
ELAPSED	INTEGER NOT NULL	IBM internal use only.
MIN_CARD_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_CARD_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_COST_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_COST_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_CARD_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_COST_KEEP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MIN_VALUE_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_VALUE_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_VALUE_CARD_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
MAX_VALUE_COST_CLIP	FLOAT(4) NOT NULL	IBM internal use only.
GROUP_MEMBER	VARCHAR(240)	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
PSEQIOCOST	FLOAT(4) NOT NULL	IBM internal use only.

Table 154. DSN_DETCOST_TABLE description (continued)

Column name	Data type	Description
PSEQIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
PSEQCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
PSEQCOST	FLOAT(4) NOT NULL	IBM internal use only.
PADJIOCOST	FLOAT(4) NOT NULL	IBM internal use only.
PADJCPUCOST	FLOAT(4) NOT NULL	IBM internal use only.
PADJYCOST	FLOAT(4) NOT NULL	IBM internal use only.



DSN_FILTER_TABLE

The filter table, DSN_FILTER_TABLE, contains information about how predicates are used during query processing.

 PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_FILTER_TABLE.

Table 155. DSN_FILTER_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
PLANNO	SMALLINT	The plan number, a number used to identify each miniplan with a query block.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
COLLID	INTEGER NOT NULL	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSN_DYNAMICSQLCACHE indicates that the row is for a cached statement.
ORDERNO	INTEGER NOT NULL	The sequence number of evaluation. Indicates the order in which the predicate is applied within each stage
PREDNO	INTEGER NOT NULL	The predicate number, a number used to identify a predicate within a query.
STAGE	CHAR(9) NOT NULL	<p>Indicates at which stage the Predicate is evaluated. The possible values are:</p> <ul style="list-style-type: none"> • 'Matching', • 'Screening', • 'Stage 1' • 'Stage 2'
ORDERCLASS	INTEGER NOT NULL	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The explain timestamp.

Table 155. DSN_FILTER_TABLE description (continued)

Column name	Data type	Description
MIXOPSEQNO	SMALLINT NOT NULL	IBM internal use only.
REEVAL	CHAR(1) NOT NULL	IBM internal use only.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.



DSN_FUNCTION_TABLE

The function table, DSN_FUNCTION_TABLE, contains descriptions of functions that are used in specified SQL statements.



Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions



The following table describes the columns of DSN_FUNCTION_TABLE.

Table 156. Descriptions of columns in DSN_FUNCTION_TABLE

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	INTEGER NOT NULL	A number that identifies each query block within a query.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row, or blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
COLLID	VARCHAR(128) NOT NULL	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSN DYNAMICSQLCACHE indicates that the row is for a cached statement.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN, or blank. For more information about the GROUP_MEMBER column, see
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time at which the statement is processed. This time is the same as the BIND_TIME column in PLAN_TABLE.
SCHEMA_NAME	VARCHAR(128) NOT NULL	The schema name of the function invoked in the explained statement.
FUNCTION_NAME	VARCHAR(128) NOT NULL	The name of the function invoked in the explained statement.
SPEC_FUNC_NAME	VARCHAR(128) NOT NULL	The specific name of the function invoked in the explained statement.
FUNCTION_TYPE	CHAR(2) NOT NULL	<p>The type of function invoked in the explained statement. Possible values are:</p> <p>CU Column function SU Scalar function TU Table function</p>
VIEW_CREATOR	VARCHAR(128) NOT NULL	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the creator of the view. Otherwise, blank.
VIEW_NAME	VARCHAR(128) NOT NULL	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the name of the view. Otherwise, blank.
PATH	VARCHAR(2048) NOT NULL	The value of the SQL path that was used to resolve the schema name of the function.

Table 156. Descriptions of columns in DSN_FUNCTION_TABLE (continued)

Column name	Data type	Description
FUNCTION_TEXT	VARCHAR(1500) NOT NULL	The text of the function reference (the function name and parameters). If the function reference is over 100 bytes, this column contains the first 100 bytes. For functions specified in infix notation, FUNCTION_TEXT contains only the function name. For example, for a function named /, which overloads the SQL divide operator, if the function reference is A/B, FUNCTION_TEXT contains only /.



DSN_PGRANGE_TABLE

The page range table, DSN_PGRANGE_TABLE, contains information about qualified partitions for all page range scans in a query.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_PGRANGE_TABLE.

Table 157. DSN_PGRANGE_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
TABNO	SMALLINT NOT NULL	The table number, a number which uniquely identifies the corresponding table reference within a query.
RANGE	SMALLINT NOT NULL	The sequence number of the current page range.
FIRSTPART	SMALLINT NOT NULL	The starting partition in the current page range.
LASTPART	SMALLINT NOT NULL	The ending partition in the current page range.
NUMPARTS	SMALLINT NOT NULL	The number of partitions in the current page range.
EXPLAIN_TIME	SMALLINT NOT NULL	The EXPLAIN timestamp.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.



DSN_PGROUP_TABLE

The parallel group table, DSN_PGROUP_TABLE, contains information about the parallel groups in a query.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_PGROUP_TABLE

Table 158. DSN_PGROUP_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
PLANNAME	VARCHAR(24) NOT NULL	The application plan name.
COLLID	VARCHAR(128) NOT NULL	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSNNDYNAMICSQLCACHE indicates that the row is for a cached statement.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The explain timestamp.
VERSION	VARCHAR(122) NOT NULL	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable.
GROUPID	SMALLINT NOT NULL	The parallel group identifier within the current query block.
FIRSTPLAN	SMALLINT NOT NULL	The plan number of the first contributing mini-plan associated within this parallel group.
LASTPLAN	SMALLINT NOT NULL	The plan number of the last mini-plan associated within this parallel group.
CPUCOST	REAL NOT NULL	The estimated total CPU cost of this parallel group in milliseconds.
IO COST	REAL NOT NULL	The estimated total I/O cost of this parallel group in milliseconds.
BESTTIME	REAL NOT NULL	The estimated elapsed time for each parallel task for this parallel group.

Table 158. DSN_PGROUPTABLE description (continued)

Column name	Data type	Description
DEGREE	SMALLINT NOT NULL	The degree of parallelism for this parallel group determined at bind time. Max parallelism degree if the Table space is large is 255, otherwise 64.
MODE	CHAR(1) NOT NULL	The parallel mode: ' I ' IO parallelism ' C ' CPU parallelism ' X ' multiple CPU Sysplex parallelism (highest level) ' N ' No parallelism
REASON	SMALLINT NOT NULL	The reason code for downgrading parallelism mode.
LOCALCPU	SMALLINT NOT NULL	The number of CPUs currently online when preparing the query.
TOTALCPU	SMALLINT NOT NULL	The total number of CPUs in Sysplex. LOCALCPU and TOTALCPU are different only for the DB2 coordinator in a Sysplex.
FIRSTBASE	SMALLINT	The table number of the table that partitioning is performed on.
LARGETS	CHAR(1)	'Y' if the TableSpace is large in this group.
PARTKIND	CHAR(1)	The partitioning type: ' L ' Logical partitioning ' P ' Physical partitioning
GROUPTYPE	CHAR(3)	Determines what operations this parallel group contains: table Access, Join, or Sort 'A' 'AJ' 'AJS'
ORDER	CHAR(1)	The ordering requirement of this parallel group : ' N ' No order. Results need no ordering. ' T ' Natural Order. Ordering is required but results already ordered if accessed via index. ' K ' Key Order. Ordering achieved by sort. Results ordered by sort key. This value applies only to parallel sort.
STYLE	CHAR(4)	The Input/Output format style of this parallel group. Blank for IO Parallelism. For other modes: ' RIRO ' Records IN, Records OUT ' WIRO ' Work file IN, Records OUT ' WIWO ' Work file IN, Work file OUT

Table 158. DSN_PGROUPTABLE description (continued)

Column name	Data type	Description
RANGEKIND	CHAR(1)	The range type: 'K' Key range 'P' Page range
NKEYCOLS	SMALLINT	The number of interesting key columns, that is, the number of columns that will participate in the key operation for this parallel group.
LOWBOUND	VARCHAR(40)	The low bound of parallel group.
HIGHBOUND	VARCHAR(40)	The high bound of parallel group.
LOWKEY	VARCHAR(40)	The low key of range if partitioned by key range.
HIGHKEY	VARCHAR(40)	The high key of range if partitioned by key range.
FIRSTPAGE	CHAR(4)	The first page in range if partitioned by page range.
LASTPAGE	CHAR(4)	The last page in range if partitioned by page range.
GROUP_MEMBER	CHAR(8) NOT NULL	IBM internal use only.
HOST_REASON	SMALLINT	IBM internal use only.
PARA_TYPE	CHAR(4)	IBM internal use only.
PART_INNER	CHAR(1)	IBM internal use only.
GRNU_KEYRNG	CHAR(1)	IBM internal use only.
OPEN_KEYRNG	CHAR(1)	IBM internal use only.



DSN_PREDICAT_TABLE

The predicate table, DSN_PREDICAT_TABLE, contains information about all of the predicates in a query.

 PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of the DSN_PREDICAT_TABLE

Table 159. DSN_PREDICAT_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
PREDNO	INTEGER NOT NULL	The predicate number, a number used to identify a predicate within a query.
TYPE	CHAR(8) NOT NULL	<p>A string used to indicate the type or the operation of the predicate. The possible values are:</p> <ul style="list-style-type: none">• 'AND'• 'OR'• 'EQUAL'• 'RANGE'• 'BETWEEN'• 'IN'• 'LIKE'• 'NOT LIKE'• 'EXISTS'• 'NOTEXIST'• 'SUBQUERY'• 'HAVING'• 'OTHERS'

Table 159. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
LEFT_HAND_SIDE	VARCHAR(128) NOT NULL	If the LHS of the predicate is a table column (LHS_TABNO > 0), then this column indicates the column name. Other possible values are: <ul style="list-style-type: none"> • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks
LEFT_HAND_PNO	INTEGER NOT NULL	If the LHS of the predicate is a table column (LHS_TABNO > 0), then this column indicates the column name. Other possible values are: <ul style="list-style-type: none"> • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks
LHS_TABNO	SMALLINT NOT NULL	If the LHS of the predicate is a table column, then this column indicates a number which uniquely identifies the corresponding table reference within a query.
LHS_QBNO	SMALLINT NOT NULL	If the LHS of the predicate is a table column, then this column indicates a number which uniquely identifies the corresponding table reference within a query.
RIGHT_HAND_SIDE	VARCHAR(128) NOT NULL	If the RHS of the predicate is a table column (RHS_TABNO > 0), then this column indicates the column name. Other possible values are: <ul style="list-style-type: none"> • 'VALUE' • 'COLEXP' • 'NONCOLEXP' • 'CORSUB' • 'NONCORSUB' • 'SUBQUERY' • 'EXPRESSION' • Blanks
RIGHT_HAND_PNO	INTEGER NOT NULL	If the predicate is a compound predicate (AND/OR), then this column indicates the second child predicate. However, this column is not reliable when the predicate tree consolidation happens. Use PARENT_PNO instead to reconstruct the predicate tree.
RHS_TABNO	CHAR(1) NOT NULL	If the RHS of the predicate is a table column, then this column indicates a number which uniquely identifies the corresponding table reference within a query.

Table 159. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
RHS_QBNO	CHAR(1) NOT NULL	If the RHS of the predicate is a subquery, then this column indicates a number which uniquely identifies the corresponding query block within a query.
FILTER_FACTOR	FLOAT NOT NULL	The estimated filter factor.
BOOLEAN_TERM	CHAR(1) NOT NULL	Whether this predicate can be used to determine the truth value of the whole WHERE clause.
SEARCHARG	CHAR(1) NOT NULL	Whether this predicate can be processed by data manager (DM). If it is not, then the relational data service (RDS) needs to be used to take care of it, which is more costly.
AFTER_JOIN	CHAR(1) NOT NULL	Indicates the predicate evaluation phase: 'A' After join 'D' During join blank Not applicable
ADDED_PRED	CHAR(1) NOT NULL	Whether it is generated by transitive closure, which means DB2 can generate additional predicates to provide more information for access path selection, when the set of predicates that belong to a query logically imply other predicates.
REDUNDANT_PRED	CHAR(1) NOT NULL	Whether it is a redundant predicate, which means evaluation of other predicates in the query already determines the result that the predicate provides.
DIRECT_ACCESS	CHAR(1) NOT NULL	Whether the predicate is direct access, which means one can navigate directly to the row through ROWID.
KEYFIELD	CHAR(1) NOT NULL	Whether the predicate includes the index key column of the involved table for all applicable indexes considered by DB2.
EXPLAIN_TIME	TIMESTAMP	The EXPLAIN timestamp.
CATEGORY	SMALLINT	IBM internal use only.
CATEGORY_B	SMALLINT	IBM internal use only.
TEXT	VARCHAR(2000)	The transformed predicate text; truncated if exceeds 2000 characters.
PRED_ENCODE	CHAR(1)	IBM internal use only.
PRED_CCSID	SMALLINT	IBM internal use only.
PRED_MCCSID	SMALLINT	IBM internal use only.
MARKER	CHAR(1)	Whether this predicate includes host variables, parameter markers, or special registers.
PARENT_PNO	INTEGER	The parent predicate number. If this predicate is a root predicate within a query block, then this column is 0.
NEGATION	CHAR(1)	Whether this predicate is negated via NOT.
LITERALS	VARCHAR(128)	This column indicates the literal value or literal values separated by colon symbols.
CLAUSE	CHAR(8)	The clause where the predicate exists: 'HAVING ' The HAVING clause 'ON ' The ON clause 'WHERE ' The WHERE clause

Table 159. DSN_PREDICAT_TABLE description (continued)

Column name	Data type	Description
GROUP_MEMBER	CHAR(8)	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.



DSN_PTASK_TABLE

The parallel tasks table, DSN_PTASK_TABLE, contains information about all of the parallel tasks in a query.



Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_PTASK_TABLE.

Table 160. DSN_PTASK_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
LPTNO	SMALLINT NOT NULL	The parallel task number.
KEYCOLID	SMALLINT	The key column ID (KEY range only).
DPSI	CHAR(1) NOT NULL	Indicates if a data partition secondary index (DPSI) is used.
LPTLOKEY	VARCHAR(40)	The low key value for this key column for this parallel task (KEY range only).
LPTHIKEY	VARCHAR(40)	The high key value for this key column for this parallel task (KEY range only).
LPTLOPAG	CHAR(4)	The low page information if partitioned by page range.
LPTLHIPAG	CHAR(4)	The high page information if partitioned by page range.
LPTLOPG ¹	CHAR(4)	The lower bound page number for this parallel task (Page range or DPSI enabled only).
LPTHIPG ¹	CHAR(4)	The upper bound page number for this parallel task (Page range or DPSI enabled only).
LPTLOPT ¹	SMALLINT	The lower bound partition number for this parallel task (Page range or DPSI enabled only).
LPTHIPT ¹	SMALLINT	The upper bound partition number for this parallel task (Page range or DPSI enabled only).
KEYCOLDT	SMALLINT	The data type for this key column (KEY range only).
KEYCOLPREC	SMALLINT	The precision/length for this key column (KEY range only).

Table 160. DSN_PTASK_TABLE description (continued)

Column name	Data type	Description
KEYCOLSCAL	SMALLINT	The scale for this key column (KEY range with Decimal datatype only).
EXPLAIN_TIME	TIMESTAMP NOT NULL	The EXPLAIN timestamp.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.

Notes:

1. The name of these columns originally contained the # symbol as the last character in the names. However, the names that contain these characters are obsolete and are no longer supported.



DSN_QUERYINFO_TABLE

The query information table, DSN_QUERYINFO_TABLE, contains information about the eligibility of query blocks for automatic query rewrite, information about the materialized query tables that are considered for eligible query blocks, and reasons why ineligible query blocks are not eligible.



Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

Table 161. Descriptions of columns in DSN_QUERYINFO_TABLE

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	INTEGER NOT NULL	A number that identifies each query block within a query.
QINAME1	VARCHAR(128) NOT NULL WITH DEFAULT	When TYPE='A' this is the creator of the accelerated query table.
QINAME2	VARCHAR(128) NOT NULL WITH DEFAULT	When TYPE='A' this is the name of the accelerated query table.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row, or blank.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable.
COLLID	VARCHAR(128) NOT NULL	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSNDDYNAMICSQLCACHE indicates that the row is for a cached statement.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN, or blank. For more information about the GROUP_MEMBER column, see
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time at which the statement is processed. This time is the same as the BIND_TIME column in PLAN_TABLE.
TYPE	CHAR(8) NOT NULL WITH DEFAULT	<p>The type of the output for this row:</p> <p>A This row is for an accelerated query table.</p>
QI_DATA	CLOB(2M) NOT NULL WITH DEFAULT	When TYPE='A': This column contains the description of the value in REASON_CODE.

Table 161. Descriptions of columns in DSN_QUERYINFO_TABLE (continued)

Column name	Data type	Description
SERVICE_INFO	BLOB(2M) NOT NULL WITH DEFAULT	IBM internal use only.

Notes:

1. The REASON_CODE column has the following values:

- | | |
|------------|--|
| 0 | The query block qualifies for routing to an accelerator with the DB2 internal object specified in QINAME1 and QINAME2. For example, the IBM Smart Analytics Optimizer, the associated data-mart name is recorded in the QINAME2 column and the following naming convention is used: DATAMARTNAME@ACCELERATORNAME@DIGITS |
| 1 | No accelerator was found when EXPLAIN was executed. If an accelerator is available, make sure it is started. |
| 2 | Rewrite not done because CURRENT REFRESH AGE <> ANY |
| 3 | The query was not routed to an accelerator because the estimated cost of the query is too low. |
| 4 | The query could not be routed to an accelerator because the query is not read-only. You can explicitly mark a query as read-only by using the FOR FETCH ONLY, FOR READ ONLY or WITH UR clause. |
| 5 | The query could not be routed to an accelerator because the query isolation level is not supported. Use isolation level cursor stability (CS) or uncommitted read (UR) to enable query acceleration. |
| 201 | There is at least one table referenced in the query block that does not appear in any of the defined data marts. |
| 204 | The query block could not be routed to an accelerator because the query block contains a table expression with correlated references. |
| 205 | The query block could not be routed to an accelerator because the query block contains an unsupported expression. The QI_DATA column contains the unsupported expression. |
| 206 | The query block could not be routed to an accelerator because the query block is a correlated subquery. Consider using a join instead of a correlated subquery. |
| 207 | The query block could not be routed to an accelerator because the query block is a type of work file that cannot be routed to an accelerator. A work file is the intermediate result of a UNION ALL, INTERSECT ALL, EXCEPT ALL, or an outer join that is materialized. |
| 208 | The query block could not be routed to an accelerator because an accelerator could not be found. For example, the IBM Smart Analytics Optimizer, the accelerator name shown in the QI_DATA column (naming convention: DATAMARTNAME@ACCELERATORNAME@DIGITS) was not found. If the accelerator is available, make sure that it is started. |
| 501 | The query block could not be routed to an accelerator because one or more tables in the FROM clause are not referenced by the DB2 internal object. |

- 502 The query block could not be routed to an accelerator because one or more columns in the GROUP BY clause are not referenced by the DB2 internal object.
- 503 The query block could not be routed to an accelerator because one or more columns in the SELECT clause are not referenced by the DB2 internal object. The QI_DATA column contains the non-matching SELECT clause.
- 504 The query block could not be routed to an accelerator because one or more columns in the HAVING clause are not referenced by the DB2 internal object. When available, the QI_DATA column contains the non-matching HAVING clause.
- 505 The query block could not be routed to an accelerator because one or more columns in the WHERE clause are not referenced by the DB2 internal object.. When available, the QI_DATA column contains the non-matching WHERE clause.
- 506 The query block could not be routed to an accelerator because the ON clause does not match with the defined references in the DB2 internal object. When available the QI_DATA column contains the non-matching ON clause.
- 508 Lossless checking failed for left outer join AQT. When available, the QI_DATA column contains the table name.

PSPI

DSN_QUERY_TABLE

The query table, DSN_QUERY_TABLE, contains information about a SQL statement, and displays the statement before and after query transformation.

PSPI

Unlike other EXPLAIN tables, rows in DSN_QUERY_TABLE are not populated for static SQL statements at BIND or REBIND with the EXPLAIN(YES) option.

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_QUERY_TABLE.

Table 162. DSN_QUERY_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
TYPE	CHAR(8) NOT NULL	The type of the data in the NODE_DATA column.
QUERY STAGE	CHAR(8) NOT NULL WITH DEFAULT	The stage during query transformation when this row is populated.
SEQNO	NOT NULL	The sequence number for this row if NODE_DATA exceeds the size of its column.
NODE_DATA	CLOB(2M)	The XML data containing the SQL statement and its query block, table, and column information.
EXPLAIN_TIME	TIMESTAMP	The EXPLAIN timestamp.
QUERY_ROWID	ROWID NOT NULL GENERATED ALWAYS	The ROWID of the statement.
GROUP MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
HASHKEY	INTEGER NOT NULL	The hash value of the contents in NODE_DATA
HAS_PRED	CHAR(1) NOT NULL	When NODE_DATA contains an SQL statement, this column indicates if the statement contains a parameter marker literal, non-parameter marker literal, or no predicates.



DSN_SORTKEY_TABLE

The sort key table, DSN_SORTKEY_TABLE, contains information about sort keys for all of the sorts required by a query.

 PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTEESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTEESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTEESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_SORTKEY_TABLE.

Table 163. DSN_SORTKEY_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
PLANNO	SMALLINT NOT NULL	The plan number, a number used to identify each miniplan with a query block.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSN_DYNAMICSQLCACHE indicates that the row is for a cached statement.
SORTNO	SMALLINT NOT NULL	The sequence number of the sort
ORDERNO	SMALLINT NOT NULL	The sequence number of the sort key
EXPTYPE	CHAR(3) NOT NULL	<p>The type of the sort key. The possible values are:</p> <ul style="list-style-type: none"> • 'COL' • 'EXP' • 'QRY'
TEXT	VARCHAR(128) NOT NULL	The sort key text, can be a column name, an expression, or a scalar subquery, or 'Record ID'.
TABNO	SMALLINT NOT NULL	The table number, a number which uniquely identifies the corresponding table reference within a query.

Table 163. DSN_SORTKEY_TABLE description (continued)

Column name	Data type	Description
COLNO	SMALLINT NOT NULL	The column number, a number which uniquely identifies the corresponding column within a query. Only applicable when the sort key is a column.
DATATYPE	CHAR(18)	The data type of sort key. The possible values are <ul style="list-style-type: none"> • 'HEXADECIMAL' • 'CHARACTER' • 'PACKED FIELD ' • 'FIXED(31)' • 'FIXED(15)' • 'DATE' • 'TIME' • 'VARCHAR' • 'PACKED FLD' • 'FLOAT' • 'TIMESTAMP' • 'UNKNOWN DATA TYPE'
LENGTH	INTEGER NOT NULL	The length of sort key.
CCSID	INTEGER NOT NULL	IBM internal use only.
ORDERCLASS	INTEGER NOT NULL	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The EXPLAIN timestamp.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.



DSN_SORT_TABLE

The sort table, DSN_SORT_TABLE, contains information about the sort operations required by a query.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_SORT_TABLE.

Table 164. DSN_SORT_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
PLANNO	SMALLINT NOT NULL	The plan number, a number used to identify each miniplan with a query block.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSNNDYNAMICSQLCACHE indicates that the row is for a cached statement.
SORTC	CHAR(5) NOT NULL WITH DEFAULT	<p>Indicates the reasons for sort of the composite table. The reasons are shown as a series of bytes:</p> <ul style="list-style-type: none"> • Byte 1 is 'G' if the reason is GROUP BY, or otherwise blank. • The second byte is 'J' if the reason is JOIN, or otherwise blank. • Byte is 'O' if the reason is ORDER BY, or otherwise blank. • The fourth byte is 'U' if the reason is uniqueness, or otherwise blank.

Table 164. DSN_SORT_TABLE description (continued)

Column name	Data type	Description
SORTN	CHAR(5) NOT NULL WITH DEFAULT	Indicates the reasons for sort of the new table. The reasons are shown as a series of bytes: <ul style="list-style-type: none"> • The first byte is 'G' if the reason is GROUP BY, or otherwise blank. • The second byte is 'J' if the reason is JOIN, or otherwise blank. • The third byte is 'O' if the reason is ORDER BY, or otherwise blank. • The fourth by is 'U' if the reason is uniqueness, or otherwise blank.
SORTNO	SMALLINT NOT NULL	The sequence number of the sort.
KEYSIZE	SMALLINT NOT NULL	The sum of the lengths of the sort keys.
ORDERCLASS	INTEGER NOT NULL	IBM internal use only.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The EXPLAIN timestamp.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.



DSN_STATEMENT_CACHE_TABLE

The statement cache table, DSN_STATEMENT_CACHE_TABLE, contains information about the SQL statements in the statement cache, information captured as the results of an EXPLAIN STATEMENT CACHE ALL statement.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userid You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table shows the descriptions of the columns in DSN_STATEMENT_CACHE_TABLE.

Table 165. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE

Column name	Data Type	Description
STMT_ID	INTEGER NOT NULL	The statement ID; this value is the EDM unique token for the statement.
STMT_TOKEN	VARCHAR(240) NOT NULL	The statement token; you provide this value as an identification string.
COLLID	VARCHAR(128) NOT NULL	The collection ID; because DSN_STATEMENT_CACHE_TABLE is used to explain cached statements, the value of this column is DSDYNAMICSQLCACHE.
PROGRAM_NAME	VARCHAR(128) NOT NULL	The name of the package or DBRM that performed the initial PREPARE for the statement.
INV_DROPALT	CHAR(1) NOT NULL	This column indicates if the statement has been invalidated by a DROP or ALTER statement.
INV_REVOKE	CHAR(1) NOT NULL	This column indicates if the statement has been invalidated by a REVOKE statement.
INV_LRU	CHAR(1) NOT NULL	This column indicates if the statement has been removed from the cache by LRU.
INV_RUNSTATS	CHAR(1) NOT NULL	This column indicates if the statement has been invalidated by RUNSTATS.
CACHED_TS	TIMESTAMP NOT NULL	The timestamp when the statement was stored in the dynamic statement cache.
USERS	INTEGER NOT NULL	The number of current users of the statement. This number indicates the users that have prepared or run the statement during their current unit of work. 1 on page 1984,
COPIES	INTEGER NOT NULL	The number of copies of the statement that are owned by all threads in the system. 1 on page 1984,
LINES	INTEGER NOT NULL	The precompiler line number from the initial PREPARE of the statement. 1 on page 1984
PRIMAUTH	VARCHAR(128) NOT NULL	The primary authorization ID that did the initial PREPARE of the statement.
CURSQLID	VARCHAR(128) NOT NULL	The CURRENT SQLID that did the initial PREPARE of the statement.
BIND_QUALIFIER	VARCHAR(128) NOT NULL	The BIND qualifier. For unqualified table names, this is the object qualifier.
BIND_ISO	CHAR(2) NOT NULL	The value of the ISOLATION BIND option that is in effect for this statement. The value will be one of the following values: 'UR' Uncommitted read 'CS' Cursor stability 'RS' Read stability 'RR' Repeatable read
BIND_CDATA	CHAR(1) NOT NULL	The value of the CURRENTDATA BIND option that is in effect for this statement. The value will be one of the following values: 'Y' CURRENTDATA(YES) 'N' CURRENTDATA(NO)
BIND_DYNRL	CHAR(1) NOT NULL	The value of the DYNAMICRULES BIND option that is in effect for this statement. The value will be one of the following values: 'B' DYNAMICRULE(BIND) 'R' DYNAMICRULES(RUN)

Table 165. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE (continued)

Column name	Data Type	Description
BIND_DEGRE	CHAR(1) NOT NULL	The value of the CURRENT DEGREE special register that is in effect for this statement. The value will be one of the following values: 'A' CURRENT DEGREE = ANY '1' CURRENT DEGREE = 1
BIND_SQLRL	CHAR(1) NOT NULL	The value of the CURRENT RULES special register that is in effect for this statement. The value will be one of the following values: 'D' CURRENT RULES = DB2 'S' CURRENT RULES = SQL
BIND_CHOLD	CHAR(1) NOT NULL	The value of the WITH HOLD attribute of the PREPARE for this statement. The value will be one of the following values: 'Y' Initial PREPARE specified WITH HOLD 'N' Initial PREPARE specified WITHOUT HOLD
STAT_TS	TIMESTAMP NOT NULL	Timestamp of the statistics. This is the timestamp when IFCID 318 is started.
STAT_EXEC	INTEGER NOT NULL	The number of times this statement has been run. For a statement with a cursor, this is the number of OPENS. 1 on page 1984,
STAT_GPAG	INTEGER NOT NULL	The number of getpage operations that are performed for the statement. 1 on page 1984
STAT_SYNR	INTEGER NOT NULL	The number of synchronous buffer reads that are performed for the statement. 1 on page 1984
STAT_WRIT	INTEGER NOT NULL	The number of buffer write operations that are performed for the statement. 1 on page 1984
STAT_EROW	INTEGER NOT NULL	The number of rows that are examined for the statement. 1 on page 1984
STAT_PROW	INTEGER NOT NULL	The number of rows that are processed for the statement. 1 on page 1984
STAT_SORT	INTEGER NOT NULL	The number of sorts that are performed for the statement. 1 on page 1984
STAT_INDX	INTEGER NOT NULL	The number of index scans that are performed for the statement. 1 on page 1984,
STAT_RSCN	INTEGER NOT NULL	The number of table space scans that are performed for the statement. 1 on page 1984,
STAT_PGRP	INTEGER NOT NULL	The number of parallel groups that are created for the statement. 1 on page 1984,
STAT_ELAP	FLOAT NOT NULL	The accumulated elapsed time that is used for the statement.
STAT_CPU	FLOAT NOT NULL	The accumulated CPU time that is used for the statement.
STAT_SUS_SYNIO	FLOAT NOT NULL	The accumulated wait time for synchronous I/O operations for the statement.
STAT_SUS_LOCK	FLOAT NOT NULL	The accumulated wait time for lock and latch requests for the statement.
STAT_SUS_SWIT	FLOAT NOT NULL	The accumulated wait time for synchronous execution unit switch for the statement.
STAT_SUS_GLCK	FLOAT NOT NULL	The accumulated wait time for global locks for this statement.
STAT_SUS_OTHR	FLOAT NOT NULL	The accumulated wait time for read activity that is done by another thread.

Table 165. Descriptions of columns in DSN_STATEMENT_CACHE_TABLE (continued)

Column name	Data Type	Description
STAT_SUS_OTHW	FLOAT NOT NULL	The accumulated wait time for write activity done by another thread.
STAT_RIDLIMT	INTEGER NOT NULL	The number of times a RID list was not used because the number of RIDs would have exceeded DB2 limits. 1,
STAT_RIDSTOR	INTEGER NOT NULL	The number of times a RID list was not used because there is not enough storage available to hold the list of RIDs. 1,
EXPLAIN_TS	TIMESTAMP NOT NULL	The timestamp for when the statement cache table is populated.
SCHEMA	VARCHAR(128) NOT NULL	The value of the CURRENT SCHEMA special register.
STMT_TEXT	CLOB(2M) NOT NULL	The statement that is being explained.
STMT_ROWID	ROWID NOT NULL	The ROWID of the statement.
BIND_RO_TYPE	CHAR(1) NOT NULL	The current specification of the REOPT option for the statement: 'N' REOPT(NONE) or its equivalent 'I' REOPT(ONCE) or its equivalent 'A' REOPT(AUTO) or its equivalent 'O' The current plan is deemed optimal and there is no need for REOPT(AUTO)
BIND_RA_TOT	INTEGER NOT NULL	The total number of REBIND commands that have been issued for the dynamic statement because of the REOPT(AUTO) option.1,

Notes:

1. If the specified value exceeds 2147483647, the column contains the value 2147483647.



DSN_STATEMNT_TABLE

The statement table, DSN_STATEMNT_TABLE, contains information about the estimated cost of specified SQL statements.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the content of each column in STATEMNT_TABLE.

Table 166. Descriptions of columns in DSN_STATEMNT_TABLE

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
COLLID	VARCHAR(128) NOT NULL	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSDYNAMICSQLCACHE indicates that the row is for a cached statement.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 that executed EXPLAIN, or blank. See the description of the GROUP_MEMBER column in PLAN_TABLE for more information.
EXPLAIN_TIME	TIMESTAMP NOT NULL	The time at which the statement is processed. This time is the same as the BIND_TIME column in PLAN_TABLE.

Table 166. Descriptions of columns in DSN_STATEMENT_TABLE (continued)

Column name	Data type	Description
STMT_TYPE	CHAR(6) NOT NULL	<p>The type of statement being explained. Possible values are:</p> <p>SELECT SELECT</p> <p>INSERT INSERT</p> <p>UPDATE UPDATE</p> <p>MERGE MERGE</p> <p>DELETE DELETE</p> <p>TRUNCATE TRUNCATE</p> <p>SELUPD SELECT with FOR UPDATE OF</p> <p>DELCUR DELETE WHERE CURRENT OF CURSOR</p> <p>UPDCUR UPDATE WHERE CURRENT OF CURSOR</p>
COST_CATEGORY	CHAR(1) NOT NULL	<p>Indicates if DB2 was forced to use default values when making its estimates. Possible values:</p> <p>A Indicates that DB2 had enough information to make a cost estimate without using default values.</p> <p>B Indicates that some condition exists for which DB2 was forced to use default values. See the values in REASON to determine why DB2 was unable to put this estimate in cost category A.</p>
PROCMS	INTEGER NOT NULL	<p>The estimated processor cost, in milliseconds, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 milliseconds, which is equivalent to approximately 24.8 days. If the estimated value exceeds this maximum, the maximum value is reported. If an accelerator product is used the difference is reflected in this value.</p>
PROCSU	INTEGER NOT NULL	<p>The estimated processor cost, in service units, for the SQL statement. The estimate is rounded up to the next integer value. The maximum value for this cost is 2147483647 service units. If the estimated value exceeds this maximum, the maximum value is reported. If an accelerator product is used, this value represents the estimated cost including any impact of acceleration.</p>

Table 166. Descriptions of columns in DSN_STATEMNT_TABLE (continued)

Column name	Data type	Description
REASON	VARCHAR(254)	<p>A string that indicates the reasons for putting an estimate into cost category B.</p> <p>HAVING CLAUSE A subselect in the SQL statement contains a HAVING clause.</p> <p>HOST VARIABLES The statement uses host variables, parameter markers, or special registers.</p> <p>MATERIALIZATION Statistics are missing because the statement uses materialized views or nested table expresses.</p> <p>PROFILEID value When profile monitoring is used for the statement, the value of the PROFILEID column in SYSIBM.DSN_PROFILE_TABLE.</p> <p>REFERENTIAL CONSTRAINTS Referential constraints of the type CASCADE or SET NULL exist on the target table of a DELETE statement.</p> <p>TABLE CARDINALITY The cardinality statistics are missing for one or more of the tables that are used in the statement.</p> <p>TRIGGERS Triggers are defined on the target table of an insert, update, or delete operation.</p> <p>UDF The statement uses user-defined functions.</p>
STMT_ENCODE	CHAR(1)	<p>Encoding scheme of the statement. If the statement represents a single CCSID set, the possible values are:</p> <p>A ASCII E EBCDIC U Unicode</p> <p>If the statement has multiple CCSID sets, the value is M.</p>
TOTAL_COST	FLOAT NOT NULL	<p>The overall estimated cost of the statement. If an accelerator product is used the benefit is reflected in this value. This cost should be used only for reference purposes.</p>



DSN_STRUCT_TABLE

The structure table, DSN_STRUCT_TABLE, contains information about all of the query blocks in a query.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_STRUCT_TABLE

Table 167. DSN_STRUCT_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
QBLOCKNO	SMALLINT NOT NULL	The query block number, a number used to identify each query block within a query.
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
PARENT	SMALLINT NOT NULL	The parent query block number of the current query block in the structure of SQL text; this is the same as the PARENT_QBLOCKNO in PLAN_TABLE.
TIMES	FLOAT NOT NULL	The estimated number of rows returned by Data Manager; also the estimated number of times this query block is executed.
ROWCOUNT	INTEGER NOT NULL	The estimated number of rows returned by RDS (Query Cardinality).
ATOPEN	CHAR(1) NOT NULL	Whether the query block is moved up for do-at-open processing; 'Y' if done-at-open; 'N': otherwise.
CONTEXT	CHAR(10) NOT NULL	<p>This column indicates what the context of the current query block is. The possible values are:</p> <ul style="list-style-type: none">• 'TOP LEVEL'• 'UNION'• 'UNION ALL'• 'PREDICATE'• 'TABLE EXP'• 'UNKNOWN'
ORDERNO	SMALLINT NOT NULL	Not currently used.
DOATOPEN_PARENT	SMALLINT NOT NULL	The parent query block number of the current query block; Do-at-open parent if the query block is done-at-open, this may be different from the PARENT_QBLOCKNO in PLAN_TABLE.

Table 167. DSN_STRUCT_TABLE description (continued)

Column name	Data type	Description
QBLOCK_TYPE	CHAR(6) NOT NULL WITH DEFAULT	<p>This column indicates the type of the current query block. The possible values are</p> <ul style="list-style-type: none"> • 'SELECT' • 'INSERT' • 'UPDATE' • 'DELETE' • 'SELUPD' • 'DELCUR' • 'UPDCUR' • 'CORSUB' • 'NCOSUB' • 'TABLEX' • 'TRIGGR' • 'UNION' • 'UNIONA' • 'CTE' <p>It is equivalent to QBLOCK_TYPE column in PLAN_TABLE, except for CTE.</p>
EXPLAIN_TIME	TIMESTAMP NOT NULL	The EXPLAIN timestamp.
QUERY_STAGE	CHAR(8) NOT NULL	IBM internal use only.
GROUP_MEMBER	CHAR(8) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.



DSN_VIEWREF_TABLE

The view reference table, DSN_VIEWREF_TABLE, contains information about all of the views and materialized query tables that are used to process a query.

PSPI

Recommendation: Do not manually insert data into or delete data from EXPLAIN tables. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools.

Important: If mixed data strings are allowed on a DB2 subsystem, EXPLAIN tables must be created with CCSID UNICODE. This includes, but is not limited to, mixed data strings that are used for tokens, SQL statements, application names, program names, correlation names, and collection IDs.

Important: EXPLAIN tables in any pre-Version 8 format or EXPLAIN tables that are encoded in EBCDIC are deprecated.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of each EXPLAIN table can be created with the SYSIBM qualifier. SQL optimization tools use these tables. You can find the SQL statement for creating these tables in member DSNTESC of the SDSNSAMP library.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind, or rebind, a plan or package with the EXPLAIN(YES) option. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

DB2OSCA

SQL optimization tools, such as Optimization Service Center for DB2 for z/OS, create EXPLAIN tables to collect information about SQL statements and workloads to enable analysis and tuning processes. You can find the SQL statements for creating EXPLAIN tables in member DSNTIJOS of the SDSNSAMP library. You can also create this table from the Optimization Service Center interface on your workstation.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

The following table describes the columns of DSN_VIEWREF_TABLE.

Table 168. DSN_VIEWREF_TABLE description

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.</p>
APPLNAME	VARCHAR(24) NOT NULL	The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.
PROGNAME	VARCHAR(128) NOT NULL	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.
VERSION	VARCHAR(128)	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable.
COLLID	VARCHAR(128)	The collection ID for the package. Applies only to an embedded EXPLAIN statement that is executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. The value DSDYNAMICSQLCACHE indicates that the row is for a cached statement.
CREATOR	VARCHAR(128)	Authorization ID of the owner of the object.
NAME	VARCHAR(128)	Name of the object.
TYPE	CHAR(1) NOT NULL WITH DEFAULT	<p>The type of the object:</p> <p>'V' View</p> <p>'R' MQT that has been used to replace the base table for rewrite</p> <p>'M' MQT</p>
MQTUSE	SMALLINT	IBM internal use only.
EXPLAIN TIME	TIMESTAMP	The EXPLAIN timestamp.
GROUP_MEMBER	VARCHAR(24) NOT NULL	The member name of the DB2 subsystem that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.



Tables that are used by accelerators

Table spaces and indexes for accelerators

Tables that are used for accelerators are contained in certain table spaces and have indexes.

The following table lists the table space and indexes for each table that is used for accelerators and lists the index fields for each index. The indexes are in ascending order, except where noted.

Table 169. Table spaces and indexes for the tables that are used for accelerators

TABLE SPACE	TABLE	INDEX	
DSNDB06. ...	SYSACCEL. ...	SYSACCEL. ...	INDEX FIELDS
	SYSACCELERATORS	DSNACC01	ACCELERATORNAME
	SYSACCELIPLIST	DSNACI01	ACCELIPNAME.ACCELIPADDR

SYSACCEL.SYSACCELERATORS table

The SYSACCEL.SYSACCELERATORS table contains one row that describes the characteristics of each accelerator server.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
ACCELERATORNAME	VARCHAR(128) NOT NULL	A unique name for the accelerator server. This is the name by which the accelerator server is known to the local DB2 accelerated query tables.	
ACCELERATOR-IPNAME	VARCHAR(24) NOT NULL	Identifies the TCP/IP attributes that are associated with this accelerator server. The value of this column corresponds to a row in the SYSACCEL.SYSACCELIPLIST table (identified by the value of the ACCELIPNAME column). The corresponding row in the SYSACCEL.SYSACCELIPLIST specifies the TCP/IP communication attributes for this accelerator server. Note: While this column is defined as VARCHAR(24) it must not contain more than 8 EBCDIC characters.	
ACCELERATORPORT	VARCHAR(96) NOT NULL	Identifies the TCP/IP port number or service name of the remote database server: Format Meaning Left justified with 1-5 numeric characters Assumed to be the TCP/IP port number of the remote database server Any other value Assumed to be a TCP/IP service name If the value is a TCP/IP service name, the value can be converted to a TCP/IP port number by using the TCP/IP getservbyname socket call. TCP/IP service names are not case sensitive. Note: While this column is defined as VARCHAR(96) it must not contain more than 32 EBCDIC characters.	
ACCELERATOR-AUTHTOKEN	VARCHAR(256) NOT NULL	The authentication token for the accelerator server.	
ACCELERATOR-SECURE	CHAR(1) NOT NULL	Indicates if the use of the Secure Socket Layer (SSL) protocol is required: Y Indicates a secure connection using SSL is required. N Indicates a secure connection is not required.	
IBMREQD	CHAR(1)	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators	

SYSACCEL.SYSACCELIPLIST table

The SYSACCEL.SYSACCELIPLIST table contains one row for each IP address that is associated with each accelerator server.

Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
ACCELIPNAME	VARCHAR(24)	This value corresponds to the value of the ACCELERATORIPNAME column of the SYSACCEL.SYSACCELERATORS table. The values of the other columns in the row of the SYSACCELIPLIST table apply to the server that is identified by the ACCELERATORIPNAME column of the SYSACCEL.SYSACCELERATORS table. Note: While this column is defined as VARCHAR(24) it must not contain more than 8 EBCDIC characters.	
ACCELIPADDR	VARCHAR(254)	Contains the IP address or domain name of a remote TCP/IP host of the server.	
IBMREQD	CHAR(1)	A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. For all other values, see Release dependency indicators.	

Using the catalog in database design

Retrieving information from the catalog by using SQL statements, can be helpful in designing your relational database.

GUPI

DB2 SQL Reference lists all the DB2 catalog tables and the information stored in them.

The information in the catalog is vital to normal DB2 operation. You can *retrieve* catalog information, but *changing* it can have serious consequences. Therefore you cannot execute insert or delete operations that affect the catalog, and only a limited number of columns exist that you can update. Exceptions to these restrictions are the SYSIBM.SYSSTRINGS, SYSIBM.SYSCOLDIST, and SYSIBM.SYSCOLDISTSTATS catalog tables, into which you can insert rows and proceed to update and delete rows.

To retrieve information from the catalog, you need at least the SELECT privilege on the appropriate catalog tables.

Note: Some catalog queries can result in long table space scans.

GUPI

Retrieving catalog information about DB2 storage groups

Query SYSIBM.SYSTOGROUP and SYSIBM.SYSVOLUMES to obtain information about DB2 storage groups and the volumes in those storage groups.

GUPI

SYSIBM.SYSSTOGROUP and SYSIBM.SYSVOLUMES contain information about DB2 storage groups and the volumes in those storage groups. The following query shows what volumes are in a DB2 storage group, how much space is used, and when that space was last calculated.

```
SELECT SGNAME,VOLID,SPACE,SPCDATE
FROM SYSIBM.SYSVOLUMES,SYSIBM.SYSSTOGROUP
WHERE SGNAME=NAME
ORDER BY SGNAME;
```

GUIP

Retrieving catalog information about a table

Query SYSIBM.SYSTABLES to obtain information about every table, view, and alias in your DB2 system.

GUIP

SYSIBM.SYSTABLES contains a row for every table, view, and alias in your DB2 system. Each row tells you whether the object is a table, a view, or an alias, its name, who created it, what database it belongs to, what table space it belongs to, and other information. SYSTABLES also has a REMARKS column in which you can store your own information about the table in question.

See “Adding and retrieving comments” on page 2006 for more information about how to do this.

The following statement displays all the information for the project activity sample table:

```
SELECT *
FROM SYSIBM.SYSTABLES
WHERE NAME = 'PROJACT'
AND CREATOR = 'DSN8910';
```

GUIP

Retrieving catalog information about partition order

Query SYSIBM.SYSTABLEPART to obtain information about the key order or logical partition order of a table.

GUIP

The LOGICAL_PART column in SYSIBM.SYSTABLEPART contains information for key order or logical partition order.

The following statement displays information on partition order in ascending limit value order:

```
SELECT LIMITKEY, PARTITION
FROM SYSIBM.SYSTABLEPART
ORDER BY LOGICAL_PART;
```

GUIP

Retrieving catalog information about aliases

Query SYSIBM.SYSTABLES to obtain information about aliases.

GUPI

Three columns in SYSIBM.SYSTABLES are used only for aliases:

- LOCATION contains your subsystem's location name for the remote system, if the object on which the alias is defined resides at a remote subsystem.
- TBCreator contains the schema table or view.
- TBNAME contains the name of the table or the view.

These sample user-defined functions make it easy to find information about aliases. See *DB2 SQL Reference* for more information.

- TABLE_NAME returns the name of a table, view, or undefined object found after resolving aliases for a user-specified object.
- TABLE_SCHEMA returns the schema name of a table, view, or undefined object found after resolving aliases for a user-specified object.
- TABLE_LOCATION returns the location name of a table, view, or undefined object found after resolving aliases for a user-specified object.

The NAME and CREATOR columns of SYSTABLES contain the name and schema of the alias, and three other columns contain the following information for aliases:

- TYPE is A.
- DBNAME is DSNDB06.
- TSNAME is SYSDBAUT.

If similar tables at different locations have names with the same second and third parts, you can retrieve the aliases for them with a query like this one:

```
SELECT LOCATION, CREATOR, NAME
  FROM SYSIBM.SYSTABLES
 WHERE TBCreator='DSN8910' AND TBNAME='EMP'
    AND TYPE='A';
```

GUPI

Retrieving catalog information about columns

Query SYSIBM.SYSCOLUMNS to obtain information about the columns of a table or view.

GUPI

SYSIBM.SYSCOLUMNS has one row for each column of every table and view. Query it, for example, if you cannot remember the column names of a table or view.

The following statement retrieves information about columns in the sample department table:

```
SELECT NAME, TBNAME, COLTYPE, LENGTH, NULLS, DEFAULT
  FROM SYSIBM.SYSCOLUMNS
 WHERE TBNAME='DEPT'
    AND TBCreator = 'DSN8910';
```

The result is shown below; for each column, the following information about each column is given:

- The column name
- The name of the table that contains it
- Its data type
- Its length attribute
- Whether it allows nulls
- Whether it allows default values

NAME	TBNAME	COLTYPE	LENGTH	NULLS	DEFAULT
DEPTNO	DEPT	CHAR	3	N	N
DEPTNAME	DEPT	VARCHAR	36	N	N
MGRNO	DEPT	CHAR	6	Y	N
ADMDEPT	DEPT	CHAR	3	N	N

For LOB columns, the LENGTH column shows the length of the pointer to the LOB. For an example of a query showing the actual LOB length, see “Retrieving catalog information about LOBs” on page 2004.

GUIP

Retrieving catalog information about indexes

Query SYSIBM.SYSINDEXES to obtain information about indexes.

GUIP

SYSIBM.SYSINDEXES contains a row for every index, including indexes on catalog tables. The following example retrieves a row about an index named XEMPL2.

```
SELECT *
  FROM SYSIBM.SYSINDEXES
 WHERE NAME = 'XEMPL2'
 AND CREATOR = 'DSN8910';
```

A table can have more than one index. To display information about all the indexes of a table, enter a statement like this one:

```
SELECT *
  FROM SYSIBM.SYSINDEXES
 WHERE TBNAME = 'EMP'
 AND TBCREATOR = 'DSN8910';
```

GUIP

Retrieving catalog information about views

For every view you create, DB2 stores descriptive information in several catalog tables. Query these catalog tables to obtain information about views in your database.

GUIP

The following actions occur in the catalog after the execution of CREATE VIEW:

- A row is inserted into SYSIBM.SYSTABLES.
- A row is inserted into SYSIBM.SYSTABAUTH to record the owner's privileges on the view.
- For each column of the view, a row is inserted into SYSIBM.SYSCOLUMNS.

- One or more rows are inserted into the SYSIBM.SYSVIEWS table to record the text of the CREATE VIEW statement.
- For each table or view on which the view is dependent, a row is inserted into SYSIBM.SYSVIEWDEP to record the dependency.
- A row is inserted into SYSIBM.SYSVTREE, and possibly into SYSIBM.SYSVLTREE, to record the parse tree of the view (an internal representation of its logic).

Users might want a view of one or more of those tables, containing information about their own tables and views.

GUIP

Retrieving catalog information about authorizations

Query SYSIBM.SYSTABAUTH to obtain information about who can access your data.

GUIP

SYSIBM.SYSTABAUTH contains information about the privileges held by authorization IDs over tables and views

The following query retrieves the names of all users who have been granted access to the DSN8910.DEPT table.

```
SELECT GRANTEE
  FROM SYSIBM.SYSTABAUTH
 WHERE TTNAME = 'DEPT'
    AND GRANTEETYPE <> 'P'
    AND TCREATOR = 'DSN8910';
```

GRANTEE is the name of the column that contains authorization IDs for users of tables. TTNAME and TCREATOR specify the DSN8910.DEPT table. The clause GRANTEETYPE <> 'P' ensures that you retrieve the names only of users (not application plans or packages) that have authority to access the table.

GUIP

Retrieving catalog information about primary keys

Query SYSIBM.SYSCOLUMNS for information about primary keys.

GUIP

SYSIBM.SYSCOLUMNS identifies columns of a primary key in column KEYSEQ; a nonzero value indicates the place of a column in the primary key. To retrieve the creator, database, and names of the columns in the primary key of the sample project activity table using SQL statements, execute:

```
SELECT TBCREATOR, TBNAME, NAME, KEYSEQ
  FROM SYSIBM.SYSCOLUMNS
 WHERE TBCREATOR = 'DSN8910'
    AND TBNAME = 'PROJACT'
    AND KEYSEQ > 0
    ORDER BY KEYSEQ;
```

SYSIBM.SYSINDEXES identifies the primary index of a table by the value P in column UNIQUERULE. To find the name, creator, database, and index space of the primary index on the project activity table, execute:

```
SELECT TBCREATOR, TBNAME, NAME, CREATOR, DBNAME, INDEXSPACE
FROM SYSIBM.SYSINDEXES
WHERE TBCREATOR = 'DSN8910'
AND TBNAME = 'PROJACT'
AND UNIQUERULE = 'P';
```

GUIP

Retrieving catalog information about foreign keys

Query SYSIBM.SYSRELS and SYSIBM.SYSFOREIGNKEYS to obtain information about referential constraints and the columns of the foreign key that defines the constraint.

GUIP

SYSIBM.SYSRELS contains information about referential constraints, and each constraint is uniquely identified by the schema and name of the dependent table and the constraint name (RELNAME). SYSIBM.SYSFOREIGNKEYS contains information about the columns of the foreign key that defines the constraint. To retrieve the constraint name, column names, and parent table names for every relationship in which the project table is a dependent, execute:

```
SELECT A.CREATOR, A.TBNAME, A.RELNAME, B.COLNAME, B.COLSEQ,
A.REFTBCREATOR, A.REFTBNAME
FROM SYSIBM.SYSRELS A, SYSIBM.SYSFOREIGNKEYS B
WHERE A.CREATOR = 'DSN8910'
AND B.CREATOR = 'DSN8910'
AND A.TBNAME = 'PROJ'
AND B.TBNAME = 'PROJ'
AND A.RELNAME = B.RELNAME
ORDER BY A.RELNAME, B.COLSEQ;
```

You can use the same tables to find information about the foreign keys of tables to which the project table is a parent, as follows:

```
SELECT A.RELNAME, A.CREATOR, A.TBNAME, B.COLNAME, B.COLNO
FROM SYSIBM.SYSRELS A, SYSIBM.SYSFOREIGNKEYS B
WHERE A.REFTBCREATOR = 'DSN8910'
AND A.REFTBNAME = 'PROJ'
AND A.RELNAME = B.RELNAME
ORDER BY A.RELNAME, B.COLNO;
```

GUIP

Retrieving catalog information about check pending

Query SYSIBM.SYSTABLESPACE to obtain information about table spaces that are in check-pending status.

GUIP

SYSIBM.SYSTABLESPACE indicates that a table space is in check-pending status by a value in column STATUS: P if the entire table space has that status, S if the status has a scope of less than the entire space. To list all table spaces whose use is restricted for *any* reason, issue this command:

-DISPLAY DATABASE (*) SPACENAM(*) RESTRICT

To retrieve the names of table spaces in check-pending status only, with the names of the tables they contain, execute:

```
SELECT A.DBNAME, A.NAME, B.CREATOR, B.NAME
FROM SYSIBM.SYSTABLESPACE A, SYSIBM.SYSTABLES B
WHERE A.DBNAME = B.DBNAME
AND A.NAME = B.TSNAME
AND (A.STATUS = 'P' OR A.STATUS = 'S')
ORDER BY 1, 2, 3, 4;
```

GUIP

Retrieving catalog information about check constraints

Query SYSIBM.SYSCHECKS and SYSIBM.SYSCHECKDEP to obtain information about check constraints.

GUIP

Information about check constraints is stored in the DB2 catalog in:

- SYSIBM.SYSCHECKS, which contains one row for each check constraint defined on a table
- SYSIBM.SYSCHECKDEP, which contains one row for each reference to a column in a check constraint

The following query shows all check constraints on all tables named SIMPDEPT and SIMPEMPL in order by column name within table schema. It shows the name, authorization ID of the creator, and text for each constraint. A constraint that uses more than one column name appears more than once in the result.

```
CREATE TABLE SIMPDEPT
(DEPTNO CHAR(3) NOT NULL,
DEPTNAME VARCHAR(12) CONSTRAINT CC1 CHECK (DEPTNAME IS NOT NULL),
MGRNO CHAR(6),
MGRNAME CHAR(6));

SELECT A.TBOWNER, A.TBNAME, B.COLNAME,
A.CHECKNAME, A.CREATOR, A.CHECKCONDITION
FROM SYSIBM.SYSCHECKS A, SYSIBM.SYSCHECKDEP B
WHERE A.TBOWNER = B.TBOWNER
AND A.TBNAME = B.TBNAME
AND B.TBNAME = 'SIMPDEPT'
AND A.CHECKNAME = B.CHECKNAME
ORDER BY TBOWNER, TBNAME, COLNAME;
```

GUIP

Retrieving catalog information about LOBs

Query SYSIBM.SYSAUXRELS to obtain information about the relationship between a base table and an auxiliary table.

GUIP

SYSIBM.SYSAUXRELS contains information about the relationship between a base table and an auxiliary table. For example, this query returns information about the name of the LOB columns for the employee table and its associated auxiliary table schema and name:

```
SELECT COLNAME, PARTITION, AXTBOWNER, AXTBNAME
FROM SYSIBM.SYSAUXRELS
WHERE TBNAME = 'EMP' AND TBOWNER = 'DSN8910';
```

Information about the length of a LOB is in the LENGTH2 column of SYSCOLUMNS. You can query information about the length of the column as it is returned to an application with the following query:

```
SELECT NAME, TBNAME, COLTYPE, LENGTH2, NULLS, DEFAULT
FROM SYSIBM.SYSCOLUMNS
WHERE TBNAME='DEPT'
AND TBCREATOR = 'DSN8910';
```

GUIP

Retrieving catalog information about user-defined functions and stored procedures

Query SYSIBM.SYSROUTINES to obtain information about user-defined functions and stored procedures.

GUIP

You can use this example to find packages with stored procedure that were created prior to Version 6 and then migrated to SYSIBM.SYSROUTINES:

```
SELECT SCHEMA, NAME FROM SYSIBM.SYSROUTINES
WHERE ROUTINETYPE = 'P';
```

You can use this query to retrieve information about user-defined functions:

```
SELECT SCHEMA, NAME, FUNCTION_TYPE, PARM_COUNT FROM SYSIBM.SYSROUTINES
WHERE ROUTINETYPE='F';
```

Stored procedures created before Version 6 have different authorization requirements than those created in Version 6. See *DB2 Application Programming and SQL Guide* for more information.

GUIP

Retrieving catalog information about triggers

Query SYSIBM.SYSTRIGGERS to obtain information about the triggers defined in your databases.

GUIP

To find all the triggers defined on a particular table, their characteristics, and to determine the order they are activated in, issue this query:

```
SELECT DISTINCT SCHEMA, NAME, TRIGTIME, TRIGEVENT, GRANULARITY, CREAEDTS
FROM SYSIBM.SYSTRIGGERS
WHERE TBNAME = 'EMP' AND TBOWNER = 'DSN8910';
```

Issue this query to retrieve the text of a particular trigger:

```
SELECT TEXT, SEQNO
FROM SYSIBM.SYSTRIGGERS
WHERE SCHEMA = schema_name
AND NAME = trigger_name
ORDER BY SEQNO;
```

Or to determine triggers that must be rebound because they are invalidated after objects are dropped or altered, issue this query:

```
SELECT COLLID, NAME
FROM SYSIBM.SYSPACKAGE
WHERE TYPE = 'T'
AND (VALID = 'N' OR OPERATIVE = 'N');
```

GUPI

Retrieving catalog information about sequences

Query SYSIBM.SYSSEQUENCES and SYSIBM.SYSSEQUENCEAUTH to obtain information about sequences.

GUPI

SYSIBM.SYSSEQUENCES and SYSIBM.SYSSEQUENCEAUTH contains information about sequences. To retrieve the attributes of a sequence, issue this query:

```
SELECT *
FROM SYSIBM.SYSSEQUENCES
WHERE NAME = 'MYSEQ' AND SCHEMA = 'USER1B';
```

Issue this query to determine the privileges that user USER1B has on sequences:

```
SELECT GRANTOR, NAME, DATEGRANTED, ALTERAUTH, USEAUTH
FROM SYSIBM.SEQUENCEAUTH
WHERE GRANTEE = 'USER1B';
```

GUPI

Adding and retrieving comments

After you create an object, you can provide explanatory information about it for future reference. For example, you can provide information about the purpose of the object, who uses it, and anything unusual about it.

GUPI

You can create comments about tables, views, indexes, aliases, packages, plans, distinct types, triggers, stored procedures, and user-defined functions. You can store a comment about the table or the view as a whole, and you can also include a comment for *each column*. A comment must not exceed 762 bytes.

A comment is especially useful if your names do not clearly indicate the contents of columns or tables. In that case, use a comment to describe the specific contents of the column or table.

Below are two examples of COMMENT:

```
COMMENT ON TABLE DSN8910.EMP IS
'Employee table. Each row in this table represents one
employee of the company.';
COMMENT ON COLUMN DSN8910.PROJ.PRSTDATE IS
'Estimated project start date. The format is DATE.';
```

After you execute a COMMENT statement, your comments are stored in the REMARKS column of SYSIBM.SYSTABLES or SYSIBM.SYSCOLUMNS. (Any

comment that is already present in the row is replaced by the new one.) The next two examples retrieve the comments that are added by the previous COMMENT statements.

```
SELECT REMARKS
  FROM SYSIBM.SYSTABLES
  WHERE NAME = 'EMP'
  AND CREATOR = 'DSN8910';

SELECT REMARKS
  FROM SYSIBM.SYSCOLUMNS
  WHERE NAME = 'PRSTDATE' AND TBNAME = 'PROJ'
  AND TBCREATOR = 'DSN8910';
```

GUIP

Verifying the accuracy of the database definition

You can use the catalog to verify the accuracy of your database definition.

GUIP

After you have created the objects in your database, display selected information from the catalog to check that no errors are in your CREATE statements. By examining the catalog tables, you can verify that your tables are in the correct table space, your table spaces are in the correct storage group, and so on.

GUIP

Sample user-defined functions

Some sample user-defined functions are provided with DB2. You can use the functions in your applications just as you would use other user-defined functions, or as examples to help you define your own user-defined functions..

- To use these functions in your applications: Use the functions only if installation job DSNTEJ2U, which prepares the functions for use, has been run. Because the external programs that implement the logic of the sample functions are written in C and C++, the installation job requires that your site has IBM C/C++ for OS/390. For information on installation job DSNTEJ2U, see *DB2 Installation Guide*.
- If you want to use these functions as examples to help you define and implement your own user-defined functions: Data set *prefix.SDSNSAMP* contains the code for the sample functions.

The following table lists the sample user-defined functions. The detailed descriptions of the functions include their external program names and specific names. The functions are in schema DSN8. The functions are defined to treat character or graphic string parameters, both input and output, as EBCDIC-encoded data.

Table 170. DB2 sample user-defined functions

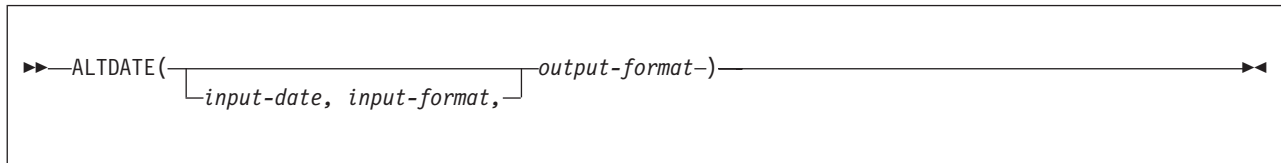
Function Name	Description
ALTDAT	Returns the current date or a user-specified date in a user-specified format
ALTTIME	Returns the current time or a user-specified time in a user-specified format
CURRENCY	Returns a floating-point number as a currency value

Table 170. DB2 sample user-defined functions (continued)

Function Name	Description
DAYNAME	Returns the name of the day of the week on which a date in ISO format falls
MONTHNAME	Returns the name of the month in which a date in ISO format falls
TABLE_LOCATION	Returns the location name of a table or view after resolving any aliases
TABLE_NAME	Returns the unqualified name of a table or view after resolving any aliases
TABLE_SCHEMA	Returns the schema name of a table or view after resolving any aliases
WEATHER	Shows how to use a user-defined table function to make non-relational data available for SQL manipulation

ALTDATE

The ALTDATE function returns the current date in the specified format or converts a user-specified date from one format to another.



The schema is DSN8.

The ALTDATE function returns the current date in one of the following formats or converts a user-specified date from one format to another:

D MONTH YY	D MONTH YYYY	DD MONTH YY	DD MONTH YYYY
D.M.YY	D.M.YYYY	DD.MM.YY	DD.MM.YYYY
D-M-YY	D-M-YYYY	DD-MM-YY	DD-MM-YYYY
D/M/YY	D/M/YYYY	DD/MM/YY	DD/MM/YYYY
M/D/YY	M/D/YYYY	MM/DD/YY	MM/DD/YYYY
YY/M/D	YYYY/M/D	YY/MM/DD	YYYY/MM/DD
YY.M.D	YYYY.M.D	YY.MM.DD	YYYY.MM.DD
	YYYY-M-D		YYYY-MM-DD
	YYYY-D-XX		YYYY-DD-XX
	YYYY-XX-D		YYYY-XX-DD

where:

- D: Suppress leading zero if the day is less than 10
- DD: Retain leading zero if the day is less than 10
- M: Suppress leading zero if the month is less than 10
- MM: Retain leading zero if the month is less than 10
- MONTH: Use English-language name of month
- XX: Use a capital Roman numeral for month
- YY: Use a year format without century
- YYYY: Use a year format with century

The ALTDATE function demonstrates how you can create an overloaded function—a function name for which there are multiple function instances. Each instance supports a different parameter list enabling you to group related but distinct functions in a single user-defined function. The ALTDATE function has two forms.

Form 1: ALTDATE(*output-format*)

This form of the function converts the current date into the specified format.

output-format

A character string that matches one of the 34 date formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 13 bytes.

The result of the function is VARCHAR(17).

Form 2: ALTDATE(*input-date*, *input-format*, *output-format*)

This form of the function converts a date (*input-date*) in one user-specified format (*input-format*) into another format (*output-format*).

input-date

The argument must be a date or a character string representation of a date

in the format specified by *input-format*. The character string must have a data type of VARCHAR and an actual length that is not greater than 17 bytes.

input-format

A character string that matches one of the 34 date formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 13 bytes.

output-format

A character string that matches one of the 34 date formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 13 bytes.

The result of the function is VARCHAR(17).

The following table shows the external and specific names for the two forms of the function, which are based on the input to the function.

Table 171. External program and specific names for ALTDAT

Conversion type	Input arguments	External name	Specific name
Current date	<i>output-format</i> (VARCHAR)	DSN8DUAD	DSN8.DSN8DUADV
User-specified date	<i>input-date</i> (VARCHAR) <i>input-format</i> (VARCHAR) <i>output-format</i> (VARCHAR)	DSN8DUCD	DSN8.DSN8DUCDVVV
	<i>input-date</i> (DATE) <i>input-format</i> (VARCHAR) <i>output-format</i> (VARCHAR)	DSN8DUCD	DSN8.DSN8DUCDDVV

Example 1: Convert the current date into format 'DD MONTH YY', a format that will include any leading zero for the month, the name of the month in English, and the year without the two digits for the century.

```
VALUES DSN8.ALTDAT( 'DD MONTH YY' );
```

Example 2: Convert the current date into format 'D.M.YYYY', a format that will suppress any leading zero for the day or month and include the year with the century.

```
VALUES DSN8.ALTDAT( 'D.M.YYYY' );
```

Example 3: Convert the current date into format 'YYYY-XX-DD', a format that will include the century, the month of the year as a roman numeral, and the day of the month with any leading zero.

```
VALUES DSN8.ALTDAT( 'YYYY-XX-DD' );
```

Example 4: Convert a date in the format of 'DD MONTH YYYY' to a date in the format of 'YYYY/MM/DD'.

```
VALUES DSN8.ALTDAT( '11 November 1918',
                    'DD MONTH YYYY',
                    'YYYY/MM/DD' );
```

The result of the above example is '1918/11/18'.

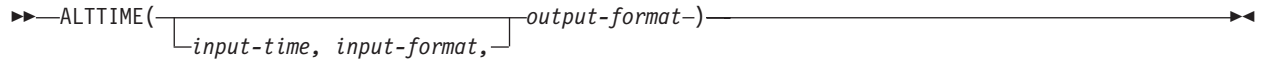
Example 5: Convert the date that employee 000130 was hired, a date in ISO format, into the format of 'D.M.YY'.

```
SELECT  FIRSTNME || ' '
        ||  LASTNAME || ' ' was hired on '
        ||  DSN8.ALTDAT( HIREDATE,
                        'YYYY-MM-DD',
                        'D.M.YY' )
FROM    EMP
WHERE   EMPNO  = '000130';
```

Assuming that the HIREDATE is '1971-07-28', the above example returns: 'DELORES QUINTANA was hired on 28.7.71'.

ALTTIME

The `ALTIME` function returns the current time in the specified format or converts a user-specified time from one format to another.



The schema is DSN8.

The `ALTTIME` function returns the current time in one of the following formats or converts a user-specified time from one of the formats to another:

H:MM AM/PM	HH:MM AM/PM
HH:MM:SS AM/PM	HH:MM:SS
H.MM	HH.MM
H.MM.SS	HH.MM.SS

where:

H: Suppress leading zero if the hour is less than 10

```
HH:      Retain leading zero if the hour is less than 10
```

M: Suppress leading zero if the minute is less than 10

MM: Retain leading zero if the minute is less than 10

AM/PM: Return time in 12-hour clock format, else 24-hour

The `ALTTIME` function demonstrates how you can create an overloaded function—a function name for which there are multiple function instances. Each instance supports a different parameter list enabling you to group related but distinct functions in a single user-defined function. The `ALTTIME` function has two forms.

Form 1: `ALTTIME(output-format)`

This form of the function converts the current time into the specified format.

output-format

A character string that matches one of the 8 time formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 14 bytes.

The result of the function is VARCHAR(11).

Form 2: `ALTTIME(input-time, input-format, output-format)`

This form of the function converts a time (*input-date*) in one user-specified format (*input-format*) into another format (*output-format*).

input-time

The argument must be a time or a character string representation of a time in the format specified by *input-format*. A character string argument must have a data type of VARCHAR and an actual length that is not greater than 11 bytes.

input-format

A character string that matches one of the 8 time formats that are shown above. The character string must have a data type of VARCHAR and an actual length that is not greater than 14 bytes.

output-format

A character string that matches one of the 8 time formats that are shown

above. The character string must have a data type of VARCHAR and an actual length that is not greater than 14 bytes.

The result of the function is VARCHAR(11).

The following table shows the external program and specific names for the two forms of the function, which are based on the input to the function.

Table 172. External and specific names for ALTTIME

Conversion type	Input arguments	External name	Specific name
Current time	<i>output-format</i> (VARCHAR)	DSN8DUAT	DSN8.DSN8DUATV
User-specified time	<i>input-time</i> (VARCHAR) <i>input-format</i> (VARCHAR) <i>output-format</i> (VARCHAR)	DSN8DUCT	DSN8.DSN8DUCTVVV
	<i>input-time</i> (TIME) <i>input-format</i> (VARCHAR) <i>output-format</i> (VARCHAR)	DSN8DUCT	DSN8.DSN8DUCTTVV

Example 1: Convert the current time into a 12-hour clock format without seconds, 'H.MM AM/PM'.

```
VALUES DSN8.ALTTIME( 'H:MM AM/PM' );
```

Example 2: Convert the current time into a 24-hour clock format without seconds, 'HH.MM'.

```
VALUES DSN8.ALTTIME( 'HH.MM' );
```

Example 3: Convert the current time into a 24-hour clock format with seconds, 'HH.MM.SS'.

```
VALUES DSN8.ALTTIME( 'HH.MM.SS' );
```

Example 4: Convert '00:00:00', a time in 24-hour clock format with seconds, to a time in 12-hour clock format without seconds.

```
VALUES DSN8.ALTTIME( '00:00:00', 'HH:MM:SS', 'HH:MM AM/PM' );
```

The function returns '12:00 AM'.

Example 5: Convert '00:00:00', a time in 24-hour clock format with seconds, to a time in 12-hour clock format without seconds and without any leading zero on the hour.

```
VALUES DSN8.ALTTIME( '06.42.37', 'HH.MM.SS', 'H:MM AM/PM' );
```

The function returns '6:42 AM'.

CURRENCY

The CURRENCY function returns a value that is formatted as an amount with a user-specified currency symbol and, if specified, one of three symbols that indicate debit or credit.

►►—CURRENCY(*-input-amount*, *currency-symbol* [, *credit/debit-indicator*])—►►

The schema is DSN8.

input-amount

An expression that specifies the value to be formatted. The expression must be a floating-point value.

currency-symbol

A character string that specifies the currency symbol. The string must have a data type of VARCHAR and an actual length that is not greater than 2 bytes.

credit/debit-indicator

A character string that specifies the symbol that is included with the result to indicate whether the value is negative or positive. The string must have a data type of VARCHAR and an actual length that is not greater than 5 bytes. If *credit/debit-indicator* is not specified or is the value null, the result is formatted without an indicator symbol. You can specify the following symbols:

CR/DB

Bank style. Negative input values are appended with 'DB'; positive input values are appended with 'CR'.

+/-

Arithmetic style. Negative input values are prefixed with a minus sign '-'; positive values are formatted without symbols.

(/)

Accounting style. Negative input values are enclosed in parentheses '('); positive values are formatted without symbols.

The result of the function is VARCHAR(19).

The CURRENCY function uses the C language functions strfmon to facilitate formatting of money amounts and setlocale to initialize strfmon for local conventions. If setlocale fails, the CURRENCY function returns an error.

The following table shows the external program and specific names for CURRENCY. The specific names differ depending on the input to the function.

Table 173. External program and specific names for CURRENCY

Input arguments	External name	Specific name
<i>input-amount</i> <i>currency-symbol</i>	DSN8DUCY	DSN8.DSN8DUCYFV
<i>input-amount</i> <i>currency-symbol</i> <i>credit/debit-indicator</i>	DSN8DUCY	DSN8.DSN8DUCYFVV

Example 1: Express[®] '-1234.56' as an amount in US dollars, using the bank style debit/credit indicator to indicate whether the value is negative or positive.

```
VALUES DSN8.CURRENCY( -1234.56, '$', 'CR/DB' );
```

The result of the function is '\$1,234.56 DB'.

Example 2: Express '-1234.56' as an amount in Deutsche marks, using the accounting style debit/credit indicator to indicate whether the value is negative or positive.

```
VALUES DSN8.CURRENCY( -1234.56, 'DM', '(/)' );
```

The result of the function is '(DM 1,234.56)'.

Example 3: Express '-1234.56' as an amount in Canadian dollars, using the accounting style debit/credit indicator to indicate whether the value is negative or positive.

```
VALUES DSN8.CURRENCY( -1234.56, 'CD', '+/-' );
```

The result of the function is '-CD 1,234.56'.

DAYNAME

The DAYNAME function returns the name of the weekday on which a given date falls. The name is returned in English.

►►—DAYNAME(*input-date*)—◄◄

The schema is DSN8.

input-date

A valid date or valid character string representation of a date. A character string representation The string must have a data type of VARCHAR and an actual length that is not greater than 10 bytes. The date must be in ISO format.

The result of the function is VARCHAR(9).

The DAYNAME function uses the IBM C++ class `IDate`.

The following table shows the external and specific names for DAYNAME. The specific names differ depending on the data type of the input argument.

Table 174. External and specific names for DAYNAME

Input arguments	External name	Specific name
<i>input-date</i> (VARCHAR)	DSN8EUDN	DSN8.DSN8EUDNV
<i>input-date</i> (DATE)	DSN8EUDN	DSN8.DSN8EUDND

Example 1: For the current date, find the day of the week.

```
VALUES DSN8.DAYNAME( CURRENT DATE );
```

Example 2: Find the day of the week on which leap year falls in the year 2008.

```
VALUES DSN8.DAYNAME( '2008-02-29' );
```

The result of the function is 'Friday'.

Example 3: Find the day of the week on which Delores Quintana, employee number 000130, was hired.

```
SELECT FIRSTNAME || ' ' ||  
       LASTNAME || ' was hired on '  
       DSN8.DAYNAME( HIREDATE ) || ', '  
       CHAR( HIREDATE )  
FROM EMP  
WHERE EMPNO = '000130';
```

The result of the function is 'DELORES QUINTANA was hired on Wednesday, 1971-07-28'.

MONTHNAME

The MONTHNAME function returns the calendar name of the month in which a given date falls. The name is returned in English.

►►—MONTHNAME(*input-date*)—◄◄

The schema is DSN8.

input-date

A valid date or valid character string representation of a date. A character string representation must have a data type of VARCHAR and an actual length that is no greater than 10 bytes. The date must be in ISO format.

The result of the function is VARCHAR(9).

The MONTHNAME function uses the IBM C++ class IDate.

The following table shows the external and specific names for MONTHNAME. The specific names differ depending on the data type of the input argument.

Table 175. External and specific names for MONTHNAME

Input arguments	External name	Specific name
<i>input-date</i> (VARCHAR)	DSN8EUMN	DSN8.DSN8EUMNV
<i>input-date</i> (DATE)	DSN8EUMN	DSN8.DSN8EUMND

Example 1: For the current date, find the name of the month.

```
VALUES DSN8.MONTHNAME( CURRENT DATE );
```

Example 2: Find the month of the year in which Delores Quintana, employee number 000130, was hired.

```
SELECT  FIRSTNME || ' ' ||  
        LASTNAME || ' was hired in the month of '  
        DSN8.MONTHNAME( HIREDATE )  
        CHAR( HIREDATE )  
FROM    EMP  
WHERE   EMPNO  = '000130';
```

The result of the function is 'DELORES QUINTANA was hired in the month of July'.

TABLE_LOCATION

The TABLE_LOCATION function searches for an object and returns the location name of the object after any alias chains have been resolved.

```
►►—TABLE_LOCATION(—object-name—, object-schema—, location-name—)►►
```

The schema is DSN8.

The starting point of the resolution is the object that is specified by *object-name* and, if specified, *object-schema* and *location-name*. If the starting point does not refer to an alias, the location name of the starting point is returned. The resulting name can be of a table, view, or undefined object. The function returns a blank if there is no location name.

object-name

A character expression that specifies the unqualified name to be resolved. The unqualified name is usually of an existing alias. *object-name* must have a data type of VARCHAR and an actual length that is no greater than 18 bytes.

object-schema

A character expression that represents the schema that is used to qualify the value specified in *object-name* before resolution. *object-schema* must have a data type of VARCHAR and an actual length that is no greater than 8 bytes.

If *object-schema* is not specified or is null, the default schema is used for the qualifier.

location-name

A character expression that represents the location that is used to qualify the value specified in *object-name* before resolution. *location-name* must have a data type of VARCHAR and an actual length that is no greater than 16 bytes.

If *location-name* is not specified or is null, the location name is equivalent to “any”.

The result of the function is VARCHAR(16). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value.

The following table shows the external and specific names for TABLE_LOCATION. The specific names differ depending on the number of input arguments to the function.

Table 176. External and specific names for TABLE_LOCATION

Input arguments	External name	Specific name
<i>object-name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILV
<i>object-name</i> (VARCHAR) <i>object-schema</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILVV

Table 176. External and specific names for TABLE_LOCATION (continued)

Input arguments	External name	Specific name
<i>object-name</i> (VARCHAR) <i>object-schema</i> (VARCHAR) <i>location-name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILVVV

Example: Assume that:

- DSN8.ALIAS_RS_SYSTABLES is an alias of SYSIBM.SYSTABLES at location name 'REMOTE_SITE'.
- The CURRENT SQLID is DSN8.

Use TABLE_LOCATION to find the location name where the base object for ALIAS_RS_SYSTABLES resides.

```
VALUES DSN8.TABLE_LOCATION( 'ALIAS_RS_SYSTABLES' );
```

The result of the function is 'REMOTE_SITE'.

TABLE_NAME

The TABLE_NAME function searches for an object and returns the unqualified name of the object after any alias chains have been resolved.

```
►►—TABLE_NAME(—object-name—, —object-schema—, —location-name—)►►
```

The schema is DSN8.

The starting point of the resolution is the object that is specified by *object-name* and, if specified, *object-schema* and *location name*. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name can be of a table, view, or undefined object.

object-name

A character expression that specifies the unqualified name to be resolved. The unqualified name is usually of an existing alias. *object-name* must have a data type of VARCHAR and an actual length that is no greater than 18 bytes.

object-schema

A character expression that represents the schema that is used to qualify the value specified in *object-name* before resolution. *object-schema* must have a data type of VARCHAR and an actual length that is no greater than 8 bytes.

If *object-schema* is not specified or is null, the default schema is used for the qualifier.

location-name

A character expression that represents the location that is used to qualify the value specified in *object-name* before resolution. *location-name* must have a data type of VARCHAR and an actual length that is no greater than 16 bytes.

If *location-name* is not specified or is null, the location name is equivalent to "any".

The result of the function is VARCHAR(128). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value.

The following table shows the external and specific names for TABLE_NAME. The specific names differ depending on the number of input arguments to the function.

Table 177. External and specific names for TABLE_NAME

Input arguments	External name	Specific name
<i>object-name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINV
<i>object-name</i> (VARCHAR) <i>object-schema</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINVV

Table 177. External and specific names for TABLE_NAME (continued)

Input arguments	External name	Specific name
<i>object-name</i> (VARCHAR) <i>object-schema</i> (VARCHAR) <i>location-name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINVVV

Example: Assume that:

- DSN8.VIEW_OF_SYSTABLES is a view of SYSIBM.SYSTABLES.
- DSN8.ALIAS_OF_VIEW is an alias of DSN8.VIEW_OF_SYSTABLES.
- The CURRENT SQLID is DSN8.

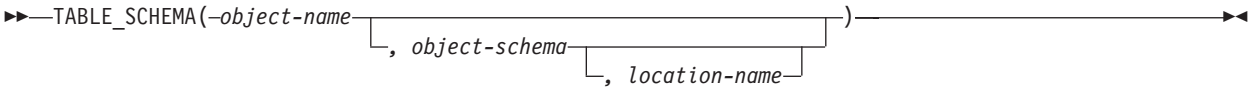
Use TABLE_NAME to find the name of the base object for ALIAS_OF_VIEW.

```
VALUES DSN8.TABLE_NAME( 'ALIAS_OF_VIEW' );
```

The result of the function is 'VIEW_OF_SYSTABLES'.

TABLE_SCHEMA

The TABLE_SCHEMA function searches for an object and returns the schema name of the object after any synonyms or alias chains have been resolved.



The schema is DSN8.

The starting point of the resolution is the object that is specified by *objectname* and *objectschema*. If the starting point does not refer to an alias or synonym, the schema name of the starting point is returned. The resulting schema name can be of a table, view, or undefined object.

object-name

A character expression that specifies the unqualified name to be resolved. The unqualified name is usually of an existing alias. *object-name* must have a data type of VARCHAR and an actual length that is no greater than 18 bytes.

object-schema

A character expression that represents the schema that is used to qualify the value specified in *object-name* before resolution. *object-schema* must have a data type of VARCHAR and an actual length that is no greater than 8 bytes.

If *object-schema* is not specified or is null, the default schema is used for the qualifier.

location-name

A character expression that represents the location that is used to qualify the value specified in *object-name* before resolution. *location-name* must have a data type of VARCHAR (and an actual length that is no greater than 16 bytes).

If *location-name* is not specified or is null, the location name is equivalent to “any”.

The result of the function is VARCHAR(128). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value.

The following table shows the external and specific names for TABLE_SCHEMA. The specific names differ depending on the number of input arguments.

Table 178. External and specific names for function TABLE_SCHEMA

Input arguments	External name	Specific name
<i>object-name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTISV
<i>object-name</i> (VARCHAR) <i>object-schema</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTISVV

Table 178. External and specific names for function TABLE_SCHEMA (continued)

Input arguments	External name	Specific name
<i>object-name</i> (VARCHAR) <i>object-schema</i> (VARCHAR) <i>location-name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTISVVV

Example: Assume that:

- DSN8.ALIAS_OF_SYSTABLES is an alias of SYSIBM.SYSTABLES.
- The CURRENT SQLID is DSN8.

Find the name of the schema of the base table for ALIAS_OF_SYSTABLES.

```
VALUES DSN8.TABLE_SCHEMA( 'ALIAS_OF_SYSTABLES' );
```

The result of the function is 'SYSIBM'.

WEATHER

The WEATHER function returns information from a TSO data set as a DB2 table. The TSO data set contains sample weather statistics for various cities in the United States. The statistics are returned to the client with a row for each city and a column for each statistic. The WEATHER function is provided primarily to help you design and implement table functions.

```
►►WEATHER(input-data-set-name)—RETURNS TABLE(

|                                  |
|----------------------------------|
| <i>name-of-city</i>              |
| <i>temperature-in-fahrenheit</i> |
| <i>percent-humidity</i>          |
| <i>wind-direction</i>            |
| <i>wind-velocity</i>             |
| <i>barometer</i>                 |
| <i>forecast</i>                  |

)—►►
```

The schema is DSN8.

Unlike the other sample user-defined functions, which are scalar functions, WEATHER is a table function. WEATHER shows how to use a table function to make non-relational data available to a client for manipulation by SQL.

input-data-set-name

The name of the TSO data set that contains sample weather statistics. The name is a character string with a data type of VARCHAR and an actual length that is not greater than 44 bytes.

The result of the function is a DB2 table with the following columns. Each column can be null.

<i>name-of-city</i>	VARCHAR(30)
<i>temperature-in-fahrenheit</i>	INTEGER
<i>percent-humidity</i>	INTEGER
<i>wind-direction</i>	VARCHAR(5)
<i>wind-velocity</i>	INTEGER
<i>barometer</i>	FLOAT
<i>forecast</i>	VARCHAR(25)

The external program name for the function is DSN8DUWF, and the specific name is DSN8.DSN8DUWF.

Example: Find the name of and the forecast for the cities that have a temperature less than 25 degrees.

```
SELECT CITY, FORECAST
FROM TABLE(DSN8.WEATHER('prefix.SDSNIVPD(DSN8LWC)')) AS W
WHERE TEMP_IN_F < 25
ORDER BY CITY;
```

This example returns:

Bessemer, MI	Slight chance of snow
Cheyenne, WY	Continued cooling
Helena, MT	Heavy snow
Pierre, SD	Continued cold

Information resources for DB2 for z/OS and related products

Many information resources are available to help you use DB2 for z/OS and many related products. A large amount of technical information about IBM products is now available online in information centers or on library websites.

Disclaimer: Any web addresses that are included here are accurate at the time this information is being published. However, web addresses sometimes change. If you visit a web address that is listed here but that is no longer valid, you can try to find the current web address for the product information that you are looking for at either of the following sites:

- <http://www.ibm.com/support/publications/us/library/index.shtml>, which lists the IBM information centers that are available for various IBM products
- <http://www.ibm.com/shop/publications/order>, which is the IBM Publications Center, where you can download online PDF books or order printed books for various IBM products

DB2 for z/OS product information

The primary place to find and use information about DB2 for z/OS is the Information Management Software for z/OS Solutions Information Center (<http://publib.boulder.ibm.com/infocenter/imzic>), which also contains information about IMS, QMF, and many DB2 and IMS Tools products. This information center is also available as an installable information center that can run on a local system or on an intranet server. You can order the Information Management for z/OS Solutions Information Center DVD (SK5T-7377) for a low cost from the IBM Publications Center (<http://www.ibm.com/shop/publications/order>).

The majority of the DB2 for z/OS information in this information center is also available in the books that are identified in the following table. You can access these books at the DB2 for z/OS library website (<http://www.ibm.com/software/data/db2/zos/library.html>) or at the IBM Publications Center (<http://www.ibm.com/shop/publications/order>).

Table 179. DB2 Version 9.1 for z/OS book titles

Title	Publication number	Available in information center	Available in PDF	Available in BookManager® format	Available in printed book
<i>DB2 Version 9.1 for z/OS Administration Guide</i>	SC18-9840	X	X	X	X
<i>DB2 Version 9.1 for z/OS Application Programming & SQL Guide</i>	SC18-9841	X	X	X	X
<i>DB2 Version 9.1 for z/OS Application Programming Guide and Reference for Java</i>	SC18-9842	X	X	X	X
<i>DB2 Version 9.1 for z/OS Codes</i>	GC18-9843	X	X	X	X
<i>DB2 Version 9.1 for z/OS Command Reference</i>	SC18-9844	X	X	X	X
<i>DB2 Version 9.1 for z/OS Data Sharing: Planning and Administration</i>	SC18-9845	X	X	X	X

Table 179. DB2 Version 9.1 for z/OS book titles (continued)

Title	Publication number	Available in information center	Available in PDF	Available in BookManager® format	Available in printed book
<i>DB2 Version 9.1 for z/OS Diagnosis Guide and Reference</i> ¹	LY37-3218		X	X	X
<i>DB2 Version 9.1 for z/OS Diagnostic Quick Reference</i>	LY37-3219				X
<i>DB2 Version 9.1 for z/OS Installation Guide</i>	GC18-9846	X	X	X	X
<i>DB2 Version 9.1 for z/OS Introduction to DB2</i>	SC18-9847	X	X	X	X
<i>DB2 Version 9.1 for z/OS Licensed Program Specifications</i>	GC18-9848		X		X
<i>DB2 Version 9.1 for z/OS Messages</i>	GC18-9849	X	X	X	X
<i>DB2 Version 9.1 for z/OS ODBC Guide and Reference</i>	SC18-9850	X	X	X	X
<i>DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide</i>	SC18-9851	X	X	X	X
<i>DB2 Version 9.1 for z/OS Optimization Service Center</i>		X			
<i>DB2 Version 9.1 for z/OS Program Directory</i>	GI10-8737		X		X
<i>DB2 Version 9.1 for z/OS RACF Access Control Module Guide</i>	SC18-9852	X	X	X	
<i>DB2 Version 9.1 for z/OS Reference for Remote DRDA Requesters and Servers</i>	SC18-9853	X	X	X	
<i>DB2 Version 9.1 for z/OS Reference Summary</i>	SX26-3854		X		
<i>DB2 Version 9.1 for z/OS SQL Reference</i>	SC18-9854	X	X	X	X
<i>DB2 Version 9.1 for z/OS Utility Guide and Reference</i>	SC18-9855	X	X	X	X
<i>DB2 Version 9.1 for z/OS What's New?</i>	GC18-9856	X	X		X
<i>DB2 Version 9.1 for z/OS XML Extender Administration and Programming</i>	SC18-9857	X	X	X	X
<i>DB2 Version 9.1 for z/OS XML Guide</i>	SC18-9858	X	X	X	X
<i>IRLM Messages and Codes for IMS and DB2 for z/OS</i>	GC19-2666	X	X	X	

Note:

1. *DB2 Version 9.1 for z/OS Diagnosis Guide and Reference* is available in PDF and BookManager formats on the DB2 Version 9.1 for z/OS Licensed Collection kit, LK3T-7195. You can order this License Collection kit on the IBM Publications Center site (<http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss>). This book is also available in online format in DB2 data set DSN910.SDSNIVPD(DSNDR).

Information resources for related products

In the following table, related product names are listed in alphabetic order, and the associated web addresses of product information centers or library web pages are indicated.

Table 180. Related product information resource locations

Related product	Information resources
C/C++ for z/OS	Library website: http://www.ibm.com/software/awdtools/czos/library/ This product is now called z/OS XL C/C++.
CICS Transaction Server for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp
COBOL	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp This product is now called Enterprise COBOL for z/OS.
DB2 Connect	Information center: http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp This resource is for DB2 Connect 9.
DB2 Database for Linux, UNIX, and Windows	Information center: http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp This resource is for DB2 9 for Linux, UNIX, and Windows.
DB2 Query Management Facility™	Information center: http://publib.boulder.ibm.com/infocenter/imzic
DB2 Server for VSE & VM	Product website: http://www.ibm.com/software/data/db2/vse-vm/
DB2 Tools	One of the following locations: <ul style="list-style-type: none"> • Information center: http://publib.boulder.ibm.com/infocenter/imzic • Library website: http://www.ibm.com/software/data/db2imstools/library.html <p>These resources include information about the following products and others:</p> <ul style="list-style-type: none"> • DB2 Administration Tool • DB2 Automation Tool • DB2 Log Analysis Tool • DB2 Object Restore Tool • DB2 Query Management Facility • DB2 SQL Performance Analyzer
DB2 Universal Database for iSeries®	Information center: http://www.ibm.com/systems/i/infocenter/
Debug Tool for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp
Enterprise COBOL for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp
Enterprise PL/I for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp
InfoSphere™ Replication Server for z/OS	Information center: http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.swg.im.iis.db.prod.repl.nav.doc/dochome/iiryrcnav_dochome.html This product was also known as DB2 DataPropagator, DB2 Information Integrator Replication Edition for z/OS, and WebSphere Replication Server for z/OS.
IMS	Information center: http://publib.boulder.ibm.com/infocenter/imzic

Table 180. Related product information resource locations (continued)

Related product	Information resources
IMS Tools	<p>One of the following locations:</p> <ul style="list-style-type: none"> • Information center: http://publib.boulder.ibm.com/infocenter/imzic • Library website: http://www.ibm.com/software/data/db2imstools/library.html <p>These resources have information about the following products and others:</p> <ul style="list-style-type: none"> • IMS Batch Terminal Simulator for z/OS • IMS Connect • IMS HALDB Conversion and Maintenance Aid • IMS High Performance Utility products • IMS DataPropagator • IMS Online Reorganization Facility • IMS Performance Analyzer
Integrated Data Management products	<p>Information center: http://publib.boulder.ibm.com/infocenter/idm/v2r2/index.jsp</p> <p>This information center has information about the following products and others:</p> <ul style="list-style-type: none"> • IBM Data Studio • InfoSphere Data Architect • InfoSphere Warehouse • Optim Database Administrator • Optim Development Studio • Optim Query Tuner
PL/I	<p>Information center: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp</p> <p>This product is now called Enterprise PL/I for z/OS.</p>
System z	http://publib.boulder.ibm.com/infocenter/eserver/v1r2/index.jsp
Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS	<p>Information center: http://publib.boulder.ibm.com/infocenter/tivihelp/v15r1/topic/com.ibm.omegamon.xe_db2.doc/ko2welcome_pe.htm</p> <p>In earlier releases, this product was called DB2 Performance Expert for z/OS.</p>
WebSphere Application Server	Information center: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp
WebSphere Message Broker with Rules and Formatter Extension	<p>Information center: http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp</p> <p>The product is also known as WebSphere MQ Integrator Broker.</p>
WebSphere MQ	<p>Information center: http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp</p> <p>The resource includes information about MQSeries.</p>
z/Architecture®	Library Center site: http://www.ibm.com/servers/eserver/zseries/zos/bkserv/

Table 180. Related product information resource locations (continued)

Related product	Information resources
z/OS	<p>Library Center site: http://www.ibm.com/servers/eserver/zseries/zos/bkserv/</p> <p>This resource includes information about the following z/OS elements and components:</p> <ul style="list-style-type: none"> • Character Data Representation Architecture • Device Support Facilities • DFSORT • Fortran • High Level Assembler • NetView[®] • SMP/E for z/OS • SNA • TCP/IP • TotalStorage Enterprise Storage Server[®] • VTAM • z/OS C/C++ • z/OS Communications Server • z/OS DCE • z/OS DFSMS • z/OS DFSMS Access Method Services • z/OS DFSMSdss • z/OS DFSMSHsm • z/OS DFSMSdftp • z/OS ICSF • z/OS ISPF • z/OS JES3 • z/OS Language Environment • z/OS Managed System Infrastructure • z/OS MVS • z/OS MVS JCL • z/OS Parallel Sysplex • z/OS RMF[™] • z/OS Security Server • z/OS UNIX System Services
z/OS XL C/C++	<p>http://www.ibm.com/software/awdtools/czos/library/</p>

The following information resources from IBM are not necessarily specific to a single product:

- The DB2 for z/OS Information Roadmap; available at: <http://www.ibm.com/software/data/db2/zos/roadmap.html>
- DB2 Redbooks[®] and Redbooks about related products; available at: <http://www.ibm.com/redbooks>
- IBM Educational resources:
 - Information about IBM educational offerings is available on the web at: <http://www.ibm.com/software/sw-training/>

- A collection of glossaries of IBM terms in multiple languages is available on the IBM Terminology website at: <http://www.ibm.com/software/globalization/terminology/index.jsp>
- National Language Support information; available at the IBM Publications Center at: <http://www.elink.ibm.link.ibm.com/public/applications/publications/cgi-bin/pbi.cgi>
- *SQL Reference for Cross-Platform Development*; available at the following developerWorks® site: <http://www.ibm.com/developerworks/db2/library/techarticle/0206sqlref/0206sqlref.html>

The following information resources are not published by IBM but can be useful to users of DB2 for z/OS and related products:

- Database design topics:
 - *DB2 for z/OS and OS/390 Development for Performance Volume I*, by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-605-2
 - *DB2 for z/OS and OS/390 Development for Performance Volume II*, by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-606-0
 - *Handbook of Relational Database Design*, by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8
- Distributed Relational Database Architecture (DRDA) specifications; <http://www.opengroup.org>
- Domain Name System: *DNS and BIND*, Third Edition, Paul Albitz and Cricket Liu, O'Reilly, ISBN 0-59600-158-4
- Microsoft Open Database Connectivity (ODBC) information; <http://msdn.microsoft.com/library/>
- Unicode information; <http://www.unicode.org>

How to obtain DB2 information

You can access the official information about the DB2 product in a number of ways.

- “DB2 on the web”
- “DB2 product information”
- “DB2 education” on page 2034
- “How to order the DB2 library” on page 2034

DB2 on the web

Stay current with the latest information about DB2 by visiting the DB2 home page on the web:

<http://www.ibm.com/software/db2zos>

On the DB2 home page, you can find links to a wide variety of information resources about DB2. You can read news items that keep you informed about the latest enhancements to the product. Product announcements, press releases, fact sheets, and technical articles help you plan and implement your database management strategy.

DB2 product information

The official DB2 for z/OS information is available in various formats and delivery methods. IBM provides mid-version updates to the information in the information center and in softcopy updates that are available on the web and on CD-ROM.

Information Management Software for z/OS Solutions Information Center

DB2 product information is viewable in the information center, which is the primary delivery vehicle for information about DB2 for z/OS, IMS, QMF, and related tools. This information center enables you to search across related product information in multiple languages for data management solutions for the z/OS environment and print individual topics or sets of related topics. You can also access, download, and print PDFs of the publications that are associated with the information center topics. Product technical information is provided in a format that offers more options and tools for accessing, integrating, and customizing information resources. The information center is based on Eclipse open source technology.

The Information Management Software for z/OS Solutions Information Center is viewable at the following website:

<http://publib.boulder.ibm.com/infocenter/imzic>

CD-ROMs and DVD

Books for DB2 are available on a CD-ROM that is included with your product shipment:

- DB2 V9.1 for z/OS Licensed Library Collection, LK3T-7195, in English

The CD-ROM contains the collection of books for DB2 V9.1 for z/OS in PDF and BookManager formats. Periodically, IBM refreshes the books on subsequent editions of this CD-ROM.

The books for DB2 for z/OS are also available on the following DVD collection kit, which contains online books for many IBM products:

- IBM z/OS Software Products DVD Collection, SK3T-4271, in English

PDF format

Many of the DB2 books are available in PDF (Portable Document Format) for viewing or printing from CD-ROM or the DB2 home page on the web or from the information center. Download the PDF books to your intranet for distribution throughout your enterprise.

BookManager format

You can use online books on CD-ROM to read, search across books, print portions of the text, and make notes in these BookManager books. Using the IBM Softcopy Reader, appropriate IBM Library Readers, or the BookManager Read product, you can view these books in the z/OS, Windows, and VM environments. You can also view and search many of the DB2 BookManager books on the web.

DB2 education

IBM Education and Training offers a wide variety of classroom courses to help you quickly and efficiently gain DB2 expertise. IBM schedules classes in cities all over the world. You can find class information, by country, at the IBM Learning Services website:

<http://www.ibm.com/services/learning>

IBM also offers classes at your location, at a time that suits your needs. IBM can customize courses to meet your exact requirements. For more information, including the current local schedule, contact your IBM representative.

How to order the DB2 library

To order books, visit the IBM Publication Center on the web:

<http://www.ibm.com/shop/publications/order>

From the IBM Publication Center, you can go to the Publication Notification System (PNS). PNS users receive electronic notifications of updated publications in their profiles. You have the option of ordering the updates by using the publications direct ordering application or any other IBM publication ordering channel. The PNS application does not send automatic shipments of publications. You will receive updated publications and a bill for them if you respond to the electronic notification.

You can also order DB2 publications and CD-ROMs from your IBM representative or the IBM branch office that serves your locality. If your location is within the United States or Canada, you can place your order by calling one of the toll-free numbers:

- In the U.S., call 1-800-879-2755.
- In Canada, call 1-800-426-4968.

To order additional copies of licensed publications, specify the SOFTWARE option. To order additional publications or CD-ROMs, specify the PUBLICATIONS option. Be prepared to give your customer number, the product number, and either the feature codes or order numbers that you want.

How to use the DB2 library

Titles of books in the library begin with DB2 Version 9.1 for z/OS. However, references from one book in the library to another are shortened and do not include the product name, version, and release. Instead, they point directly to the section that holds the information. The primary place to find and use information about DB2 for z/OS is the Information Management Software for z/OS Solutions Information Center (<http://publib.boulder.ibm.com/infocenter/imzic>).

If you are new to DB2 for z/OS, *Introduction to DB2 for z/OS* provides a comprehensive introduction to DB2 Version 9.1 for z/OS. Topics included in this book explain the basic concepts that are associated with relational database management systems in general, and with DB2 for z/OS in particular.

The most rewarding task associated with a database management system is asking questions of it and getting answers, the task called *end use*. Other tasks are also necessary—defining the parameters of the system, putting the data in place, and so on. The tasks that are associated with DB2 are grouped into the following major categories.

Installation

If you are involved with installing DB2, you will need to use a variety of resources, such as:

- *DB2 Program Directory*
- *DB2 Installation Guide*
- *DB2 Administration Guide*
- *DB2 Application Programming Guide and Reference for Java*
- *DB2 Codes*
- *DB2 Internationalization Guide*
- *DB2 Messages*
- *DB2 Performance Monitoring and Tuning Guide*
- *DB2 RACF Access Control Module Guide*
- *DB2 Utility Guide and Reference*

If you will be using data sharing capabilities you also need *DB2 Data Sharing: Planning and Administration*, which describes installation considerations for data sharing.

If you will be installing and configuring DB2 ODBC, you will need *DB2 ODBC Guide and Reference*.

If you are installing IBM Spatial Support for DB2 for z/OS, you will need *IBM Spatial Support for DB2 for z/OS User's Guide and Reference*.

If you are installing IBM OmniFind® Text Search Server for DB2 for z/OS, you will need *IBM OmniFind Text Search Server for DB2 for z/OS Installation, Administration, and Reference*.

End use

End users issue SQL statements to retrieve data. They can also insert, update, or delete data, with SQL statements. They might need an introduction to SQL, detailed instructions for using SPUI, and an alphabetized reference to the types of SQL statements. This information is found in *DB2 Application Programming and SQL Guide*, and *DB2 SQL Reference*.

End users can also issue SQL statements through the DB2 Query Management Facility (QMF) or some other program, and the library for that licensed program might provide all the instruction or reference material they need.

Application programming

Some users access DB2 without knowing it, using programs that contain SQL statements. DB2 application programmers write those programs. Because they write SQL statements, they need the same resources that end users do.

Application programmers also need instructions for many other topics:

- How to transfer data between DB2 and a host program—written in Java, C, or COBOL, for example
- How to prepare to compile a program that embeds SQL statements
- How to process data from two systems simultaneously, for example, DB2 and IMS or DB2 and CICS
- How to write distributed applications across operating systems
- How to write applications that use Open Database Connectivity (ODBC) to access DB2 servers
- How to write applications that use JDBC and SQLJ with the Java programming language to access DB2 servers
- How to write applications to store XML data on DB2 servers and retrieve XML data from DB2 servers.

The material needed for writing a host program containing SQL is in *DB2 Application Programming and SQL Guide*.

The material needed for writing applications that use JDBC and SQLJ to access DB2 servers is in *DB2 Application Programming Guide and Reference for Java*. The material needed for writing applications that use DB2 CLI or ODBC to access DB2 servers is in *DB2 ODBC Guide and Reference*. The material needed for working with XML data in DB2 is in *DB2 XML Guide*. For handling errors, see *DB2 Messages* and *DB2 Codes*.

If you are a software vendor implementing DRDA clients and servers, you will need *DB2 Reference for Remote DRDA Requesters and Servers*.

Information about writing applications across operating systems can be found in *IBM DB2 SQL Reference for Cross-Platform Development*.

System and database administration

Administration covers almost everything else. *DB2 Administration Guide* divides some of those tasks among the following sections:

- **Designing a database:** Discusses the decisions that must be made when designing a database and tells how to implement the design by creating and altering DB2 objects, loading data, and adjusting to changes.
- **Security and auditing:** Describes ways of controlling access to the DB2 system and to data within DB2, to audit aspects of DB2 usage, and to answer other security and auditing concerns.
- **Operation and recovery:** Describes the steps in normal day-to-day operation and discusses the steps one should take to prepare for recovery in the event of some failure.

DB2 Performance Monitoring and Tuning Guide explains how to monitor the performance of the DB2 system and its parts. It also lists things that can be done to make some parts run faster.

If you will be using the RACF access control module for DB2 authorization checking, you will need *DB2 RACF Access Control Module Guide*.

If you are involved with DB2 only to design the database, or plan operational procedures, you need *DB2 Administration Guide*. If you also want to carry out your own plans by creating DB2 objects, granting privileges, running utility jobs, and so on, you also need:

- *DB2 SQL Reference*, which describes the SQL statements you use to create, alter, and drop objects and grant and revoke privileges
- *DB2 Utility Guide and Reference*, which explains how to run utilities
- *DB2 Command Reference*, which explains how to run commands

If you will be using data sharing, you need *DB2 Data Sharing: Planning and Administration*, which describes how to plan for and implement data sharing.

Additional information about system and database administration can be found in *DB2 Messages* and *DB2 Codes*, which list messages and codes issued by DB2, with explanations and suggested responses.

Diagnosis

Diagnosticians detect and describe errors in the DB2 program. They might also recommend or apply a remedy. The documentation for this task is in *DB2 Diagnosis Guide and Reference*, *DB2 Messages*, and *DB2 Codes*.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.


Programming Interface Information

This information is intended to help you to code SQL statements. This information primarily documents General-use Programming Interface and Associated Guidance Information provided by DB2 Version 9.1 for z/OS. This information also documents Product-sensitive Programming Interface and Associated Guidance Information provided by DB2 Version 9.1 for z/OS.

General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of DB2 Version 9.1 for z/OS.

General-use Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:


 General-use Programming Interface and Associated Guidance Information...



Product-sensitive Programming Interface and Associated Guidance Information

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:

 Product-sensitive Programming Interface and Associated Guidance Information... 

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com/legal/copytrade.shtml)[®] are trademarks or registered marks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at <http://www.ibm.com/legal/copytrade.shtml>.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Glossary

abend See abnormal end of task.

abend reason code

A 4-byte hexadecimal code that uniquely identifies a problem with DB2.

abnormal end of task (abend)

Termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve during execution.

access method services

The facility that is used to define, alter, delete, print, and reproduce VSAM key-sequenced data sets.

access path

The path that is used to locate data that is specified in SQL statements. An access path can be indexed or sequential.

access path stability

A characteristic of an access path that defines reliability for dynamic or static queries. Access paths are not regenerated unless there is a schema change or manual intervention.

active log

The portion of the DB2 log to which log records are written as they are generated. The active log always contains the most recent log records. See also archive log.

address space

A range of virtual storage pages that is identified by a number (ASID) and a collection of segment and page tables that map the virtual pages to real pages of the computer's memory.

address space connection

The result of connecting an allied address space to DB2. See also allied address space and task control block.

address space identifier (ASID)

A unique system-assigned identifier for an address space.

AFTER trigger

A trigger that is specified to be activated after a defined trigger event (an insert, update, or delete operation on the table that is specified in a trigger definition).

Contrast with BEFORE trigger and INSTEAD OF trigger.

agent In DB2, the structure that associates all processes that are involved in a DB2 unit of work. See also allied agent and system agent.

aggregate function

An operation that derives its result by using values from one or more rows. Contrast with scalar function.

alias

An alternative name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem. An alias can be qualified with a schema qualifier and can thereby be referenced by other users. Contrast with synonym.

allied address space

An area of storage that is external to DB2 and that is connected to DB2. An allied address space can request DB2 services. See also address space.

allied agent

An agent that represents work requests that originate in allied address spaces. See also system agent.

allied thread

A thread that originates at the local DB2 subsystem and that can access data at a remote DB2 subsystem.

allocated cursor

A cursor that is defined for a stored procedure result set by using the SQL ALLOCATE CURSOR statement.

ambiguous cursor

A database cursor for which DB2 cannot determine whether it is used for update or read-only purposes.

APAR See authorized program analysis report.

APF See authorized program facility.

API See application programming interface.

APPL A VTAM network definition statement that is used to define DB2 to VTAM as an application program that uses SNA LU 6.2 protocols.

application

A program or set of programs that performs a task; for example, a payroll application.

application period

A pair of columns with application-maintained values that indicate the period of time when a row is valid.

application-period temporal table

A table that includes an application period.

application plan

The control structure that is produced during the bind process. DB2 uses the application plan to process SQL statements that it encounters during statement execution.

application process

The unit to which resources and locks are allocated. An application process involves the execution of one or more programs.

application programming interface (API)

A functional interface that is supplied by the operating system or by a separately ordered licensed program that allows an application program that is written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester

The component on a remote system that generates DRDA requests for data on behalf of an application.

application server

The target of a request from a remote application. In the DB2 environment, the application server function is provided by the distributed data facility and is used to access DB2 data from remote applications.

archive log

The portion of the DB2 log that contains log records that have been copied from the active log. See also active log.

ASCII An encoding scheme that is used to represent strings in many environments, typically on personal computers and workstations. Contrast with EBCDIC and Unicode.

ASID See address space identifier.

attachment facility

An interface between DB2 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2.

attribute

A characteristic of an entity. For example, in database design, the phone number of an employee is an attribute of that employee.

authorization ID

A string that can be verified for connection to DB2 and to which a set of privileges is allowed. An authorization ID can represent an individual or an organizational group.

authorized program analysis report (APAR)

A report of a problem that is caused by a suspected defect in a current release of an IBM supplied program.

authorized program facility (APF)

A facility that allows an installation to identify system or user programs that can use sensitive system functions.

automatic bind

(More correctly *automatic rebind*.) A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.

automatic query rewrite

A process that examines an SQL statement that refers to one or more base tables or materialized query tables, and, if appropriate, rewrites the query so that it performs better.

auxiliary index

An index on an auxiliary table in which each index entry refers to a LOB or XML document.

auxiliary table

A table that contains columns outside the actual table in which they are defined. Auxiliary tables can contain either LOB or XML data.

backout

The process of undoing uncommitted changes that an application process made. A backout is often performed in the event

of a failure on the part of an application process, or as a result of a deadlock situation.

backward log recovery

The final phase of restart processing during which DB2 scans the log in a backward direction to apply UNDO log records for all aborted changes.

base table

A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with clone table, materialized query table, result table, temporary table, and transition table.

base table space

A table space that contains base tables.

basic row format

A row format in which values for columns are stored in the row in the order in which the columns are defined by the CREATE TABLE statement. Contrast with reordered row format.

basic sequential access method (BSAM)

An access method for storing or retrieving data blocks in a continuous sequence, using either a sequential-access or a direct-access device.

BEFORE trigger

A trigger that is specified to be activated before a defined trigger event (an insert, an update, or a delete operation on the table that is specified in a trigger definition). Contrast with AFTER trigger and INSTEAD OF trigger.

begin column

In a system period or an application period, the column that indicates the beginning of the period.

binary large object (BLOB)

A binary string data type that contains a sequence of bytes that can range in size from 0 bytes to 2 GB, less 1 byte. This string does not have an associated code page and character set. BLOBs can contain, for example, image, audio, or video data. In general, BLOB values are used whenever a binary string might exceed the limits of the VARBINARY type.

binary string

A sequence of bytes that is not associated

with a CCSID. Binary string data type can be further classified as BINARY, VARBINARY, or BLOB.

binary XML format

A system of storing XML data in binary, as opposed to text, that facilitates more efficient storage and exchange.

bind

A process by which a usable control structure with SQL statements is generated; the structure is often called an access plan, an application plan, or a package. During this bind process, access paths to the data are selected, and some authorization checking is performed. See also automatic bind.

bit data

- Data with character type CHAR or VARCHAR that is defined with the FOR BIT DATA clause. Note that using BINARY or VARBINARY rather than FOR BIT DATA is highly recommended.
- Data with character type CHAR or VARCHAR that is defined with the FOR BIT DATA clause.
- A form of character data. Binary data is generally more highly recommended than character-for-bit data.

bitemporal table

A table that is both a system-period temporal table and an application-period temporal table.

BLOB See binary large object.

block fetch

A capability in which DB2 can retrieve, or fetch, a large set of rows together. Using block fetch can significantly reduce the number of messages that are being sent across the network. Block fetch applies only to non-rowset cursors that do not update data.

bootstrap data set (BSDS)

A VSAM data set that contains name and status information for DB2 and RBA range specifications for all active and archive log data sets. The BSDS also contains passwords for the DB2 directory and catalog, and lists of conditional restart and checkpoint records.

BSAM

See basic sequential access method.

BSDS See bootstrap data set.

buffer pool

An area of memory into which data pages are read, modified, and held during processing.

built-in data type

A data type that IBM supplies. Among the built-in data types for DB2 for z/OS are string, numeric, XML, ROWID, and datetime. Contrast with distinct type.

built-in function

A function that is generated by DB2 and that is in the SYSIBM schema. Contrast with user-defined function. See also function, cast function, external function, sourced function, and SQL function.

business dimension

A category of data, such as products or time periods, that an organization might want to analyze.

cache structure

A coupling facility structure that stores data that can be available to all members of a Sysplex. A DB2 data sharing group uses cache structures as group buffer pools.

CAF See call attachment facility.

call attachment facility (CAF)

A DB2 attachment facility for application programs that run in TSO or z/OS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment. Contrast with Recoverable Resource Manager Services attachment facility.

call-level interface (CLI)

A callable application programming interface (API) for database access, which is an alternative to using embedded SQL.

cascade delete

A process by which DB2 enforces referential constraints by deleting all descendent rows of a deleted parent row.

CASE expression

An expression that is selected based on the evaluation of one or more conditions.

cast function

A function that is used to convert instances of a (source) data type into instances of a different (target) data type.

castout

The DB2 process of writing changed pages from a group buffer pool to disk.

castout owner

The DB2 member that is responsible for casting out a particular page set or partition.

catalog

In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table

Any table in the DB2 catalog.

CCSID

See coded character set identifier.

CDB

See communications database.

CDRA

See Character Data Representation Architecture.

CEC

See central processor complex.

central electronic complex (CEC)

See central processor complex.

central processor complex (CPC)

A physical collection of hardware that consists of main storage, one or more central processors, timers, and channels.

central processor (CP)

The part of the computer that contains the sequencing and processing facilities for instruction execution, initial program load, and other machine operations.

CFRM See coupling facility resource management.

CFRM policy

The allocation rules for a coupling facility structure that are declared by a z/OS administrator.

character conversion

The process of changing characters from one encoding scheme to another.

Character Data Representation Architecture (CDRA)

An architecture that is used to achieve consistent representation, processing, and interchange of string data.

character large object (CLOB)

A character string data type that contains a sequence of bytes that represent

	characters (single-byte, multibyte, or both)	from occurring until the claim is released,
	that can range in size from 0 bytes to 2	which usually occurs at a commit point.
	GB, less 1 byte. In general, CLOB values	Contrast with drain.
	are used whenever a character string	
	might exceed the limits of the VARCHAR	
	type.	
	character set	
	A defined set of characters.	
	character string	
	A sequence of bytes that represent bit	
	data, single-byte characters, or a mixture	
	of single-byte and multibyte characters.	
	Character data can be further classified as	
	CHARACTER, VARCHAR, or CLOB.	
	check constraint	
	A user-defined constraint that specifies	
	the values that specific columns of a base	
	table can contain.	
	check integrity	
	The condition that exists when each row	
	in a table conforms to the check	
	constraints that are defined on that table.	
	check pending	
	A state of a table space or partition that	
	prevents its use by some utilities and by	
	some SQL statements because of rows	
	that violate referential constraints, check	
	constraints, or both.	
	checkpoint	
	A point at which DB2 records status	
	information on the DB2 log; the recovery	
	process uses this information if DB2	
	abnormally terminates.	
	child lock	
	For explicit hierarchical locking, a lock	
	that is held on either a table, page, row,	
	or a large object (LOB). Each child lock	
	has a parent lock. See also parent lock.	
	CI	See control interval.
	CICS	Represents (in this information): CICS
	Transaction Server for z/OS: Customer	
	Information Control System Transaction	
	Server for z/OS.	
	CICS attachment facility	
	A facility that provides a multithread	
	connection to DB2 to allow applications	
	that run in the CICS environment to	
	execute DB2 statements.	
	claim	A notification to DB2 that an object is
		being accessed. Claims prevent drains
	claim class	A specific type of object access that can be
		one of the following isolation levels:
		• Cursor stability (CS)
		• Repeatable read (RR)
		• Write
	class of service	
	A VTAM term for a list of routes through	
	a network, arranged in an order of	
	preference for their use.	
	clause	In SQL, a distinct part of a statement,
		such as a SELECT clause or a WHERE
		clause.
	CLI	See call-level interface.
	client	See requester.
	CLOB	See character large object.
	clone object	
	An object that is associated with a clone	
	table, including the clone table itself and	
	check constraints, indexes, and BEFORE	
	triggers on the clone table.	
	clone table	
	A table that is structurally identical to a	
	base table. The base and clone table each	
	have separate underlying VSAM data	
	sets, which are identified by their data set	
	instance numbers. Contrast with base	
	table.	
	closed application	
	An application that requires exclusive use	
	of certain statements on certain DB2	
	objects, so that the objects are managed	
	solely through the external interface of	
	that application.	
	clustering index	
	An index that determines how rows are	
	physically ordered (<i>clustered</i>) in a table	
	space. If a clustering index on a	
	partitioned table is not a partitioning	
	index, the rows are ordered in cluster	
	sequence within each data partition	
	instead of spanning partitions.	
	CM	See conversion mode.
	CM*	See conversion mode*.

C++ member	A data object or function in a structure, union, or class.	cold start	A process by which DB2 restarts without processing any log records. Contrast with warm start.
C++ member function	An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and to the member functions of objects in its class. Member functions are also called methods.	collection	A group of packages that have the same qualifier.
C++ object	A region of storage. An object is created when a variable is defined or a new function is invoked. An instance of a class.	column	The vertical component of a table. A column has a name and a particular data type (for example, character, decimal, or integer).
coded character set	A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.	column function	See aggregate function.
coded character set identifier (CCSID)	A 16-bit number that uniquely identifies a coded representation of graphic characters. It designates an encoding scheme identifier and one or more pairs that consist of a character set identifier and an associated code page identifier.	"come from" checking	An LU 6.2 security option that defines a list of authorization IDs that are allowed to connect to DB2 from a partner LU.
code page	A set of assignments of characters to code points. Within a code page, each code point has only one specific meaning. In EBCDIC, for example, the character <i>A</i> is assigned code point X'C1', and character <i>B</i> is assigned code point X'C2'.	command	A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.
code point	In CDRA, a unique bit pattern that represents a character in a code page.	command prefix	A 1- to 8-character command identifier. The command prefix distinguishes the command as belonging to an application or subsystem rather than to z/OS.
code unit	The fundamental binary width in a computer architecture that is used for representing character data, such as 7 bits, 8 bits, 16 bits, or 32 bits. Depending on the character encoding form that is used, each code point in a coded character set can be represented by one or more code units.	command recognition character (CRC)	A character that permits a z/OS console operator or an IMS subsystem user to route DB2 commands to specific DB2 subsystems.
coexistence	During migration, the period of time in which two releases exist in the same data sharing group.	command scope	The scope of command operation in a data sharing group.
		commit	The operation that ends a unit of work by releasing locks so that the database changes that are made by that unit of work can be perceived by other processes. Contrast with rollback.
		commit point	A point in time when data is considered consistent.
		common service area (CSA)	In z/OS, a part of the common area that contains data areas that are addressable by all address spaces. Most DB2 use is in the extended CSA, which is above the 16-MB line.

communications database (CDB)

A set of tables in the DB2 catalog that are used to establish conversations with remote database management systems.

comparison operator

A token (such as =, >, or <) that is used to specify a relationship between two values.

compatibility mode

See conversion mode.

compatibility mode* (CM*)

See conversion mode*.

composite key

An ordered set of key columns or expressions of the same table.

compression dictionary

The dictionary that controls the process of compression and decompression. This dictionary is created from the data in the table space or table space partition.

concurrency

The shared use of resources by more than one application process at the same time.

conditional restart

A DB2 restart that is directed by a user-defined conditional restart control record (CRCR).

connection

In SNA, the existence of a communication path between two partner LUs that allows information to be exchanged (for example, two DB2 subsystems that are connected and communicating by way of a conversation).

connection context

In SQLJ, a Java object that represents a connection to a data source.

connection declaration clause

In SQLJ, a statement that declares a connection to a data source.

connection handle

The data object containing information that is associated with a connection that DB2 ODBC manages. This includes general status information, transaction status, and diagnostic information.

connection ID

An identifier that is supplied by the attachment facility and that is associated with a specific address space connection.

consistency token

A timestamp that is used to generate the version identifier for an application. See also version.

constant

A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with variable.

constraint

A rule that limits the values that can be inserted, deleted, or updated in a table. See referential constraint, check constraint, and unique constraint.

context

An application's logical connection to the data source and associated DB2 ODBC connection information that allows the application to direct its operations to a data source. A DB2 ODBC context represents a DB2 thread.

contracting conversion

A process that occurs when the length of a converted string is smaller than that of the source string. For example, this process occurs when an EBCDIC mixed-data string that contains DBCS characters is converted to ASCII mixed data; the converted string is shorter because the shift codes are removed.

control interval (CI)

- A unit of information that VSAM transfers between virtual and auxiliary storage.
- In a key-sequenced data set or file, the set of records that an entry in the sequence-set index record points to.

conversation

Communication, which is based on LU 6.2 or Advanced Program-to-Program Communication (APPC), between an application and a remote transaction program over an SNA logical unit-to-logical unit (LU-LU) session that allows communication while processing a transaction.

conversion mode* (CM*)

A stage of the version-to-version migration process that applies to a DB2 subsystem or data sharing group that was in enabling-new-function mode (ENFM), enabling-new-function mode* (ENFM*), or

new-function mode (NFM) at one time. Fallback to a prior version is not supported. When in conversion mode*, a DB2 data sharing group cannot coexist with members that are still at the prior version level. Contrast with conversion mode, enabling-new-function mode, enabling-new-function mode*, and new-function mode.

Previously known as compatibility mode* (CM*).

conversion mode (CM)

The first stage of the version-to-version migration process. In a DB2 data sharing group, members in conversion mode can coexist with members that are still at the prior version level. Fallback to the prior version is also supported. When in conversion mode, the DB2 subsystem cannot use most new functions of the new version. Contrast with conversion mode*, enabling-new-function mode, enabling-new-function mode*, and new-function mode.

Previously known as compatibility mode (CM).

coordinator

The system component that coordinates the commit or rollback of a unit of work that includes work that is done on one or more other systems.

coprocessor

See SQL statement coprocessor.

copy pool

A collection of names of storage groups that are processed collectively for fast replication operations.

copy target

A named set of SMS storage groups that are to be used as containers for copy pool volume copies. A copy target is an SMS construct that lets you define which storage groups are to be used as containers for volumes that are copied by using FlashCopy functions.

copy version

A point-in-time FlashCopy copy that is managed by HSM. Each copy pool has a version parameter that specifies the number of copy versions to be maintained on disk.

correlated columns

A relationship between the value of one column and the value of another column.

correlated subquery

A subquery (part of a WHERE or HAVING clause) that is applied to a row or group of rows of a table or view that is named in an outer subselect statement.

correlation ID

An identifier that is associated with a specific thread. In TSO, it is either an authorization ID or the job name.

correlation name

An identifier that is specified and used within a single SQL statement as the exposed name for objects such as a table, view, table function reference, nested table expression, or result of a data change statement. Correlation names are useful in an SQL statement to allow two distinct references to the same base table and to allow an alternative name to be used to represent an object.

cost category

A category into which DB2 places cost estimates for SQL statements at the time the statement is bound. The cost category is externalized in the COST_CATEGORY column of the DSN_STATEMENT_TABLE when a statement is explained.

coupling facility

A special PR/SM logical partition (LPAR) that runs the coupling facility control program and provides high-speed caching, list processing, and locking functions in a Parallel Sysplex.

coupling facility resource management (CFRM)

A component of z/OS that provides the services to manage coupling facility resources in a Parallel Sysplex. This management includes the enforcement of CFRM policies to ensure that the coupling facility and structure requirements are satisfied.

CP See central processor.

CPC See central processor complex.

CRC See command recognition character.

created temporary table

A persistent table that holds temporary data and is defined with the SQL statement CREATE GLOBAL

- TEMPORARY TABLE. Information about created temporary tables is stored in the DB2 catalog and can be shared across application processes. Contrast with declared temporary table. See also temporary table.
- cross-system coupling facility (XCF)**
A component of z/OS that provides functions to support cooperation between authorized programs that run within a Sysplex.
- cross-system extended services (XES)**
A set of z/OS services that allow multiple instances of an application or subsystem, running on different systems in a Sysplex environment, to implement high-performance, high-availability data sharing by using a coupling facility.
- CS** See cursor stability.
- CSA** See common service area.
- CT** See cursor table.
- current data**
Data within a host structure that is current with (identical to) the data within the base table.
- current status rebuild**
The second phase of restart processing during which the status of the subsystem is reconstructed from information on the log.
- cursor** A control structure that an application program uses to point to a single row or multiple rows within some ordered set of rows of a result table. A cursor can be used to retrieve, update, or delete rows from a result table.
- cursor sensitivity**
The degree to which database updates are visible to the subsequent FETCH statements in a cursor.
- cursor stability (CS)**
The isolation level that provides maximum concurrency without the ability to read uncommitted data. With cursor stability, a unit of work holds locks only on its uncommitted changes and on the current row of each of its cursors. See also read stability, repeatable read, and uncommitted read.
- cursor table (CT)**
The internal representation of a cursor.
- cycle** A set of tables that can be ordered so that each table is a descendent of the one before it, and the first table is a descendent of the last table. A self-referencing table is a cycle with a single member. See also referential cycle.
- database**
A collection of tables, or a collection of table spaces and index spaces.
- database access thread (DBAT)**
A thread that accesses data at the local subsystem on behalf of a remote subsystem.
- database administrator (DBA)**
An individual who is responsible for designing, developing, operating, safeguarding, maintaining, and using a database.
- database alias**
The name of the target server if it is different from the location name. The database alias is used to provide the name of the database server as it is known to the network.
- database descriptor (DBD)**
An internal representation of a DB2 database definition, which reflects the data definition that is in the DB2 catalog. The objects that are defined in a database descriptor are table spaces, tables, indexes, index spaces, relationships, check constraints, and triggers. A DBD also contains information about accessing tables in the database.
- database exception status**
In a data sharing environment, an indication that something is wrong with a database.
- database identifier (DBID)**
An internal identifier of the database.
- database management system (DBMS)**
A software system that controls the creation, organization, and modification of a database and the access to the data that is stored within it.
- database request module (DBRM)**
A data set member that is created by the DB2 precompiler and that contains

information about SQL statements.
DBRMs are used in the bind process.

database server

The target of a request from a local application or a remote intermediate database server.

data currency

The state in which the data that is retrieved into a host variable in a program is a copy of the data in the base table.

data-dependent pagination

The process used when applications need to access part of a DB2 result set that is based on a logical key value.

data dictionary

A repository of information about an organization's application programs, databases, logical data models, users, and authorizations.

data partition

A VSAM data set that is contained within a partitioned table space.

data-partitioned secondary index (DPSI)

A secondary index that is partitioned according to the underlying data.
Contrast with nonpartitioned secondary index.

| **data set instance number**

| A number that indicates the data set that
| contains the data for an object.

data sharing

A function of DB2 for z/OS that enables applications on different DB2 subsystems to read from and write to the same data concurrently.

data sharing group

A collection of one or more DB2 subsystems that directly access and change the same data while maintaining data integrity.

data sharing member

A DB2 subsystem that is assigned by XCF services to a data sharing group.

data source

A local or remote relational or non-relational data manager that is capable of supporting data access via an ODBC driver that supports the ODBC

APIs. In the case of DB2 for z/OS, the data sources are always relational database managers.

data type

An attribute of columns, constants, variables, parameters, special registers, and the results of functions and expressions.

data warehouse

A system that provides critical business information to an organization. The data warehouse system cleanses the data for accuracy and currency, and then presents the data to decision makers so that they can interpret and use it effectively and efficiently.

DBA See database administrator.

DBAT See database access thread.

DB2 catalog

A collection of tables that are maintained by DB2 and contain descriptions of DB2 objects, such as tables, views, and indexes.

DBCLOB

See double-byte character large object.

DB2 command

An instruction to the DB2 subsystem that a user enters to start or stop DB2, to display information on current users, to start or stop databases, to display information on the status of databases, and so on.

DBCS See double-byte character set.

DBD See database descriptor.

DB2I See DB2 Interactive.

DBID See database identifier.

DB2 Interactive (DB2I)

An interactive service within DB2 that facilitates the execution of SQL statements, DB2 (operator) commands, and programmer commands, and the invocation of utilities.

DBMS

See database management system.

DBRM

See database request module.

DB2 thread

| The database manager structure that

describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to the database manager resources and services. Most DB2 for z/OS functions execute under a thread structure.

DCLGEN

See declarations generator.

DDF See distributed data facility.

deadlock

Unresolved contention for the use of a resource, such as a table or an index.

declarations generator (DCLGEN)

A subcomponent of DB2 that generates SQL table declarations and COBOL, C, or PL/I data structure declarations that conform to the table. The declarations are generated from DB2 system catalog information.

declared temporary table

A non-persistent table that holds temporary data and is defined with the SQL statement DECLARE GLOBAL TEMPORARY TABLE. Information about declared temporary tables is not stored in the DB2 catalog and can be used only by the application process that issued the DECLARE statement. Contrast with created temporary table. See also temporary table.

default value

A predetermined value, attribute, or option that is assumed when no other value is specified. A default value can be defined for column data in DB2 tables by specifying the DEFAULT keyword in an SQL statement that changes data (such as INSERT, UPDATE, and MERGE).

deferred embedded SQL

SQL statements that are neither fully static nor fully dynamic. These statements are embedded within an application and are prepared during the execution of the application.

deferred write

The process of asynchronously writing changed data pages to disk.

degree of parallelism

The number of concurrently executed operations that are initiated to process a query.

delete hole

The location on which a cursor is positioned when a row in a result table is refetched and the row no longer exists on the base table. See also update hole.

delete rule

The rule that tells DB2 what to do to a dependent row when a parent row is deleted. Delete rules include CASCADE, RESTRICT, SET NULL, or NO ACTION.

delete trigger

A trigger that is defined with the triggering delete SQL operation.

delimited identifier

A sequence of one or more characters enclosed by escape characters, such as quotation marks ("").

delimiter token

A string constant, a delimited identifier, an operator symbol, or any of the special characters that are shown in DB2 syntax diagrams.

denormalization

The intentional duplication of columns in multiple tables to increase data redundancy. Denormalization is sometimes necessary to minimize performance problems. Contrast with normalization.

dependent

An object (row, table, or table space) that has at least one parent. The object is also said to be a dependent (row, table, or table space) of its parent. See also parent row, parent table, and parent table space.

dependent row

A row that contains a foreign key that matches the value of a primary key in the parent row.

dependent table

A table that is a dependent in at least one referential constraint.

descendent

An object that is a dependent of an object or is the dependent of a descendent of an object.

descendent row

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

descendent table

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

deterministic function

A user-defined function whose result is dependent on the values of the input arguments. That is, successive invocations with the same input values produce the same answer. Sometimes referred to as a *not-variant* function. Contrast with nondeterministic function (sometimes called a *variant function*).

dimension

A data category such as time, products, or markets. The elements of a dimension are referred to as members. See also dimension table.

dimension table

The representation of a dimension in a star schema. Each row in a dimension table represents all of the attributes for a particular member of the dimension. See also dimension, star schema, and star join.

directory

The DB2 system database that contains internal objects such as database descriptors and skeleton cursor tables.

disk

A direct-access storage device that records data magnetically.

distinct type

A user-defined data type that is represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

distributed data

Data that resides on a DBMS other than the local system.

distributed data facility (DDF)

A set of DB2 components through which DB2 communicates with another relational database management system.

Distributed Relational Database Architecture (DRDA)

A connection protocol for distributed relational database processing that is used by IBM relational database products. DRDA includes protocols for communication between an application and a remote relational database

management system, and for communication between relational database management systems. See also DRDA access.

DNS See domain name server.

DOCID

See document ID.

document ID

A value that uniquely identifies a row that contains an XML column. This value is stored with the row and never changes.

domain

The set of valid values for an attribute.

domain name

The name by which TCP/IP applications refer to a TCP/IP host within a TCP/IP network.

domain name server (DNS)

A special TCP/IP network server that manages a distributed directory that is used to map TCP/IP host names to IP addresses.

double-byte character large object (DBCLOB)

A graphic string data type in which a sequence of bytes represent double-byte characters that range in size from 0 bytes to 2 GB, less 1 byte. In general, DBCLOB values are used whenever a double-byte character string might exceed the limits of the VARGRAPHIC type.

double-byte character set (DBCS)

A set of characters, which are used by national languages such as Japanese and Chinese, that have more symbols than can be represented by a single byte. Each character is 2 bytes in length. Contrast with single-byte character set and multibyte character set.

double-precision floating point number

A 64-bit approximate representation of a real number.

DPSI See data-partitioned secondary index.

drain

The act of acquiring a locked resource by quiescing access to that object. Contrast with claim.

drain lock

A lock on a claim class that prevents a claim from occurring.

DRDA

See Distributed Relational Database Architecture.

DRDA access

An open method of accessing distributed data that you can use to connect to another database server to execute packages that were previously bound at the server location.

DSN

- The default DB2 subsystem name.
- The name of the TSO command processor of DB2.
- The first three characters of DB2 module and macro names.

dynamic cursor

A named control structure that an application program uses to change the size of the result table and the order of its rows after the cursor is opened. Contrast with static cursor.

dynamic dump

A dump that is issued during the execution of a program, usually under the control of that program.

dynamic SQL

SQL statements that are prepared and executed at run time. In dynamic SQL, the SQL statement is contained as a character string in a host variable or as a constant, and it is not precompiled.

EA-enabled table space

A table space or index space that is enabled for extended addressability and that contains individual partitions (or pieces, for LOB table spaces) that are greater than 4 GB.

EB

See exabyte.

EBCDIC

Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the z/OS, VM, VSE, and iSeries environments. Contrast with ASCII and Unicode.

embedded SQL

SQL statements that are coded within an application program. See static SQL.

enabling-new-function mode* (ENFM*)

A transitional stage of the

version-to-version migration process that applies to a DB2 subsystem or data sharing group that was in new-function mode (NFM) at one time. When in enabling-new-function mode*, a DB2 subsystem or data sharing group is preparing to use the new functions of the new version but cannot yet use them. A data sharing group that is in enabling-new-function mode* cannot coexist with members that are still at the prior version level. Fallback to a prior version is not supported. Contrast with conversion mode, conversion mode*, enabling-new-function mode, and new-function mode.

enabling-new-function mode (ENFM)

A transitional stage of the version-to-version migration process during which the DB2 subsystem or data sharing group is preparing to use the new functions of the new version. When in enabling-new-function mode, a DB2 data sharing group cannot coexist with members that are still at the prior version level. Fallback to a prior version is not supported, and most new functions of the new version are not available for use in enabling-new-function mode. Contrast with conversion mode, conversion mode*, enabling-new-function mode*, and new-function mode.

enclave

In Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is similar to a program or run unit. See also WLM enclave.

encoding scheme

A set of rules to represent character data (ASCII, EBCDIC, or Unicode).

end column

In a system period or an application period, the column that indicates the end of the period.

ENFM See enabling-new-function mode.

ENFM*

See enabling-new-function mode*.

entity

A person, object, or concept about which information is stored. In a relational database, entities are represented as tables. A database includes information

| about the entities in an organization or
| business, and their relationships to each
| other.

enumerated list

A set of DB2 objects that are defined with a LISTDEF utility control statement in which pattern-matching characters (*, %;, _ or ?) are not used.

environment

A collection of names of logical and physical resources that are used to support the performance of a function.

environment handle

| A handle that identifies the global context
| for database access. All data that is
| pertinent to all objects in the environment
| is associated with this handle.

equijoin

A join operation in which the join-condition has the form *expression = expression*. See also join, full outer join, inner join, left outer join, outer join, and right outer join.

error page range

A range of pages that are considered to be physically damaged. DB2 does not allow users to access any pages that fall within this range.

escape character

The symbol, a double quotation (") for example, that is used to enclose an SQL delimited identifier.

exabyte

A unit of measure for processor, real and virtual storage capacities, and channel volume that has a value of 1 152 921 504 606 846 976 bytes or 2⁶⁰.

exception

| An SQL operation that involves the
| EXCEPT set operator, which combines
| two result tables. The result of an
| exception operation consists of all of the
| rows that are in only one of the result
| tables.

exception table

A table that holds rows that violate referential constraints or check constraints that the CHECK DATA utility finds.

exclusive lock

A lock that prevents concurrently

executing application processes from reading or changing data. Contrast with share lock.

executable statement

An SQL statement that can be embedded in an application program, dynamically prepared and executed, or issued interactively.

execution context

In SQLJ, a Java object that can be used to control the execution of SQL statements.

exit routine

A user-written (or IBM-provided default) program that receives control from DB2 to perform specific functions. Exit routines run as extensions of DB2.

expanding conversion

A process that occurs when the length of a converted string is greater than that of the source string. For example, this process occurs when an ASCII mixed-data string that contains DBCS characters is converted to an EBCDIC mixed-data string; the converted string is longer because shift codes are added.

explicit hierarchical locking

Locking that is used to make the parent-child relationship between resources known to IRLM. This kind of locking avoids global locking overhead when no inter-DB2 interest exists on a resource.

explicit privilege

| A privilege that has a name and is held as
| the result of an SQL GRANT statement
| and revoked as the result of an SQL
| REVOKE statement. For example, the
| SELECT privilege.

exposed name

A correlation name or a table or view name for which a correlation name is not specified.

expression

An operand or a collection of operators and operands that yields a single value.

Extended Recovery Facility (XRF)

A facility that minimizes the effect of failures in z/OS, VTAM, the host processor, or high-availability applications during sessions between high-availability applications and designated terminals.

This facility provides an alternative subsystem to take over sessions from the failing subsystem.

Extensible Markup Language (XML)

A standard metalanguage for defining markup languages that is a subset of Standardized General Markup Language (SGML).

external function

A function that has its functional logic implemented in a programming language application that resides outside the database, in the file system of the database server. The association of the function with the external code application is specified by the EXTERNAL clause in the CREATE FUNCTION statement. External functions can be classified as external scalar functions and external table functions. Contrast with sourced function, built-in function, and SQL function.

external procedure

A procedure that has its procedural logic implemented in an external programming language application. The association of the procedure with the external application is specified by a CREATE PROCEDURE statement with a LANGUAGE clause that has a value other than SQL and an EXTERNAL clause that implicitly or explicitly specifies the name of the external application. Contrast with external SQL procedure and native SQL procedure.

external routine

A user-defined function or stored procedure that is based on code that is written in an external programming language.

external SQL procedure

An SQL procedure that is processed using a generated C program that is a representation of the procedure. When an external SQL procedure is called, the C program representation of the procedure is executed in a stored procedures address space. Contrast with external procedure and native SQL procedure.

failed member state

A state of a member of a data sharing group in which the member's task,

address space, or z/OS system terminates before the state changes from active to quiesced.

fallback

The process of returning to a previous release of DB2 after attempting or completing migration to a current release. Fallback is supported only from a subsystem that is in conversion mode.

false global lock contention

A contention indication from the coupling facility that occurs when multiple lock names are hashed to the same indicator and when no real contention exists.

fan set

A direct physical access path to data, which is provided by an index, hash, or link; a fan set is the means by which DB2 supports the ordering of data.

federated database

The combination of a DB2 server (in Linux, UNIX, and Windows environments) and multiple data sources to which the server sends queries. In a federated database system, a client application can use a single SQL statement to join data that is distributed across multiple database management systems and can view the data as if it were local.

fetch orientation

The specification of the desired placement of the cursor as part of a FETCH statement. The specification can be before or after the rows of the result table (with BEFORE or AFTER). The specification can also have either a single-row fetch orientation (for example, NEXT, LAST, or ABSOLUTE *n*) or a rowset fetch orientation (for example, NEXT ROWSET, LAST ROWSET, or ROWSET STARTING AT ABSOLUTE *n*).

field procedure

A user-written exit routine that is designed to receive a single value and transform (encode or decode) it in any way the user can specify.

file reference variable

A host variable that is declared with one of the derived data types (BLOB_FILE, CLOB_FILE, DBCLOB_FILE); file

| reference variables direct the reading of a
| LOB from a file or the writing of a LOB
| into a file.

filter factor

A number between zero and one that estimates the proportion of rows in a table for which a predicate is true.

fixed-length string

| A character, graphic, or binary string
| whose length is specified and cannot be
| changed. Contrast with varying-length
| string.

FlashCopy

| A function on the IBM Enterprise Storage
| Server that can, in conjunction with the
| BACKUP SYSTEM utility, create a
| point-in-time copy of data while an
| application is running.

foreign key

A column or set of columns in a dependent table of a constraint relationship. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table. Each foreign key value must either match a parent key value in the related parent table or be null.

forest An ordered set of subtrees of XML nodes.

forward log recovery

The third phase of restart processing during which DB2 processes the log in a forward direction to apply all REDO log records.

free space

The total amount of unused space in a page; that is, the space that is not used to store records or control information is free space.

full outer join

The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of both tables. See also join, equijoin, inner join, left outer join, outer join, and right outer join.

| **fullselect**

| A subselect, a fullselect in parentheses, or
| a number of both that are combined by
| set operators. Fullselect specifies a result
| table. If a set operator is not used, the

result of the fullselect is the result of the specified subselect or fullselect.

fully escaped mapping

A mapping from an SQL identifier to an XML name when the SQL identifier is a column name.

function

A mapping, which is embodied as a program (the function body) that can be invoked by means of zero or more input values (arguments) to a single value (the result). See also aggregate function and scalar function.

Functions can be user-defined, built-in, or generated by DB2. (See also built-in function, cast function, external function, sourced function, SQL function, and user-defined function.)

function definer

The authorization ID of the owner of the schema of the function that is specified in the CREATE FUNCTION statement.

function package

A package that results from binding the DBRM for a function program.

function package owner

The authorization ID of the user who binds the function program's DBRM into a function package.

function signature

The logical concatenation of a fully qualified function name with the data types of all of its parameters.

GB Gigabyte. A value of (1 073 741 824 bytes).

GBP See group buffer pool.

GBP-dependent

The status of a page set or page set partition that is dependent on the group buffer pool. Either read/write interest is active among DB2 subsystems for this page set, or the page set has changed pages in the group buffer pool that have not yet been cast out to disk.

generalized trace facility (GTF)

A z/OS service program that records significant system events such as I/O interrupts, SVC interrupts, program interrupts, or external interrupts.

generic resource name

A name that VTAM uses to represent

several application programs that provide the same function in order to handle session distribution and balancing in a Sysplex environment.

getpage

An operation in which DB2 accesses a data page.

global lock

A lock that provides concurrency control within and among DB2 subsystems. The scope of the lock is across all DB2 subsystems of a data sharing group.

global lock contention

Conflicts on locking requests between different DB2 members of a data sharing group when those members are trying to serialize shared resources.

governor

See resource limit facility.

graphic string

A sequence of DBCS characters. Graphic data can be further classified as GRAPHIC, VARGRAPHIC, or DBCLOB.

GRECP

See group buffer pool recovery pending.

gross lock

The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group buffer pool duplexing

The ability to write data to two instances of a group buffer pool structure: a primary group buffer pool and a secondary group buffer pool. z/OS publications refer to these instances as the “old” (for primary) and “new” (for secondary) structures.

group buffer pool (GBP)

A coupling facility cache structure that is used by a data sharing group to cache data and to ensure that the data is consistent for all members.

group buffer pool recovery pending (GRECP)

The state that exists after the buffer pool for a data sharing group is lost. When a page set is in this state, changes that are recorded in the log must be applied to the affected page set before the page set can be used.

group level

The release level of a data sharing group,

which is established when the first member migrates to a new release.

group name

The z/OS XCF identifier for a data sharing group.

group restart

A restart of at least one member of a data sharing group after the loss of either locks or the shared communications area.

GTF

See generalized trace facility.

handle

In DB2 ODBC, a variable that refers to a data structure and associated resources. See also statement handle, connection handle, and environment handle.

hash access

Access to a table using the hash value of a key that is defined by the *organization-clause* of a CREATE TABLE statement or ALTER TABLE statement.

hash overflow index

A DB2 index used to track data rows that do not fit into the fixed hash space, and therefore, reside in the hash overflow space. DB2 accesses the hash overflow index to fetch rows from the hash overflow area.

help panel

A screen of information that presents tutorial text to assist a user at the workstation or terminal.

heuristic damage

The inconsistency in data between one or more participants that results when a heuristic decision to resolve an indoubt LUW at one or more participants differs from the decision that is recorded at the coordinator.

heuristic decision

A decision that forces indoubt resolution at a participant by means other than automatic resynchronization between coordinator and participant.

histogram statistics

A way of summarizing data distribution. This technique divides up the range of possible values in a data set into intervals, such that each interval contains approximately the same percentage of the values. A set of statistics are collected for each interval.

history table

A base table that is associated with a system-period temporal table. A history table is used by DB2 to store the historical versions of the rows from the associated system-period temporal table.

hole

A row of the result table that cannot be accessed because of a delete or an update that has been performed on the row. See also delete hole and update hole.

home address space

The area of storage that z/OS currently recognizes as *dispatched*.

host

The set of programs and resources that are available on a given TCP/IP instance.

host expression

A Java variable or expression that is referenced by SQL clauses in an SQLJ application program.

host identifier

A name that is declared in the host program.

host language

A programming language in which you can embed SQL statements.

host program

An application program that is written in a host language and that contains embedded SQL statements.

host structure

In an application program, a structure that is referenced by embedded SQL statements.

host variable

In an application program written in a host language, an application variable that is referenced by embedded SQL statements.

host variable array

An array of elements, each of which corresponds to a value for a column. The dimension of the array determines the maximum number of rows for which the array can be used.

IBM System z9 Integrated Processor (zIIP)

A specialized processor that can be used for some DB2 functions.

IDCAMS

An IBM program that is used to process access method services commands. It can

be invoked as a job or jobstep, from a TSO terminal, or from within a user's application program.

IDCAMS LISTCAT

A facility for obtaining information that is contained in the access method services catalog.

identity column

A column that provides a way for DB2 to automatically generate a numeric value for each row. Identity columns are defined with the AS IDENTITY clause. Uniqueness of values can be ensured by defining a unique index that contains only the identity column. A table can have no more than one identity column.

IFCID See instrumentation facility component identifier.

IFI See instrumentation facility interface.

IFI call

An invocation of the instrumentation facility interface (IFI) by means of one of its defined functions.

image copy

An exact reproduction of all or part of a table space. DB2 provides utility programs to make full image copies (to copy the entire table space) or incremental image copies (to copy only those pages that have been modified since the last image copy).

IMS attachment facility

A DB2 subcomponent that uses z/OS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

in-abort

A status of a unit of recovery. If DB2 fails after a unit of recovery begins to be rolled back, but before the process is completed, DB2 continues to back out the changes during restart.

in-commit

A status of a unit of recovery. If DB2 fails after beginning its phase 2 commit processing, it "knows," when restarted, that changes made to data are consistent. Such units of recovery are termed *in-commit*.

independent

An object (row, table, or table space) that is neither a parent nor a dependent of another object.

index A set of pointers that are logically ordered by the values of a key. Indexes can provide faster access to data and can enforce uniqueness on the rows in a table.

index-controlled partitioning

A type of partitioning in which partition boundaries for a partitioned table are controlled by values that are specified on the CREATE INDEX statement. Partition limits are saved in the LIMITKEY column of the SYSIBM.SYSINDEXPART catalog table.

index key

The set of columns in a table that is used to determine the order of index entries.

index partition

A VSAM data set that is contained within a partitioning index space.

index space

A page set that is used to store the entries of one index.

indicator column

A 4-byte value that is stored in a base table in place of a LOB column.

indicator variable

A variable that is used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

indoubt

A status of a unit of recovery. If DB2 fails after it has finished its phase 1 commit processing and before it has started phase 2, only the commit coordinator knows if an individual unit of recovery is to be committed or rolled back. At restart, if DB2 lacks the information it needs to make this decision, the status of the unit of recovery is *indoubt* until DB2 obtains this information from the coordinator. More than one unit of recovery can be indoubt at restart.

indoubt resolution

The process of resolving the status of an indoubt logical unit of work to either the committed or the rollback state.

inflight

A status of a unit of recovery. If DB2 fails before its unit of recovery completes phase 1 of the commit process, it merely backs out the updates of its unit of recovery at restart. These units of recovery are termed *inflight*.

inheritance

The passing downstream of class resources or attributes from a parent class in the class hierarchy to a child class.

initialization file

For DB2 ODBC applications, a file containing values that can be set to adjust the performance of the database manager.

inline copy

A copy that is produced by the LOAD or REORG utility. The data set that the inline copy produces is logically equivalent to a full image copy that is produced by running the COPY utility with read-only access (SHRLEVEL REFERENCE).

inline SQL PL

A subset of SQL procedural language that can be used in SQL functions, triggers, and dynamic compound statements. See also SQL procedural language.

inner join

The result of a join operation that includes only the matched rows of both tables that are being joined. See also join, equijoin, full outer join, left outer join, outer join, and right outer join.

inoperative package

In DB2 Version 9.1 for z/OS and earlier, a package that cannot be used because one or more user-defined functions or procedures that the package depends on were dropped. Such a package must be explicitly rebound. Contrast with invalid package.

insensitive cursor

A cursor that is not sensitive to inserts, updates, or deletes that are made to the underlying rows of a result table after the result table has been materialized.

insert trigger

A trigger that is defined with the triggering SQL operation, an insert.

install The process of preparing a DB2 subsystem to operate as a z/OS subsystem.

INSTEAD OF trigger

A trigger that is associated with a single view and is activated by an insert, update, or delete operation on the view and that can define how to propagate the insert, update, or delete operation on the view to the underlying tables of the view. Contrast with BEFORE trigger and AFTER trigger.

instrumentation facility component identifier (IFCID)

A value that names and identifies a trace record of an event that can be traced. As a parameter on the START TRACE and MODIFY TRACE commands, it specifies that the corresponding event is to be traced.

instrumentation facility interface (IFI)

A programming interface that enables programs to obtain online trace data about DB2, to submit DB2 commands, and to pass data to DB2.

Interactive System Productivity Facility (ISPF)

An IBM licensed program that provides interactive dialog services in a z/OS environment.

inter-DB2 R/W interest

A property of data in a table space, index, or partition that has been opened by more than one member of a data sharing group and that has been opened for writing by at least one of those members.

intermediate database server

The target of a request from a local application or a remote application requester that is forwarded to another database server.

internal resource lock manager (IRLM)

A z/OS subsystem that DB2 uses to control communication and database locking.

internationalization

The support for an encoding scheme that is able to represent the code points of characters from many different geographies and languages. To support all geographies, the Unicode standard requires more than 1 byte to represent a single character. See also Unicode.

intersection

An SQL operation that involves the INTERSECT set operator, which combines two result tables. The result of an intersection operation consists of all of the rows that are in both result tables.

invalid package

In DB2 Version 9.1 for z/OS and earlier, a package that depends on an object (other than a user-defined function) that is dropped. Such a package is implicitly rebound on invocation. Contrast with inoperative package.

IP address

A value that uniquely identifies a TCP/IP host.

IRLM See internal resource lock manager.

isolation level

The degree to which a unit of work is isolated from the updating operations of other units of work. See also cursor stability, read stability, repeatable read, and uncommitted read.

ISPF See Interactive System Productivity Facility.

iterator

In SQLJ, an object that contains the result set of a query. An iterator is equivalent to a cursor in other host languages.

iterator declaration clause

In SQLJ, a statement that generates an iterator declaration class. An iterator is an object of an iterator declaration class.

JAR See Java Archive.

Java Archive (JAR)

A file format that is used for aggregating many files into a single file.

JDBC A Sun Microsystems database application programming interface (API) for Java that allows programs to access database management systems by using callable SQL.

join A relational operation that allows retrieval of data from two or more tables based on matching column values. See also equijoin, full outer join, inner join, left outer join, outer join, and right outer join.

KB Kilobyte. A value of 1024 bytes.

Kerberos

A network authentication protocol that is designed to provide strong authentication for client/server applications by using secret-key cryptography.

Kerberos ticket

A transparent application mechanism that transmits the identity of an initiating principal to its target. A simple ticket contains the principal's identity, a session key, a timestamp, and other information, which is sealed using the target's secret key.

key A column, an ordered collection of columns, or an expression that is identified in the description of a table, index, or referential constraint. The same column or expression can be part of more than one key.

key-sequenced data set (KSDS)

A VSAM file or data set whose records are loaded in key sequence and controlled by an index.

KSDS See key-sequenced data set.

large object (LOB)

A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single- and double-byte characters. A LOB can be up to 2 GB minus 1 byte in length. See also binary large object, character large object, and double-byte character large object.

last agent optimization

An optimized commit flow for either presumed-nothing or presumed-abort protocols in which the last agent, or final participant, becomes the commit coordinator. This flow saves at least one message.

latch A DB2 mechanism for controlling concurrent events or the use of system resources.

LCID See log control interval definition.

LDS See linear data set.

leaf page

An index page that contains pairs of keys and RIDs and that points to actual data. Contrast with nonleaf page.

left outer join

The result of a join operation that

includes the matched rows of both tables that are being joined, and that preserves the unmatched rows of the first table. See also join, equijoin, full outer join, inner join, outer join, and right outer join.

limit key

The highest value of the index key for a partition.

linear data set (LDS)

A VSAM data set that contains data but no control information. A linear data set can be accessed as a byte-addressable string in virtual storage.

linkage editor

A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses.

link-edit

The action of creating a loadable computer program using a linkage editor.

list A type of object, which DB2 utilities can process, that identifies multiple table spaces, multiple index spaces, or both. A list is defined with the LISTDEF utility control statement.

list structure

A coupling facility structure that lets data be shared and manipulated as elements of a queue.

L-lock See logical lock.

load module

A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB See large object.

LOB locator

A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a fullword integer value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL operations to the associated LOB value using the locator.

LOB lock

A lock on a LOB value.

LOB table space

A table space that contains all the data for a particular LOB column in the related base table.

local A way of referring to any object that the local DB2 subsystem maintains. A *local table*, for example, is a table that is maintained by the local DB2 subsystem. Contrast with remote.

locale The definition of a subset of a user's environment that combines a CCSID and characters that are defined for a specific language and country.

local lock

A lock that provides intra-DB2 concurrency control, but not inter-DB2 concurrency control; that is, its scope is a single DB2.

local subsystem

The unique relational DBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

location

The unique name of a database server. An application uses the location name to access a DB2 database server. A database alias can be used to override the location name when accessing a remote server.

location alias

Another name by which a database server identifies itself in the network. Applications can use this name to access a DB2 database server.

lock A means of controlling concurrent events or access to data. DB2 locking is performed by the IRLM.

lock duration

The interval over which a DB2 lock is held.

lock escalation

The promotion of a lock from a row, page, or LOB lock to a table space lock because the number of page locks that are concurrently held on a given resource exceeds a preset limit.

locking

The process by which the integrity of data is ensured. Locking prevents concurrent users from accessing inconsistent data. See also claim, drain, and latch.

lock mode

A representation for the type of access that concurrently running programs can have to a resource that a DB2 lock is holding.

lock object

The resource that is controlled by a DB2 lock.

lock promotion

The process of changing the size or mode of a DB2 lock to a higher, more restrictive level.

lock size

The amount of data that is controlled by a DB2 lock on table data; the value can be a row, a page, a LOB, a partition, a table, or a table space.

lock structure

A coupling facility data structure that is composed of a series of lock entries to support shared and exclusive locking for logical resources.

log

A collection of records that describe the events that occur during DB2 execution and that indicate their sequence. The information thus recorded is used for recovery in the event of a failure during DB2 execution.

log control interval definition

A suffix of the physical log record that tells how record segments are placed in the physical control interval.

logical claim

A claim on a logical partition of a nonpartitioning index.

logical index partition

The set of all keys that reference the same data partition.

logical lock (L-lock)

The lock type that transactions use to control intra- and inter-DB2 data concurrency between transactions. Contrast with physical lock (P-lock).

logically complete

A state in which the concurrent copy process is finished with the initialization of the target objects that are being copied. The target objects are available for update.

logical page list (LPL)

A list of pages that are in error and that cannot be referenced by applications until the pages are recovered. The page is in *logical error* because the actual media (coupling facility or disk) might not contain any errors. Usually a connection to the media has been lost.

logical partition

A set of key or RID pairs in a nonpartitioning index that are associated with a particular partition.

logical recovery pending (LRECP)

The state in which the data and the index keys that reference the data are inconsistent.

logical unit (LU)

An access point through which an application program accesses the SNA network in order to communicate with another application program. See also LU name.

logical unit of work

The processing that a program performs between synchronization points.

logical unit of work identifier (LUWID)

A name that uniquely identifies a thread within a network. This name consists of a fully-qualified LU network name, an LUW instance number, and an LUW sequence number.

log initialization

The first phase of restart processing during which DB2 attempts to locate the current end of the log.

log record header (LRH)

A prefix, in every log record, that contains control information.

log record sequence number (LRSN)

An identifier for a log record that is associated with a data sharing member. DB2 uses the LRSN for recovery in the data sharing environment.

log truncation

A process by which an explicit starting RBA is established. This RBA is the point at which the next byte of log data is to be written.

LPL See logical page list.

LRECP

See logical recovery pending.

LRH

See log record header.

LRSN

See log record sequence number.

LU

See logical unit.

LU name

Logical unit name, which is the name by which VTAM refers to a node in a network.

LUW

See logical unit of work.

LUWID

See logical unit of work identifier.

mapping table

A table that the REORG utility uses to map the associations of the RIDs of data records in the original copy and in the shadow copy. This table is created by the user.

mass delete

The deletion of all rows of a table.

materialize

- The process of putting rows from a view or nested table expression into a work file for additional processing by a query.
- The placement of a LOB value into contiguous storage. Because LOB values can be very large, DB2 avoids materializing LOB data until doing so becomes absolutely necessary.

materialized query table

A table that is used to contain information that is derived and can be summarized from one or more source tables. Contrast with base table.

MB

Megabyte (1 048 576 bytes).

MBCS

See multibyte character set.

member name

The z/OS XCF identifier for a particular DB2 subsystem in a data sharing group.

menu

A displayed list of available functions for selection by the operator. A menu is sometimes called a *menu panel*.

metalanguage

A language that is used to create other specialized languages.

migration

The process of converting a subsystem with a previous release of DB2 to an updated or current release. In this process, you can acquire the functions of the updated or current release without losing the data that you created on the previous release.

mixed data string

A character string that can contain both single-byte and double-byte characters.

mode name

A VTAM name for the collection of physical and logical characteristics and attributes of a session.

modify locks

An L-lock or P-lock with a MODIFY attribute. A list of these active locks is kept at all times in the coupling facility lock structure. If the requesting DB2 subsystem fails, that DB2 subsystem's modify locks are converted to retained locks.

multibyte character set (MBCS)

A character set that represents single characters with more than a single byte. UTF-8 is an example of an MBCS. Characters in UTF-8 can range from 1 to 4 bytes in DB2. Contrast with single-byte character set and double-byte character set. See also Unicode.

multidimensional analysis

The process of assessing and evaluating an enterprise on more than one level.

Multiple Virtual Storage (MVS)

An element of the z/OS operating system. This element is also called the Base Control Program (BCP).

multisite update

Distributed relational database processing in which data is updated in more than one location within a single unit of work.

multithreading

Multiple TCBs that are executing one copy of DB2 ODBC code concurrently (sharing a processor) or in parallel (on separate central processors).

MVS See Multiple Virtual Storage.

native SQL procedure

An SQL procedure that is processed by converting the procedural statements to a

native representation that is stored in the database directory, as is done with other SQL statements. When a native SQL procedure is called, the native representation is loaded from the directory, and DB2 executes the procedure. Contrast with external procedure and external SQL procedure.

nested table expression

A fullselect in a FROM clause (surrounded by parentheses).

network identifier (NID)

The network ID that is assigned by IMS or CICS, or if the connection type is RRSAF, the RRS unit of recovery ID (URID).

new-function mode (NFM)

The normal mode of operation that exists after successful completion of a version-to-version migration. At this stage, all new functions of the new version are available for use. A DB2 data sharing group cannot coexist with members that are still at the prior version level, and fallback to a prior version is not supported. Contrast with conversion mode, conversion mode*, enabling-new-function mode, and enabling-new-function mode*.

NFM See new-function mode.

NID See network identifier.

node ID index

See XML node ID index.

nondeterministic function

A user-defined function whose result is not solely dependent on the values of the input arguments. That is, successive invocations with the same argument values can produce a different answer. This type of function is sometimes called a *variant* function. Contrast with deterministic function (sometimes called a *not-variant function*).

nonleaf page

A page that contains keys and page numbers of other pages in the index (either leaf or nonleaf pages). Nonleaf pages never point to actual data. Contrast with leaf page.

nonpartitioned index

An index that is not physically

partitioned. Both partitioning indexes and secondary indexes can be nonpartitioned.

nonpartitioned secondary index (NPSI)

An index on a partitioned table space that is not the partitioning index and is not partitioned. Contrast with data-partitioned secondary index.

nonpartitioning index

See secondary index.

nonscrollable cursor

A cursor that can be moved only in a forward direction. Nonscrollable cursors are sometimes called forward-only cursors or serial cursors.

normalization

A key step in the task of building a logical relational database design. Normalization helps you avoid redundancies and inconsistencies in your data. An entity is normalized if it meets a set of constraints for a particular normal form (first normal form, second normal form, and so on). Contrast with denormalization.

not-variant function

See deterministic function.

NPSI See nonpartitioned secondary index.

NUL The null character ('\0'), which is represented by the value X'00'. In C, this character denotes the end of a string.

null A special value that indicates the absence of information.

null terminator

In C, the value that indicates the end of a string. For EBCDIC, ASCII, and Unicode UTF-8 strings, the null terminator is a single-byte value (X'00'). For Unicode UTF-16 or UCS-2 (wide) strings, the null terminator is a double-byte value (X'0000').

ODBC

See Open Database Connectivity.

ODBC driver

A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

OLAP See online analytical processing.

online analytical processing (OLAP)

The process of collecting data from one or

many sources; transforming and analyzing the consolidated data quickly and interactively; and examining the results across different dimensions of the data by looking for patterns, trends, and exceptions within complex relationships of that data.

Open Database Connectivity (ODBC)

A Microsoft database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time. This means that applications no longer need to be directly linked to the modules of all the database management systems that are supported.

ordinary identifier

An uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier must not be a reserved word.

ordinary token

A numeric constant, an ordinary identifier, a host identifier, or a keyword.

originating task

In a parallel group, the primary agent that receives data from other execution units (referred to as *parallel tasks*) that are executing portions of the query in parallel.

outer join

The result of a join operation that includes the matched rows of both tables that are being joined and preserves some or all of the unmatched rows of the tables that are being joined. See also join, equijoin, full outer join, inner join, left outer join, and right outer join.

overloaded function

A function name for which multiple function instances exist.

package

An object containing a set of SQL statements that have been statically bound and that is available for

processing. A package is sometimes also called an *application package*.

package list

An ordered list of package names that may be used to extend an application plan.

package name

The name of an object that is used for an application package or an SQL procedure package. An application package is a bound version of a database request module (DBRM) that is created by a BIND PACKAGE or REBIND PACKAGE command. An SQL procedural language package is created by a CREATE or ALTER PROCEDURE statement for a native SQL procedure. The name of a package consists of a location name, a collection ID, a package ID, and a version ID.

page

A unit of storage within a table space (4 KB, 8 KB, 16 KB, or 32 KB) or index space (4 KB, 8 KB, 16 KB, or 32 KB). In a table space, a page contains one or more rows of a table. In a LOB or XML table space, a LOB or XML value can span more than one page, but no more than one LOB or XML value is stored on a page.

page set

Another way to refer to a table space or index space. Each page set consists of a collection of VSAM data sets.

page set recovery pending (PSRCP)

A restrictive state of an index space. In this case, the entire page set must be recovered. Recovery of a logical part is prohibited.

panel

A predefined display image that defines the locations and characteristics of display fields on a display surface (for example, a *menu panel*).

parallel complex

A cluster of machines that work together to handle multiple transactions and applications.

parallel group

A set of consecutive operations that execute in parallel and that have the same number of parallel tasks.

parallel I/O processing

A form of I/O processing in which DB2

initiates multiple concurrent requests for a single user query and performs I/O processing concurrently (in *parallel*) on multiple data partitions.

parallelism assistant

In Sysplex query parallelism, a DB2 subsystem that helps to process parts of a parallel query that originates on another DB2 subsystem in the data sharing group.

parallelism coordinator

In Sysplex query parallelism, the DB2 subsystem from which the parallel query originates.

Parallel Sysplex

A set of z/OS systems that communicate and cooperate with each other through certain multisystem hardware components and software services to process customer workloads.

parallel task

The execution unit that is dynamically created to process a query in parallel. A parallel task is implemented by a z/OS service request block.

parameter marker

A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a variable could appear if the statement string were a static SQL statement.

parameter-name

An SQL identifier that designates a parameter in a routine that is written by a user. Parameter names are required for SQL procedures and SQL functions, and they are used in the body of the routine to refer to the values of the parameters. Parameter names are optional for external routines.

parent key

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

parent lock

For explicit hierarchical locking, a lock that is held on a resource that might have child locks that are lower in the hierarchy.

A parent lock is usually the table space lock or the partition intent lock. See also child lock.

parent row

A row whose primary key value is the foreign key value of a dependent row.

parent table

A table whose primary key is referenced by the foreign key of a dependent table.

parent table space

A table space that contains a parent table. A table space containing a dependent of that table is a dependent table space.

participant

An entity other than the commit coordinator that takes part in the commit process. The term participant is synonymous with agent in SNA.

partition

A portion of a page set. Each partition corresponds to a single, independently extendable data set. The maximum size of a partition depends on the number of partitions in the partitioned page set. All partitions of a given page set have the same maximum size.

partition-by-growth table space

A table space whose size can grow to accommodate data growth. DB2 for z/OS manages partition-by-growth table spaces by automatically adding new data sets when the database needs more space to satisfy an insert operation. Contrast with range-partitioned table space. See also universal table space.

partitioned data set (PDS)

A data set in disk storage that is divided into partitions, which are called members. Each partition can contain a program, part of a program, or data. A program library is an example of a partitioned data set.

partitioned index

An index that is physically partitioned. Both partitioning indexes and secondary indexes can be partitioned.

partitioned page set

A partitioned table space or an index space. Header pages, space map pages,

data pages, and index pages reference data only within the scope of the partition.

partitioned table space

A table space that is based on a single table and that is subdivided into partitions, each of which can be processed independently by utilities. Contrast with segmented table space and universal table space.

partitioning index

An index in which the leftmost columns are the partitioning columns of the table. The index can be partitioned or nonpartitioned.

partner logical unit

An access point in the SNA network that is connected to the local DB2 subsystem by way of a VTAM conversation.

path See SQL path.

PDS See partitioned data set.

period An interval of time that is defined by two datetime columns in a temporal table. A period contains a begin column and an end column. See also begin column and end column.

physical consistency

The state of a page that is not in a partially changed state.

physical lock (P-lock)

A type of lock that DB2 acquires to provide consistency of data that is cached in different DB2 subsystems. Physical locks are used only in data sharing environments. Contrast with logical lock (L-lock).

physically complete

The state in which the concurrent copy process is completed and the output data set has been created.

piece A data set of a nonpartitioned page set.

plan See application plan.

plan allocation

The process of allocating DB2 resources to a plan in preparation for execution.

plan member

The bound copy of a DBRM that is identified in the member clause.

plan name

The name of an application plan.

P-lock See physical lock.

point of consistency

A time when all recoverable data that an application accesses is consistent with other data. The term point of consistency is synonymous with sync point or commit point.

policy See CFRM policy.

postponed abort UR

A unit of recovery that was in-flight or in-abort, was interrupted by system failure or cancellation, and did not complete backout during restart.

precision

In SQL, the total number of digits in a decimal number (called the *size* in the C language). In the C language, the number of digits to the right of the decimal point (called the *scale* in SQL). The DB2 information uses the SQL terms.

precompilation

A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

predicate

An element of a search condition that expresses or implies a comparison operation.

prefix A code at the beginning of a message or record.

preformat

The process of preparing a VSAM linear data set for DB2 use, by writing specific data patterns.

prepare

The first phase of a two-phase commit process in which all participants are requested to prepare for commit.

prepared SQL statement

A named object that is the executable

form of an SQL statement that has been processed by the PREPARE statement.

primary authorization ID

The authorization ID that is used to identify the application process to DB2.

primary group buffer pool

For a duplexed group buffer pool, the structure that is used to maintain the coherency of cached data. This structure is used for page registration and cross-invalidation. The z/OS equivalent is *old* structure. Compare with secondary group buffer pool.

primary index

An index that enforces the uniqueness of a primary key.

primary key

In a relational database, a unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

principal

An entity that can communicate securely with another entity. In Kerberos, principals are represented as entries in the Kerberos registry database and include users, servers, computers, and others.

principal name

The name by which a principal is known to the DCE security services.

privilege

The capability of performing a specific function, sometimes on a specific object. See also explicit privilege.

privilege set

- For the installation SYSADM ID, the set of all possible privileges.
- For any other authorization ID, including the PUBLIC authorization ID, the set of all privileges that are recorded for that ID in the DB2 catalog.

process

In DB2, the unit to which DB2 allocates resources and locks. Sometimes called an application process, a process involves the execution of one or more programs. The execution of an SQL statement is always associated with some process. The means of initiating and terminating a process are dependent on the environment.

program

A single, compilable collection of executable statements in a programming language.

program temporary fix (PTF)

A solution or bypass of a problem that is diagnosed as a result of a defect in a current unaltered release of a licensed program. An authorized program analysis report (APAR) fix is corrective service for an existing problem. A PTF is preventive service for problems that might be encountered by other users of the product. A PTF is *temporary*, because a permanent fix is usually not incorporated into the product until its next release.

protected conversation

A VTAM conversation that supports two-phase commit flows.

PSRCP

See page set recovery pending.

PTF See program temporary fix.

QSAM

See queued sequential access method.

query A component of certain SQL statements that specifies a result table.

query block

The part of a query that is represented by one of the FROM clauses. Each FROM clause can have multiple query blocks, depending on DB2 processing of the query.

query CP parallelism

Parallel execution of a single query, which is accomplished by using multiple tasks. See also Sysplex query parallelism.

query I/O parallelism

Parallel access of data, which is accomplished by triggering multiple I/O requests within a single query.

queued sequential access method (QSAM)

An extended version of the basic sequential access method (BSAM). When this method is used, a queue of data blocks is formed. Input data blocks await processing, and output data blocks await transfer to auxiliary storage or to an output device.

quiesce point

A point at which data is consistent as a result of running the DB2 QUIESCE utility.

RACF Resource Access Control Facility. A component of the z/OS Security Server.

range-partitioned table space

A type of universal table space that is based on partitioning ranges and that contains a single table. Contrast with partition-by-growth table space. See also universal table space.

RBA See relative byte address.

RCT See resource control table.

RDO See resource definition online.

read stability (RS)

An isolation level that is similar to repeatable read but does not completely isolate an application process from all other concurrently executing application processes. See also cursor stability, repeatable read, and uncommitted read.

rebind

The creation of a new application plan for an application program that has been bound previously. If, for example, you have added an index for a table that your application accesses, you must rebind the application in order to take advantage of that index.

rebuild

The process of reallocating a coupling facility structure. For the shared communications area (SCA) and lock structure, the structure is repopulated; for the group buffer pool, changed pages are usually cast out to disk, and the new structure is populated only with changed pages that were not successfully cast out.

record The storage representation of a row or other data.

record identifier (RID)

A unique identifier that DB2 uses to identify a row of data in a table. Compare with row identifier.

record identifier (RID) pool

An area of main storage that is used for sorting record identifiers during list-prefetch processing.

record length

The sum of the length of all the columns in a table, which is the length of the data as it is physically stored in the database. Records can be fixed length or varying length, depending on how the columns are defined. If all columns are fixed-length columns, the record is a fixed-length record. If one or more columns are varying-length columns, the record is a varying-length record.

Recoverable Resource Manager Services attachment facility (RRSAF)

A DB2 subcomponent that uses Resource Recovery Services to coordinate resource commitment between DB2 and all other resource managers that also use RRS in a z/OS system.

recovery

The process of rebuilding databases after a system failure.

recovery log

A collection of records that describes the events that occur during DB2 execution and indicates their sequence. The recorded information is used for recovery in the event of a failure during DB2 execution.

recovery manager

A subcomponent that supplies coordination services that control the interaction of DB2 resource managers during commit, abort, checkpoint, and restart processes. The recovery manager also supports the recovery mechanisms of other subsystems (for example, IMS) by acting as a participant in the other subsystem's process for protecting data that has reached a point of consistency.

A coordinator or a participant (or both), in the execution of a two-phase commit, that can access a recovery log that maintains the state of the logical unit of work and names the immediate upstream coordinator and downstream participants.

recovery pending (RECP)

A condition that prevents SQL access to a table space that needs to be recovered.

recovery token

An identifier for an element that is used in recovery (for example, NID or URID).

RECP See recovery pending.

redo A state of a unit of recovery that indicates that changes are to be reapplied to the disk media to ensure data integrity.

reentrant code

Executable code that can reside in storage as one shared copy for all threads. Reentrant code is not self-modifying and provides separate storage areas for each thread. See also threadsafe.

referential constraint

The requirement that nonnull values of a designated foreign key are valid only if they equal values of the primary key of a designated table.

referential cycle

A set of referential constraints such that each base table in the set is a descendent of itself. The tables that are involved in a referential cycle are ordered so that each table is a descendent of the one before it, and the first table is a descendent of the last table.

referential integrity

The state of a database in which all values of all foreign keys are valid. Maintaining referential integrity requires the enforcement of referential constraints on all operations that change the data in a table on which the referential constraints are defined.

referential structure

A set of tables and relationships that includes at least one table and, for every table in the set, all the relationships in which that table participates and all the tables to which it is related.

refresh age

The time duration between the current time and the time during which a materialized query table was last refreshed.

registry

See registry database.

registry database

A database of security information about principals, groups, organizations, accounts, and security policies.

relational database

A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

relational database management system (RDBMS)

A collection of hardware and software that organizes and provides access to a relational database.

relational schema

See SQL schema.

relationship

A defined connection between the rows of a table or the rows of two tables. A relationship is the internal representation of a referential constraint.

relative byte address (RBA)

The offset of a data record or control interval from the beginning of the storage space that is allocated to the data set or file to which it belongs.

remigration

The process of returning to a current release of DB2 following a fallback to a previous release. This procedure constitutes another migration process.

remote

Any object that is maintained by a remote DB2 subsystem (that is, by a DB2 subsystem other than the local one). A *remote view*, for example, is a view that is maintained by a remote DB2 subsystem. Contrast with local.

remote subsystem

Any relational DBMS, except the local subsystem, with which the user or application can communicate. The subsystem need not be remote in any physical sense, and might even operate on the same processor under the same z/OS system.

reoptimization

The DB2 process of reconsidering the access path of an SQL statement at run time; during reoptimization, DB2 uses the values of host variables, parameter markers, or special registers.

reordered row format

A row format that facilitates improved performance in retrieval of rows that have varying-length columns. DB2 rearranges the column order, as defined in the CREATE TABLE statement, so that the fixed-length columns are stored at the beginning of the row and the

varying-length columns are stored at the end of the row. Contrast with basic row format.

REORG pending (REORP)

A condition that restricts SQL access and most utility access to an object that must be reorganized.

REORP

See REORG pending.

repeatable read (RR)

The isolation level that provides maximum protection from other executing application programs. When an application program executes with repeatable read protection, rows that the program references cannot be changed by other programs until the program reaches a commit point. See also cursor stability, read stability, and uncommitted read.

repeating group

A situation in which an entity includes multiple attributes that are inherently the same. The presence of a repeating group violates the requirement of first normal form. In an entity that satisfies the requirement of first normal form, each attribute is independent and unique in its meaning and its name. See also normalization.

replay detection mechanism

A method that allows a principal to detect whether a request is a valid request from a source that can be trusted or whether an untrustworthy entity has captured information from a previous exchange and is replaying the information exchange to gain access to the principal.

request commit

The vote that is submitted to the prepare phase if the participant has modified data and is prepared to commit or roll back.

requester

The source of a request to access data at a remote server. In the DB2 environment, the requester function is provided by the distributed data facility.

resource

The object of a lock or claim, which could be a table space, an index space, a data partition, an index partition, or a logical partition.

resource allocation

The part of plan allocation that deals specifically with the database resources.

resource control table

A construct of previous versions of the CICS attachment facility that defines authorization and access attributes for transactions or transaction groups. Beginning in CICS Transaction Server Version 1.3, resources are defined by using resource definition online instead of the resource control table. See also resource definition online.

resource definition online (RDO)

The recommended method of defining resources to CICS by creating resource definitions interactively, or by using a utility, and then storing them in the CICS definition data set. In earlier releases of CICS, resources were defined by using the resource control table (RCT), which is no longer supported.

resource limit facility (RLF)

A portion of DB2 code that prevents dynamic manipulative SQL statements from exceeding specified time limits. The resource limit facility is sometimes called the governor.

resource limit specification table (RLST)

A site-defined table that specifies the limits to be enforced by the resource limit facility.

resource manager

- A function that is responsible for managing a particular resource and that guarantees the consistency of all updates made to recoverable resources within a logical unit of work. The resource that is being managed can be physical (for example, disk or main storage) or logical (for example, a particular type of system service).
- A participant, in the execution of a two-phase commit, that has recoverable resources that could have been modified. The resource manager has access to a recovery log so that it can commit or roll back the effects of the logical unit of work to the recoverable resources.

restart pending (RESTP)

A restrictive state of a page set or

partition that indicates that restart (backout) work needs to be performed on the object.

RESTP

See restart pending.

result set

The set of rows that a stored procedure returns to a client application.

result set locator

A 4-byte value that DB2 uses to uniquely identify a query result set that a stored procedure returns.

result table

The set of rows that are specified by a SELECT statement.

retained lock

A MODIFY lock that a DB2 subsystem was holding at the time of a subsystem failure. The lock is retained in the coupling facility lock structure across a DB2 for z/OS failure.

RID See record identifier.

RID pool

See record identifier pool.

right outer join

The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of the second join operand. See also join, equijoin, full outer join, inner join, left outer join, and outer join.

RLF See resource limit facility.

RLST See resource limit specification table.

role A database entity that groups together one or more privileges and that can be assigned to a primary authorization ID or to PUBLIC. The role is available only in a trusted context.

rollback

The process of restoring data that was changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with commit.

root page

The index page that is at the highest level (or the beginning point) in an index.

routine

A database object that encapsulates

	procedural logic and SQL statements, is stored on the database server, and can be invoked from an SQL statement or by using the CALL statement. The main classes of routines are procedures and functions.	SBCS	See single-byte character set.
		SCA	See shared communications area.
		scalar function	An SQL operation that produces a single value from another value and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses.
		scale	In SQL, the number of digits to the right of the decimal point (called the precision in the C language). The DB2 information uses the SQL definition.
		schema	The organization or structure of a database. A collection of, and a way of qualifying, database objects such as tables, views, routines, indexes or triggers that define a database. A database schema provides a logical classification of database objects.
		scrollability	The ability to use a cursor to fetch in either a forward or backward direction. The FETCH statement supports multiple fetch orientations to indicate the new position of the cursor. See also fetch orientation.
		scrollable cursor	A cursor that can be moved in both a forward and a backward direction.
		search condition	A criterion for selecting rows from a table. A search condition consists of one or more predicates.
		secondary authorization ID	An authorization ID that has been associated with a primary authorization ID by an authorization exit routine.
		secondary group buffer pool	For a duplexed group buffer pool, the structure that is used to back up changed pages that are written to the primary group buffer pool. No page registration or cross-invalidation occurs using the secondary group buffer pool. The z/OS equivalent is <i>new</i> structure.
		secondary index	A nonpartitioning index that is useful for enforcing a uniqueness constraint, for clustering data, or for providing access
		row	The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.
		row identifier (ROWID)	A value that uniquely identifies a row. This value is stored with the row and never changes.
		row lock	A lock on a single row of data.
		row-positioned fetch orientation	The specification of the desired placement of the cursor as part of a FETCH statement, with respect to a single row (for example, NEXT, LAST, or ABSOLUTE <i>n</i>). Contrast with rowset-positioned fetch orientation.
		rowset	A set of rows for which a cursor position is established.
		rowset cursor	A cursor that is defined so that one or more rows can be returned as a rowset for a single FETCH statement, and the cursor is positioned on the set of rows that is fetched.
		rowset-positioned fetch orientation	The specification of the desired placement of the cursor as part of a FETCH statement, with respect to a rowset (for example, NEXT ROWSET, LAST ROWSET, or ROWSET STARTING AT ABSOLUTE <i>n</i>). Contrast with row-positioned fetch orientation.
		row trigger	A trigger that is defined with the trigger granularity FOR EACH ROW.
		RRSAF	See Recoverable Resource Manager Services attachment facility.
		RS	See read stability.
		savepoint	A named entity that represents the state of data and schemas at a particular point in time within a unit of work.

paths to data for queries. A secondary index can be partitioned or nonpartitioned. See also data-partitioned secondary index (DPSI) and nonpartitioned secondary index (NPSI).

section

The segment of a plan or package that contains the executable structures for a single SQL statement. For most SQL statements, one section in the plan exists for each SQL statement in the source program. However, for cursor-related statements, the DECLARE, OPEN, FETCH, and CLOSE statements reference the same section because they each refer to the SELECT statement that is named in the DECLARE CURSOR statement. SQL statements such as COMMIT, ROLLBACK, and some SET statements do not use a section.

security label

A classification of users' access to objects or data rows in a multilevel security environment."

segment

A group of pages that holds rows of a single table. See also segmented table space.

segmented table space

A table space that is divided into equal-sized groups of pages called segments. Segments are assigned to tables so that rows of different tables are never stored in the same segment. Contrast with partitioned table space and universal table space.

self-referencing constraint

A referential constraint that defines a relationship in which a table is a dependent of itself.

self-referencing table

A table with a self-referencing constraint.

sensitive cursor

A cursor that is sensitive to changes that are made to the database after the result table has been materialized.

sequence

A user-defined object that generates a sequence of numeric values according to user specifications.

sequential data set

A non-DB2 data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Several of the DB2 database utilities require sequential data sets.

sequential prefetch

A mechanism that triggers consecutive asynchronous I/O operations. Pages are fetched before they are required, and several pages are read with a single I/O operation.

serialized profile

A Java object that contains SQL statements and descriptions of host variables. The SQLJ translator produces a serialized profile for each connection context.

server The target of a request from a remote requester. In the DB2 environment, the server function is provided by the distributed data facility, which is used to access DB2 data from remote applications.

service class

An eight-character identifier that is used by the z/OS Workload Manager to associate user performance goals with a particular DDF thread or stored procedure. A service class is also used to classify work on parallelism assistants.

service request block

A unit of work that is scheduled to execute.

session

A link between two nodes in a VTAM network.

session protocols

The available set of SNA communication requests and responses.

set operator

The SQL operators UNION, EXCEPT, and INTERSECT corresponding to the relational operators union, difference, and intersection. A set operator derives a result table by combining two other result tables.

shared communications area (SCA)

A coupling facility list structure that a DB2 data sharing group uses for inter-DB2 communication.

share lock

A lock that prevents concurrently executing application processes from changing data, but not from reading data. Contrast with exclusive lock.

shift-in character

A special control character (X'0F') that is used in EBCDIC systems to denote that the subsequent bytes represent SBCS characters. See also shift-out character.

shift-out character

A special control character (X'0E') that is used in EBCDIC systems to denote that the subsequent bytes, up to the next shift-in control character, represent DBCS characters. See also shift-in character.

sign-on

A request that is made on behalf of an individual CICS or IMS application process by an attachment facility to enable DB2 to verify that it is authorized to use DB2 resources.

simple page set

A nonpartitioned page set. A simple page set initially consists of a single data set (page set piece). If and when that data set is extended to 2 GB, another data set is created, and so on, up to a total of 32 data sets. DB2 considers the data sets to be a single contiguous linear address space containing a maximum of 64 GB. Data is stored in the next available location within this address space without regard to any partitioning scheme.

simple table space

A table space that is neither partitioned nor segmented. Creation of simple table spaces is not supported in DB2 Version 9.1 for z/OS. Contrast with partitioned table space, segmented table space, and universal table space.

single-byte character set (SBCS)

A set of characters in which each character is represented by a single byte. Contrast with double-byte character set or multibyte character set.

single-precision floating point number

A 32-bit approximate representation of a real number.

SMP/E

See System Modification Program/Extended.

SNA See Systems Network Architecture.

SNA network

The part of a network that conforms to the formats and protocols of Systems Network Architecture (SNA).

socket A callable TCP/IP programming interface that TCP/IP network applications use to communicate with remote TCP/IP partners.

sourced function

A function that is implemented by another built-in or user-defined function that is already known to the database manager. This function can be a scalar function or an aggregate function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with built-in function, external function, and SQL function.

source program

A set of host language statements and SQL statements that is processed by an SQL precompiler.

source table

A table that can be a base table, a view, a table expression, or a user-defined table function.

source type

An existing type that DB2 uses to represent a distinct type.

space A sequence of one or more blank characters.

special register

A storage area that DB2 defines for an application process to use for storing information that can be referenced in SQL statements. Examples of special registers are SESSION_USER and CURRENT DATE.

specific function name

A particular user-defined function that is known to the database manager by its specific name. Many specific user-defined functions can have the same function name. When a user-defined function is defined to the database, every function is assigned a specific name that is unique within its schema. Either the user can provide this name, or a default name is used.

SPUFI See SQL Processor Using File Input.

SQL See Structured Query Language.

SQL authorization ID (SQL ID)

The authorization ID that is used for checking dynamic SQL statements in some situations.

SQLCA

See SQL communication area.

SQL communication area (SQLCA)

A structure that is used to provide an application program with information about the execution of its SQL statements.

SQL connection

An association between an application process and a local or remote application server or database server.

SQLDA

See SQL descriptor area.

SQL descriptor area (SQLDA)

A structure that describes input variables, output variables, or the columns of a result table.

SQL escape character

The symbol that is used to enclose an SQL delimited identifier. This symbol is the double quotation mark ("). See also escape character.

SQL function

A user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL user-defined function can return the result of an expression. See also built-in function, external function, and sourced function.

SQL ID

See SQL authorization ID.

SQLJ Structured Query Language (SQL) that is embedded in the Java programming language.

SQL path

An ordered list of schema names that are used in the resolution of unqualified references to user-defined functions, distinct types, and stored procedures. In dynamic SQL, the SQL path is found in the CURRENT PATH special register. In static SQL, it is defined in the PATH bind option.

SQL PL

See SQL procedural language.

SQL procedural language (SQL PL)

A language extension of SQL that consists of statements and language elements that can be used to implement procedural logic in SQL statements. SQL PL provides statements for declaring variables and condition handlers, assigning values to variables, and for implementing procedural logic. See also inline SQL PL.

SQL procedure

A user-written program that can be invoked with the SQL CALL statement. An SQL procedure is written in the SQL procedural language. Two types of SQL procedures are supported: external SQL procedures and native SQL procedures. See also external procedure and native SQL procedure.

SQL processing conversation

Any conversation that requires access of DB2 data, either through an application or by dynamic query requests.

SQL Processor Using File Input (SPUFI)

A facility of the TSO attachment subcomponent that enables the DB2I user to execute SQL statements without embedding them in an application program.

SQL return code

Either SQLCODE or SQLSTATE.

SQL routine

A user-defined function or stored procedure that is based on code that is written in SQL.

SQL schema

A collection of database objects such as tables, views, indexes, functions, distinct types, schemas, or triggers that defines a database. An SQL schema provides a logical classification of database objects.

SQL statement coprocessor

An alternative to the DB2 precompiler that lets the user process SQL statements at compile time. The user invokes an SQL statement coprocessor by specifying a compiler option.

SQL string delimiter

A symbol that is used to enclose an SQL string constant. The SQL string delimiter

is the apostrophe ('), except in COBOL applications, where the user assigns the symbol, which is either an apostrophe or a double quotation mark (").

SRB See service request block.

stand-alone

An attribute of a program that means that it is capable of executing separately from DB2, without using DB2 services.

star join

A method of joining a dimension column of a fact table to the key column of the corresponding dimension table. See also join, dimension, and star schema.

star schema

The combination of a fact table (which contains most of the data) and a number of dimension tables. See also star join, dimension, and dimension table.

statement handle

In DB2 ODBC, the data object that contains information about an SQL statement that is managed by DB2 ODBC. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values, and status information. Each statement handle is associated with the connection handle.

statement string

For a dynamic SQL statement, the character string form of the statement.

statement trigger

A trigger that is defined with the trigger granularity FOR EACH STATEMENT.

static cursor

A named control structure that does not change the size of the result table or the order of its rows after an application opens the cursor. Contrast with dynamic cursor.

static SQL

SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of variables that are specified by the statement might change).

storage group

A set of storage objects on which DB2 for z/OS data can be stored. A storage object can have an SMS data class, a management class, a storage class, and a list of volume serial numbers.

stored procedure

A user-written application program that can be invoked through the use of the SQL CALL statement. Stored procedures are sometimes called procedures.

string See binary string, character string, or graphic string.

strong typing

A process that guarantees that only user-defined functions and operations that are defined on a distinct type can be applied to that type. For example, you cannot directly compare two currency types, such as Canadian dollars and U.S. dollars. But you can provide a user-defined function to convert one currency to the other and then do the comparison.

structure

- A name that refers collectively to different types of DB2 objects, such as tables, databases, views, indexes, and table spaces.
- A construct that uses z/OS to map and manage storage on a coupling facility. See also cache structure, list structure, or lock structure.

Structured Query Language (SQL)

A standardized language for defining and manipulating data in a relational database.

structure owner

In relation to group buffer pools, the DB2 member that is responsible for the following activities:

- Coordinating rebuild, checkpoint, and damage assessment processing
- Monitoring the group buffer pool threshold and notifying castout owners when the threshold has been reached

subcomponent

A group of closely related DB2 modules that work together to provide a general function.

subject table

The table for which a trigger is created. When the defined triggering event occurs on this table, the trigger is activated.

subquery

A SELECT statement within the WHERE or HAVING clause of another SQL statement; a nested SQL statement.

subselect

That form of a query that includes only a SELECT clause, FROM clause, and optionally a WHERE clause, GROUP BY clause, HAVING clause, ORDER BY clause, or FETCH FIRST clause.

substitution character

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

subsystem

In z/OS, a service provider that performs one or many functions, but does nothing until a request is made. For example, each WebSphere MQ for z/OS queue manager or instance of a DB2 for z/OS database management system is a z/OS subsystem.

surrogate pair

A coded representation for a single character that consists of a sequence of two 16-bit code units, in which the first value of the pair is a high-surrogate code unit in the range U+D800 through U+DBFF, and the second value is a low-surrogate code unit in the range U+DC00 through U+DFFF. Surrogate pairs provide an extension mechanism for encoding 917 476 characters without requiring the use of 32-bit characters.

SVC dump

A dump that is issued when a z/OS or a DB2 functional recovery routine detects an error.

sync point

See commit point.

syncpoint tree

The tree of recovery managers and resource managers that are involved in a logical unit of work, starting with the recovery manager, that make the final commit decision.

synonym

In SQL, an alternative name for a table or view. Synonyms can be used to refer only to objects at the subsystem in which the synonym is defined. A synonym cannot be qualified and can therefore not be used by other users. Contrast with alias.

Sysplex

See Parallel Sysplex.

Sysplex query parallelism

Parallel execution of a single query that is accomplished by using multiple tasks on more than one DB2 subsystem. See also query CP parallelism.

system administrator

The person at a computer installation who designs, controls, and manages the use of the computer system.

system agent

A work request that DB2 creates such as prefetch processing, deferred writes, and service tasks. See also allied agent.

system authorization ID

The primary DB2 authorization ID that is used to establish a trusted connection. A system authorization ID is derived from the system user ID that is provided by an external entity, such as a middleware server.

system conversation

The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

system-defined routine

In DB2 10 for z/OS and later, an object (function or procedure) for which system DBADM and SQLADM authorities have implicit execute privilege on the routine and any packages executed within the routine.

System Modification Program/Extended (SMP/E)

A z/OS tool for making software changes in programming systems (such as DB2) and for controlling those changes.

system period

A pair of columns with system-maintained values that indicate the period of time when a row is valid.

system-period data versioning

Automatic maintenance of historical data by DB2 using a system period.

system-period temporal table

A table that is defined with system-period data versioning.

Systems Network Architecture (SNA)

The description of the logical structure, formats, protocols, and operational sequences for transmitting information through and controlling the configuration and operation of networks.

table A named data object consisting of a specific number of columns and some number of unordered rows. See also base table or temporary table. Contrast with auxiliary table, clone table, materialized query table, result table, and transition table.

table-controlled partitioning

A type of partitioning in which partition boundaries for a partitioned table are controlled by values that are defined in the CREATE TABLE statement.

table function

A function that receives a set of arguments and returns a table to the SQL statement that references the function. A table function can be referenced only in the FROM clause of a subselect.

table locator

A mechanism that allows access to trigger tables in SQL or from within user-defined functions. A table locator is a fullword integer value that represents a transition table.

table space

A page set that is used to store the records in one or more tables. See also partitioned table space, segmented table space, and universal table space.

table space set

A set of table spaces and partitions that should be recovered together for one of the following reasons:

- Each of them contains a table that is a parent or descendent of a table in one of the others.
- The set contains a base table and associated auxiliary tables.

A table space set can contain both types of relationships.

task control block (TCB)

A z/OS control block that is used to communicate information about tasks within an address space that is connected to a subsystem. See also address space connection.

TB Terabyte. A value of 1 099 511 627 776 bytes.

TCB See task control block.

TCP/IP

A network communication protocol that computer systems use to exchange information across telecommunication links.

TCP/IP port

A 2-byte value that identifies an end user or a TCP/IP network application within a TCP/IP host.

template

A DB2 utilities output data set descriptor that is used for dynamic allocation. A template is defined by the TEMPLATE utility control statement.

temporal table

A table which records the period of time when a row is valid. See also system-period temporal table, application-period temporal table, and bitemporal table.

temporary table

A table that holds temporary data. Temporary tables are useful for holding or sorting intermediate results from queries that contain a large number of rows. The two types of temporary table, which are created by different SQL statements, are the created temporary table and the declared temporary table. Contrast with result table. See also created temporary table and declared temporary table.

textual XML format

A system of storing XML data in text, as opposed to binary, that allows for direct human reading.

thread See DB2 thread.

threadsafe

A characteristic of code that allows

multithreading both by providing private storage areas for each thread, and by properly serializing shared (global) storage areas.

three-part name

The full name of a table, view, or alias. It consists of a location name, a schema name, and an object name, separated by a period.

time A three-part value that designates a time of day in hours, minutes, and seconds.

timeout

Abnormal termination of either the DB2 subsystem or of an application because of the unavailability of resources. Installation specifications are set to determine both the amount of time DB2 is to wait for IRLM services after starting, and the amount of time IRLM is to wait if a resource that an application requests is unavailable. If either of these time specifications is exceeded, a timeout is declared.

Time-Sharing Option (TSO)

An option in z/OS that provides interactive time sharing from remote terminals.

timestamp

A seven-part value that consists of a date and time. The timestamp is expressed in years, months, days, hours, minutes, seconds, and microseconds.

timestamp with time zone

A two-part value that consists of a timestamp and time zone. The timestamp with time zone is expressed in years, months, days, hours, minutes, seconds, microseconds, time zone hours, and time zone minutes.

trace A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

transaction

An atomic series of SQL statements that make up a logical unit of work. All of the data modifications made during a transaction are either committed together as a unit or rolled back as a unit.

transaction lock

A lock that is used to control concurrent execution of SQL statements.

transaction program name

In SNA LU 6.2 conversations, the name of the program at the remote logical unit that is to be the other half of the conversation.

transition table

A temporary table that contains all the affected rows of the subject table in their state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the table of changed rows in the old state or the new state. Contrast with auxiliary table, base table, clone table, and materialized query table.

transition variable

A variable that contains a column value of the affected row of the subject table in its state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the set of old values or the set of new values.

tree structure

A data structure that represents entities in nodes, with a most one parent node for each node, and with only one root node.

trigger

A database object that is associated with a single base table or view and that defines a rule. The rule consists of a set of SQL statements that run when an insert, update, or delete database operation occurs on the associated base table or view.

trigger activation

The process that occurs when the trigger event that is defined in a trigger definition is executed. Trigger activation consists of the evaluation of the triggered action condition and conditional execution of the triggered SQL statements.

trigger activation time

An indication in the trigger definition of whether the trigger should be activated before or after the triggered event.

trigger body

The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to

true. A trigger body is also called triggered SQL statements.

trigger cascading

The process that occurs when the triggered action of a trigger causes the activation of another trigger.

triggered action

The SQL logic that is performed when a trigger is activated. The triggered action consists of an optional triggered action condition and a set of triggered SQL statements that are executed only if the condition evaluates to true.

triggered action condition

An optional part of the triggered action. This Boolean condition appears as a WHEN clause and specifies a condition that DB2 evaluates to determine if the triggered SQL statements should be executed.

triggered SQL statements

The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. Triggered SQL statements are also called the trigger body.

trigger granularity

In SQL, a characteristic of a trigger, which determines whether the trigger is activated:

- Only once for the triggering SQL statement
- Once for each row that the SQL statement modifies

triggering event

The specified operation in a trigger definition that causes the activation of that trigger. The triggering event is comprised of a triggering operation (insert, update, or delete) and a subject table or view on which the operation is performed.

triggering SQL operation

The SQL operation that causes a trigger to be activated when performed on the subject table or view.

trigger package

A package that is created when a CREATE TRIGGER statement is executed. The package is executed when the trigger is activated.

trust attribute

An attribute on which to establish trust. A trusted relationship is established based on one or more trust attributes.

trusted connection

A database connection whose attributes match the attributes of a unique trusted context defined at the DB2 database server.

trusted connection reuse

The ability to switch the current user ID on a trusted connection to a different user ID.

trusted context

A database security object that enables the establishment of a trusted relationship between a DB2 database management system and an external entity.

trusted context default role

A role associated with a trusted context. The privileges granted to the trusted context default role can be acquired only when a trusted connection based on the trusted context is established or reused.

trusted context user

A user ID to which switching the current user ID on a trusted connection is permitted.

trusted context user-specific role

A role that is associated with a specific trusted context user. It overrides the trusted context default role if the current user ID on the trusted connection matches the ID of the specific trusted context user.

trusted relationship

A privileged relationship between two entities such as a middleware server and a database server. This relationship allows for a unique set of interactions between the two entities that would be impossible otherwise.

TSO See Time-Sharing Option.

TSO attachment facility

A DB2 facility consisting of the DSN command processor and DB2I. Applications that are not written for the CICS or IMS environments can run under the TSO attachment facility.

typed parameter marker

A parameter marker that is specified along with its target data type. It has the general form:

CAST(? AS data-type)

type 2 indexes

Indexes that are created on a release of DB2 after Version 7 or that are specified as type 2 indexes in Version 4 or later.

UCS-2 Universal Character Set, coded in 2 octets, which means that characters are represented in 16-bits per character.

UDF See user-defined function.

UDT User-defined data type. In DB2 for z/OS, the term distinct type is used instead of user-defined data type. See distinct type.

uncommitted read (UR)

The isolation level that allows an application to read uncommitted data. See also cursor stability, read stability, and repeatable read.

underlying view

The view on which another view is directly or indirectly defined.

undo A state of a unit of recovery that indicates that the changes that the unit of recovery made to recoverable DB2 resources must be backed out.

Unicode

A standard that parallels the ISO-10646 standard. Several implementations of the Unicode standard exist, all of which have the ability to represent a large percentage of the characters that are contained in the many scripts that are used throughout the world.

| **union** An SQL operation that involves the
| UNION set operator, which combines the
| results of two SELECT statements. Unions
| are often used to merge lists of values
| that are obtained from two tables.

unique constraint

An SQL rule that no two values in a primary key, or in the key of a unique index, can be the same.

unique index

An index that ensures that no identical key values are stored in a column or a set of columns in a table.

unit of recovery (UOR)

A recoverable sequence of operations within a single resource manager, such as an instance of DB2. Contrast with unit of work.

unit of work (UOW)

A recoverable sequence of operations within an application process. At any time, an application process is a single unit of work, but the life of an application process can involve many units of work as a result of commit or rollback operations. In a multisite update operation, a single unit of work can include several *units of recovery*. Contrast with unit of recovery.

| **universal table space**

| A table space that is both segmented and
| partitioned. Contrast with partitioned
| table space, segmented table space,
| partition-by-growth table space, and
| range-partitioned table space.

unlock

The act of releasing an object or system resource that was previously locked and returning it to general availability within DB2.

untyped parameter marker

A parameter marker that is specified without its target data type. It has the form of a single question mark (?).

updatability

The ability of a cursor to perform positioned updates and deletes. The updatability of a cursor can be influenced by the SELECT statement and the cursor sensitivity option that is specified on the DECLARE CURSOR statement.

update hole

The location on which a cursor is positioned when a row in a result table is fetched again and the new values no longer satisfy the search condition. See also delete hole.

update trigger

A trigger that is defined with the triggering SQL operation update.

UR See uncommitted read.

user-defined data type (UDT)

See distinct type.

user-defined function (UDF)

A function that is defined to DB2 by using the CREATE FUNCTION statement and that can be referenced thereafter in SQL statements. A user-defined function can be an external function, a sourced function, or an SQL function. Contrast with built-in function.

user view

In logical data modeling, a model or representation of critical information that the business requires.

UTF-16

Unicode Transformation Format, 16-bit encoding form, which is designed to provide code values for over a million characters and a superset of UCS-2. The CCSID value for data in UTF-16 format is 1200. DB2 for z/OS supports UTF-16 in graphic data fields.

UTF-8 Unicode Transformation Format, 8-bit encoding form, which is designed for ease of use with existing ASCII-based systems. The CCSID value for data in UTF-8 format is 1208. DB2 for z/OS supports UTF-8 in mixed data fields.

value The smallest unit of data that is manipulated in SQL.

variable

A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a host variable. Contrast with constant.

variant function

See nondeterministic function.

varying-length string

A character, graphic, or binary string whose length varies within set limits. Contrast with fixed-length string.

version

A member of a set of similar programs, DBRMs, packages, or LOBs.

- **A version of a program** is the source code that is produced by precompiling the program. The program version is identified by the program name and a timestamp (consistency token).
- **A version of an SQL procedural language routine** is produced by

issuing the CREATE or ALTER PROCEDURE statement for a native SQL procedure.

- **A version of a DBRM** is the DBRM that is produced by precompiling a program. The DBRM version is identified by the same program name and timestamp as a corresponding program version.
- **A version of an application package** is the result of binding a DBRM within a particular database system. The application package version is identified by the same program name and consistency token as the DBRM.
- **A version of a LOB** is a copy of a LOB value at a point in time. The version number for a LOB is stored in the auxiliary index entry for the LOB.
- **A version of a record** is a copy of the record at a point in time.

view A logical table that consists of data that is generated by a query. A view can be based on one or more underlying base tables or views, and the data in a view is determined by a SELECT statement that is run on the underlying base tables or views.

Virtual Storage Access Method (VSAM)

An access method for direct or sequential processing of fixed- and varying-length records on disk devices.

Virtual Telecommunications Access Method (VTAM)

An IBM licensed program that controls communication and the flow of data in an SNA network (in z/OS).

volatile table

A table for which SQL operations choose index access whenever possible.

VSAM

See Virtual Storage Access Method.

VTAM

See Virtual Telecommunications Access Method.

warm start

The normal DB2 restart process, which involves reading and processing log records so that data that is under the control of DB2 is consistent. Contrast with cold start.

WLM application environment

A z/OS Workload Manager attribute that is associated with one or more stored procedures. The WLM application environment determines the address space in which a given DB2 stored procedure runs.

WLM enclave

A construct that can span multiple dispatchable units (service request blocks and tasks) in multiple address spaces, allowing them to be reported on and managed by WLM as part of a single work request.

write to operator (WTO)

An optional user-coded service that allows a message to be written to the system console operator informing the operator of errors and unusual system conditions that might need to be corrected (in z/OS).

WTO See write to operator.

WTOR

Write to operator (WTO) with reply.

XCF See cross-system coupling facility.

XES See cross-system extended services.

XML See Extensible Markup Language.

XML attribute

A name-value pair within a tagged XML element that modifies certain features of the element.

XML column

A column of a table that stores XML values and is defined using the data type XML. The XML values that are stored in XML columns are internal representations of well-formed XML documents.

XML data type

A data type for XML values.

XML element

A logical structure in an XML document that is delimited by a start and an end tag. Anything between the start tag and the end tag is the content of the element.

XML index

An index on an XML column that provides efficient access to nodes within an XML document by providing index keys that are based on XML patterns.

XML lock

A column-level lock for XML data. The operation of XML locks is similar to the operation of LOB locks.

XML node

The smallest unit of valid, complete structure in a document. For example, a node can represent an element, an attribute, or a text string.

XML node ID index

An implicitly created index, on an XML table that provides efficient access to XML documents and navigation among multiple XML data rows in the same document.

XML pattern

A slash-separated list of element names, an optional attribute name (at the end), or kind tests, that describe a path within an XML document in an XML column. The pattern is a restrictive form of path expressions, and it selects nodes that match the specifications. XML patterns are specified to create indexes on XML columns in a database.

XML publishing function

A function that returns an XML value from SQL values. An XML publishing function is also known as an XML constructor.

XML schema

In XML, a mechanism for describing and constraining the content of XML files by indicating which elements are allowed and in which combinations. XML schemas are an alternative to document type definitions (DTDs) and can be used to extend functionality in the areas of data typing, inheritance, and presentation.

XML schema repository (XSR)

A repository that allows the DB2 database system to store XML schemas. When registered with the XSR, these objects have a unique identifier and can be used to validate XML instance documents.

XML serialization function

A function that returns a serialized XML string from an XML value.

XML table

An auxiliary table that is implicitly created when an XML column is added to

| a base table. This table stores the XML
| data, and the column in the base table
| points to it.

| **XML table space**

A table space that is implicitly created when an XML column is added to a base table. The table space stores the XML table. If the base table is partitioned, one partitioned table space exists for each XML column of data.

X/Open

An independent, worldwide open systems organization that is supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal is to increase the portability of applications by combining existing and emerging standards.

XRF See Extended Recovery Facility.

| **XSR** See XML schema repository.

| **zIIP** See IBM System z9 Integrated Processor.

z/OS An operating system for the System z product line that supports 64-bit real and virtual storage.

z/OS Distributed Computing Environment (z/OS DCE) A set of technologies that are provided by the Open Software Foundation to implement distributed computing.

Index

Special characters

- _ (underscore character) as escape character 237
- (minus sign) 182
- , (comma) as decimal point 251
- : (colon)
 - preceding a host variable 160
- ! (exclamation mark) as not sign 224
- ? (question mark) 1276
- / (divide sign) 182
- . (period) as decimal point 251
- * (asterisk)
 - COUNT function 267
 - COUNT_BIG function 267
 - multiply sign 182
 - use in subselect 633
- % (percent sign) as escape character 237
- || (vertical bars) 189
- + (plus sign) 182
- + (plus sign) as escape character 237

A

- ABS function 284
- ABSOLUTE clause
 - FETCH statement 1296
- ABSVAL function 284
- accelerator tables
 - SYSACCELERATORS 1997
 - SYSACCELIPLIST 1998
- accelerators tables
 - indexes 1996
 - table space 1996
- accessibility
 - keyboard xviii
 - shortcut keys xviii
- ACCESSPATH column
 - SYSPACKSTMT catalog table 1823
 - SYSSTMT catalog table 1868
- ACOS function 285
- ACQUIRE
 - column of SYSPLAN catalog table 1832
- ACTIVATE VERSION clause
 - ALTER PROCEDURE (SQL - native) statement 762
- ACTIVE column
 - SYSROUTINES catalog table 1848
- active log 17
- ACTIVE VERSION clause
 - ALTER PROCEDURE (SQL - native) statement 761
- ADD ATTRIBUTES clause
 - ALTER TRUSTED CONTEXT statement 859
- ADD clause
 - ALTER TABLE statement 799
- ADD CLONE clause
 - ALTER TABLE statement 829
- ADD COLUMN clause
 - ALTER INDEX statement 736
- ADD MATERIALIZED QUERY clause
 - ALTER TABLE statement 826
- ADD PARTITION clause
 - ALTER TABLE statement 821

- ADD RESTRICT ON DROP clause
 - ALTER TABLE statement 830
- ADD USE FOR clause
 - ALTER TRUSTED CONTEXT statement 861
- ADD VERSION clause
 - ALTER PROCEDURE (SQL - native) statement 762
- ADD VOLUMES clause of ALTER STOGROUP statement 787
- ADD_MONTHS function 286
- ADDRESS clause
 - ALTER TRUSTED CONTEXT statement 859
 - CREATE TRUSTED CONTEXT statement 1170
- ADMIN_TASK_LIST function 601
- ADMIN_TASK_STATUS function 606
- AFTER clause
 - FETCH statement 1293
- AFTER clause of CREATE TRIGGER statement 1153
- alias
 - creating 905
 - description 59
 - dropping 1259
 - naming convention 51
 - qualifying a column name 154
 - retrieving catalog information about 2000
 - unqualified name 58
- ALIAS clause
 - COMMENT statement 890
 - CREATE ALIAS statement 905
 - DROP statement 1259
 - LABEL statement 1384
- ALL
 - clause of RELEASE statement 1424
 - clause of subselect 633
 - keyword
 - aggregate functions 267
 - AVG function 269
 - COUNT function 271
 - COUNT_BIG function 272
 - MAX function 275
 - MIN function 276
 - STDDEV function 277
 - STDDEV_SAMP function 277
 - SUM function 278
 - VARIANCE function 279
 - VARIANCE_SAMP function 279
 - quantified predicate 226
- ALL PRIVILEGES clause
 - GRANT statement 1355
 - REVOKE statement 1458
- ALL SQL clause of RELEASE statement 1424
- ALLOCATE CURSOR statement
 - description 696
 - example 696
- ALLOW DEBUG MODE clause
 - ALTER PROCEDURE (external) statement 750
 - ALTER PROCEDURE (SQL - native) statement 764
 - CREATE PROCEDURE (external) statement 1023
 - CREATE PROCEDURE (SQL - native) statement 1052
- ALLOW PARALLEL clause
 - ALTER FUNCTION statement 714
 - CREATE FUNCTION statement 932

- ALLOWPUBLIC column
 - SYSCONTEXT catalog table 1737
- alphabetic extender 47
- ALTDATE function 2009
- ALTER ATTRIBUTES clause
 - ALTER TRUSTED CONTEXT statement 859
- ALTER clause
 - ALTER TRUSTED CONTEXT statement 857
- ALTER COLUMN clause
 - ALTER TABLE statement 807
- ALTER DATABASE statement
 - description 698
 - example 698
- ALTER FUNCTION (external scalar) statement
 - example 717
- ALTER FUNCTION (external) statement
 - description 701
- ALTER FUNCTION (SQL scalar) statement
 - description 719
 - examples 724
- ALTER INDEX statement
 - description 725
 - example 739
- ALTER MATERIALIZED QUERY clause
 - ALTER TABLE statement 828
- ALTER PARTITION
 - clause of ALTER INDEX statement 737
 - clause of CREATE TABLESPACE statement 1140
- ALTER PARTITION clause
 - ALTER TABLE statement 823
 - ALTER TABLESPACE statement 851
- ALTER privilege
 - GRANT statement 1351, 1355
 - REVOKE statement 1453, 1458
- ALTER PROCEDURE (external) statement
 - description 741
 - example 751
- ALTER PROCEDURE (SQL - external) statement
 - description 752
 - example 757
- ALTER PROCEDURE (SQL - native) statement
 - description 758
 - examples 779
- ALTER SEQUENCE statement
 - description 781
 - example 785
- ALTER STOGROUP statement
 - description 786
 - example 788
- ALTER TABLE statement
 - alternative syntax 839
 - description 789
 - examples 839
- ALTER TABLESPACE statement
 - description 841
 - example 854
- ALTER TRUSTED CONTEXT statement
 - description 855
 - examples 866
 - usage notes 864
- ALTER VERSION clause
 - ALTER PROCEDURE (SQL - native) statement 762
- ALTER VIEW statement
 - description 867
- ALTERAUTH
 - column of SYSSEQUENCEAUTH catalog table 1863
- ALTERAUTH column of SYSTABAUTH catalog table 1878
- ALTEREDTS
 - SYSSTOGROUP catalog table 1872
- ALTEREDTS column
 - SYSCOLUMNS catalog table 1723
 - SYSCONTEXT catalog table 1737
 - SYSDATABASE catalog table 1747
 - SYSINDEXES catalog table 1764
 - SYSINDEXPART catalog table 1771
 - SYSJAROBJECTS catalog table 1788
 - SYSROUTINES catalog table 1848
 - SYSSEQUENCES catalog table 1865
 - SYSTABLEPART catalog table 1882
 - SYSTABLES catalog table 1890
 - SYSTABLESPACE catalog table 1896
- ALTERIN privilege
 - GRANT statement 1349
 - REVOKE statement 1451
- ALTERINAUTH column of SYSSCHEMAAUTH catalog table 1862
- alternative syntax
 - GRANT (type or JAR file privileges) statement 1359
 - REVOKE (type or JAR file privileges) statement 1462
 - SET PATH statement 1509
- ALTIME function 2012
- AND
 - truth table 247
- ANY
 - quantified predicate 226
 - USING clause of DESCRIBE statement 1244, 1254
 - USING clause of PREPARE statement 1407
- APOST option
 - precompiler 253
- apostrophe
 - string delimiter precompiler option 253
- APOSTSQL option
 - precompiler 253
- APP_ENCODING_CCSID column
 - SYSVIEWS catalog table 1915
- APPEND
 - clause of CREATE TABLE statement 1114
- APPEND clause
 - ALTER TABLE statement 830
- APPEND column
 - SYSTABLES catalog table 1890
- APPLICATION ENCODING SCHEME clause
 - ALTER PROCEDURE (SQL - native) statement 768
 - CREATE PROCEDURE (SQL - native) statement 1056
- application plan
 - invalidated
 - ALTER TABLE statement 836
 - privileges
 - GRANT statement 1348
 - REVOKE statement 1449
- application plans 27
 - described 27
- application process 25
 - initial state in distributed unit of work 32
 - initial state in remote unit of work 36
 - state transitions 32
- application program
 - recovery 25
 - SQLCA 1646
 - SQLDA 1656
- application requester
 - definition of 31
- application server
 - definition of 31

- APPLICATION_ENCODING_CCSID column
 - SYSENVIRONMENT catalog table 1759
- APPLICATION_ENCODING_SCHEME session variable 168
- archive log 17
- ARCHIVE privilege
 - GRANT statement 1352
 - REVOKE statement 1455
- ARCHIVEAUTH column of SYSUSERAUTH catalog table 1911
- arithmetic operators 182
- AS (fullselect) WITH NO DATA clause
 - DECLARE GLOBAL TEMPORARY TABLE statement 1208
- AS clause
 - CREATE VIEW statement 1186
 - naming result columns 634
 - use in subselect 634
- AS IDENTITY clause
 - ALTER TABLE statement 803
 - CREATE TABLE statement 1095
 - DECLARE GLOBAL TEMPORARY TABLE statement 1207
- AS LOCATOR clause
 - CREATE FUNCTION statement 922, 945, 962
 - CREATE PROCEDURE (external) statement 1022
 - CREATE PROCEDURE (SQL - native) statement 1050
- AS SECURITY LABEL clause
 - ALTER TABLE statement 807
 - CREATE TABLE statement 1099
- AS WORKFILE clause of CREATE DATABASE statement 913
- ASC clause
 - ALTER TABLE statement 820
 - CREATE INDEX statement 995
 - CREATE TABLE statement 1109
 - select-statement 653
- ASCII
 - definition 38
 - effect on MBCS and DBCS characters 74
- ASCII function 288
- ASCII_CHR function 289
- ASCII_STR function 290
- ASENSITIVE clause
 - DECLARE CURSOR statement 1193
- ASIN function 291
- assembler application program
 - host variable
 - EXECUTE IMMEDIATE statement 1280
 - referencing 160
 - INCLUDE SQLCA 1652
 - INCLUDE SQLDA 1670
 - varying-length string variables 74
- assignment
 - compatibility rules 102
 - datetime values 112
 - distinct type values 113
 - IEEE floating-point numbers 106
 - numbers 105
 - retrieval rules 110
 - row ID values 113
 - statement
 - example 1548, 1613
 - SQL procedure 1548, 1613
 - storage rules 110
 - strings, basic rules for 109
 - XML values 113
- ASSOCIATE LOCATORS statement
 - description 868
 - example 870
- asterisk (*)
 - COUNT function 271
 - COUNT_BIG function 272
 - multiply sign 182
 - use in subselect 633
- ASUTIME clause
 - ALTER FUNCTION statement 715
 - ALTER PROCEDURE (external) statement 748
 - ALTER PROCEDURE (SQL - external) statement 754
 - ALTER PROCEDURE (SQL - native) statement 765
 - CREATE FUNCTION statement 934, 954
 - CREATE PROCEDURE (external) statement 1029
 - CREATE PROCEDURE (SQL - external) statement 1042
 - CREATE PROCEDURE (SQL - native) statement 1053
- ASUTIME column
 - SYSROUTINES catalog table 1848
- ATAN function 292
- ATAN2 function 294
- ATANH function 293
- ATOMIC clause
 - INSERT statement 1374
 - PREPARE statement 1412
- ATTRIBUTES clause
 - CREATE TRUSTED CONTEXT statement 1170
 - PREPARE statement 1408
- AUDIT
 - clause of CREATE TABLE statement 1111
- AUDIT clause
 - ALTER TABLE statement 830
- auditing
 - ALTER TABLE statement 830
 - CREATE TABLE statement 1111
- AUDITING column of SYSTABLES catalog table 1890
- AUTHENTICATE column
 - SYSCONTEXTAUTHIDS catalog table 1739
- AUTHENTICATEPUBLIC column
 - SYSCONTEXT catalog table 1737
- AUTHHOWGOT
 - column of SYSSEQUENCEAUTH catalog table 1863
- AUTHHOWGOT column
 - SYSDBAUTH catalog table 1751
 - SYSBACKAUTH catalog table 1818
 - SYSPLANAUTH catalog table 1837
 - SYSRESAUTH catalog table 1843
 - SYSROUTINEAUTH catalog table 1846
 - SYSSCHEMAAUTH catalog table 1862
 - SYSUSERAUTH catalog table 1911
- AUTHHOWGOT column of SYSTABAUTH catalog table 1878
- AUTHID
 - column of MODESELECT catalog table 1708
 - column of SYSCOPY catalog table 1740
 - column of USERNAMES catalog table 1920
- AUTHID column
 - SYSCONTEXTAUTHIDS catalog table 1739
- authority
 - retrieving catalog information 2002
- authorization
 - clause of CONNECT statement 900
 - naming convention 51
- authorization ID
 - primary 62
 - privileges 60
 - resulting from errors 1472
 - secondary 62
 - translating
 - concepts 68

- AUX clause of CREATE AUXILIARY TABLE statement 909
- aux-table
 - naming convention 51
- AUXILIARY clause of CREATE AUXILIARY TABLE statement 909
- auxiliary table
 - CREATE AUXILIARY TABLE statement 908
- AUXRELOBID column
 - SYSAUXRELS catalog table 1709
- AUXTBNAME column of SYSAUXRELS catalog table 1709
- AUXTBOWNER column of SYSAUXRELS catalog table 1709
- AVG function 269
- AVGKEYLEN column
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
- AVGROWLEN
 - column of SYSTABLESPACE catalog table 1896
- AVGROWLEN column
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
- AVGSIZE column
 - SYSLOBSTATS catalog table 1807
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832

B

- BASED UPON CONNECTION clause
 - CREATE TRUSTED CONTEXT statement 1169
- basic operations in SQL 102
- basic predicate 224
- BCOLNAME column
 - SYSDEPENDENCIES catalog table 1756
- BCOLNO column
 - SYSDEPENDENCIES catalog table 1756
- BCREATOR column
 - SYSPLANDEP catalog table 1839
 - SYSVIEWDEP catalog table 1914
- BEGIN DECLARE SECTION statement
 - description 872
 - example 873
- BETWEEN predicate 229
- BIGINT
 - data type 71
 - CREATE TABLE statement 1087
- BIGINT (binary large integer) function 295
- BINARY
 - data type 1090
- BINARY function 297
- binary large object (BLOB) 85
- BINARY LARGE OBJECT data type 85
- binary string
 - assignment 109
 - constants 129
 - description 85
- binary strings
 - varying-length
 - description 85
- bind behavior for dynamic SQL statements 64
- BIND PACKAGE subcommand of DSN
 - options
 - QUALIFIER 58
- BIND PLAN subcommand of DSN
 - options
 - QUALIFIER 58
- BIND privilege
 - GRANT statement 1345, 1348
 - REVOKE statement 1447, 1449
- bind process 63
- BIND_OPTS column
 - SYSJAVA_OPTS catalog table 1789
 - SYSROUTINES_OPTS catalog table 1859
- BINDADD privilege
 - binding a package 67
 - GRANT statement 1352
 - REVOKE statement 1455
- BINDADDAUTH column of SYSUSERAUTH catalog table 1911
- BINDAGENT privilege
 - GRANT statement 1352
 - REVOKE statement 1455
- BINDAGENTAUTH column of SYSUSERAUTH catalog table 1911
- BINDAUTH column
 - SYSPACKAUTH catalog table 1818
 - SYSPLANAUTH catalog table 1837
- BINDERROR column of SYSPACKSTMT catalog table 1823
- binding
 - process 63
 - SQL statements 1
- BINDTIME column
 - SYSPACKAGE catalog table 1810
- BIT data
 - description 74
- BLOB (binary large object)
 - data type 85, 1090
 - description 85
 - file reference 165
 - host variable 163
 - locator 163
- BLOB (binary large object) function 299
- BLOB LARGE OBJECT data type 1090
- BNAME column
 - SYSCONSTDEP catalog table 1736
 - SYSDEPENDENCIES catalog table 1756
 - SYSPACKDEP catalog table 1820
 - SYSPLANDEP catalog table 1839
 - SYSVIEWDEP catalog table 1914
- BNAME column of SYSSEQUENCEDEP catalog table 1867
- bootstrap data set (BSDS)
 - defined 18
- BOTH
 - USING clause of DESCRIBE statement 1244, 1254
- BOUNDDBY column of SYSPLAN catalog table 1832
- BOUNDTS column
 - SYSPLAN catalog table 1832
- BOWNER column
 - SYSDEPENDENCIES catalog table 1756
- BOWNERTYPE column
 - SYSDEPENDENCIES catalog table 1756
- BPOOL column
 - SYSDATABASE catalog table 1747
 - SYSINDEXES catalog table 1764
 - SYSTABLESPACE catalog table 1896
- BQUALIFIER column of SYSPACKDEP catalog table 1820
- BSHEMA column
 - SYSCONSTDEP catalog table 1736
 - SYSDEPENDENCIES catalog table 1756

- BSHEMA column (*continued*)
 - SYSVIEWDEP catalog table 1914
- BSHEMA column of SYSSEQUENCEDEP catalog table 1867
- BSDS (bootstrap data set)
 - privilege
 - granting 1352
 - revoking 1455
- BSDSAUTH column of SYSUSERAUTH catalog table 1911
- BSEQUENCEID column of SYSSEQUENCEDEP catalog table 1867
- BTYPE column
 - SYSCONSTDEP catalog table 1736
 - SYSDEPENDENCIES catalog table 1756
 - SYSPACKDEP catalog table 1820
 - SYSPLANDEP catalog table 1839
 - SYSVIEWDEP catalog table 1914
- buffer pool
 - naming convention 51
- buffer pools
 - described 18
- BUFFERPOOL clause
 - ALTER DATABASE statement 698
 - ALTER INDEX statement 728
 - ALTER TABLESPACE statement 843
 - CREATE DATABASE statement 913
 - CREATE INDEX statement 1007
 - CREATE TABLESPACE statement 1140
- BUFFERPOOL privilege
 - GRANT statement 1361
 - REVOKE statement 1463
- BUILDDATE column
 - SYSROUTINES_OPTS catalog table 1859
 - SYSROUTINES_SRC catalog table 1861
- BUILDNAME column
 - SYSJAVA_OPTS catalog table 1789
 - SYSROUTINES_OPTS catalog table 1859
- BUILDOWNER column
 - SYSJAVA_OPTS catalog table 1789
 - SYSROUTINES_OPTS catalog table 1859
- BUILDSHEMA column
 - SYSJAVA_OPTS catalog table 1789
 - SYSROUTINES_OPTS catalog table 1859
- BUILDSTATUS column
 - SYSROUTINES_OPTS catalog table 1859
 - SYSROUTINES_SRC catalog table 1861
- BUILDTIME column
 - SYSROUTINES_OPTS catalog table 1859
 - SYSROUTINES_SRC catalog table 1861
- built-in data type 69
- built-in function
 - description 173
 - invocation 178
 - resolution 175
 - string units 76
- business rules
 - enforcing 20
- BY clause of REVOKE statement 1433

C

- C application program
 - host variable
 - EXECUTE IMMEDIATE statement 1280
 - referencing 160
 - INCLUDE SQLCA 1652
 - INCLUDE SQLDA 1670
 - varying-length string 74

- CACHE
 - clause of ALTER SEQUENCE statement 784
- CACHE clause
 - ALTER TABLE statement 805
 - CREATE SEQUENCE statement 1070
- CACHE column of SYSSEQUENCES catalog table 1865
- CACHESIZE
 - column of SYSPLAN catalog table 1832
- Call Level Interface (CLI) 3
- CALL statement
 - description 874
 - example 884, 1550, 1615
 - SQL procedure 1550, 1615
- CALLED ON NULL INPUT clause
 - ALTER FUNCTION statement 710, 724
 - CREATE FUNCTION statement 928, 949, 979
 - CREATE PROCEDURE (external) statement 1031
 - CREATE PROCEDURE (SQL - external) statement 1041
- capturing changed data
 - ALTER TABLE statement 828
 - CREATE TABLE statement 1112
- CARD column
 - SYSTABLEPART catalog table
 - description 1882
 - SYSTABSTATS catalog table
 - description 1907
- CARDF column
 - SYSOLDIST catalog table 1715
 - SYSOLDIST_HIST catalog table 1719
 - SYSOLDISTSTATS catalog table 1717
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSKEYTARGETSTATS catalog table 1796
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
 - SYSTABSTATS catalog table 1907
 - SYSTABSTATS_HIST catalog table 1908
- CARDINALITY clause 640
- CARDINALITY column
 - SYSROUTINES catalog table 1848
- CARDINALITY MULTIPLIER clause 640
- CASCADE delete rule
 - ALTER TABLE statement 816
 - CREATE TABLE statement 1101
- cascade revoke 1434
- CASE expression
 - description 199
- CASE statement
 - example 1552, 1617
 - SQL procedure 1552, 1617
- cast function 174
- CAST specification
 - definition 202
 - NULL 202
 - parameter marker 202
 - string units 76
- CAST_FUNCTION column
 - SYSPARAMS catalog table 1826
 - SYSROUTINES catalog table 1848

- CAST_FUNCTION_ID column of SYSPARMS catalog table 1826
- casting
 - XML values 210
- casts
 - data types 96
- catalog
 - defined 16
 - naming convention 51
 - tables
 - defined 16
- catalog name
 - VCAT clause
 - ALTER INDEX statement 730
 - CREATE INDEX statement 1000
 - CREATE TABLESPACE statement 1131, 1134
- catalog tables
 - description 1677
 - indexes 1679
 - IPLIST 1697
 - IPNAMES 1698
 - LOCATIONS 1701
 - LULIST 1703
 - LUMODES 1704
 - LUNAMES 1705
 - MODESELECT 1708
 - release dependency indicators 1677
 - retrieving information about
 - primary keys 2002
 - status 2003
 - SQL statements allowed 1689
 - SYSAUXRELS 1709, 2004
 - SYSCHECKDEP 1710
 - SYSCHECKS 1711
 - SYSCHECKS2 1712
 - SYSCOLAUTH 1713
 - SYSOLDIST
 - contents 1715
 - SYSOLDISTSTATS
 - contents 1717
 - SYSOLSTATS
 - contents 1721
 - SYSCOLUMNS
 - contents 1723
 - updated by COMMENT ON statement 2006
 - updated by CREATE VIEW statement 2001
 - SYSCOLUMNS_HIST
 - contents 1732
 - SYSCONSTDEP 1736
 - SYSCONTEXT
 - contents 1737
 - SYSCONTEXTAUTHIDS
 - contents 1739
 - SYSCOPY
 - contents 1740
 - SYSCTXTRUSTATTRS
 - contents 1746
 - SYSDATABASE
 - contents 1747
 - SYSDATATYPES 1749
 - SYSDBAUTH 1751
 - SYSDBRM 1754
 - SYSDEPENDENCIES 1756
 - SYSDUMMY1 1758
 - SYSENVIRONMENT 1759
 - SYSFIELDS 1761
 - SYSFOREIGNKEYS 1763, 2003

- catalog tables (*continued*)
 - SYSINDEXES
 - contents 1764
 - SYSINDEXES_HIST
 - contents 1769
 - SYSINDEXPART
 - contents 1771
 - SYSINDEXPART_HIST 1775
 - SYSINDEXSPACESTATS
 - contents 1777
 - SYSINDEXSTATS
 - contents 1782
 - SYSINDEXSTATS_HIST 1784
 - SYSJARCLASS_SOURCE 1785
 - SYSJARCONTENTS 1786
 - SYSJARDATA 1787
 - SYSJAROBJECTS 1788
 - SYSJAVA_OPTS 1789
 - SYSJAVAPATHS 1790
 - SYSKEYCOLUSE 1791
 - SYSKEYS 1792
 - SYSKEYTARGETS
 - contents 1793
 - SYSKEYTARGETS_HIST
 - contents 1798
 - SYSKEYTARGETSTATS
 - contents 1796
 - SYSKEYTGTDIST
 - contents 1801
 - SYSKEYTGTDIST_HIST
 - contents 1805
 - SYSKEYTGTDISTSTATS
 - contents 1803
 - SYSLOBSTATS 1807
 - SYSLOBSTATS_HIST 1808
 - SYSOBJROLEDEP
 - contents 1809
 - SYSPACKAGE 1810
 - SYSPACKAUTH 1818
 - SYSPACKDEP 1820
 - SYSPACKLIST 1822
 - SYSPACKSTMT 1823
 - SYSPARMS 1826
 - SYSPKSYSTEM 1830
 - SYSPLAN 1832
 - SYSPLANAUTH
 - contents 1837
 - SYSPLANDEP
 - contents 1839
 - SYSPLSYSTEM 1840
 - SYSRELS
 - contents 1841
 - describes referential constraints 2003
 - SYSRESAUTH 1843
 - SYSROLES
 - contents 1845
 - SYSROUTINEAUTH 1846
 - SYSROUTINES 2005
 - contents 1848
 - SYSROUTINES_OPTS 1859
 - SYSROUTINES_SRC 1861
 - SYSROUTINESTEXT
 - contents 1858
 - SYSSCHEMAAUTH 1862
 - SYSSEQUENCEAUTH 1863
 - SYSSEQUENCES 1865, 2006
 - SYSSEQUENCESDEP 1867

- catalog tables (*continued*)
 - SYSTMT 1868
 - SYSTOGROUP
 - contents 1872
 - sample query 1998
 - SYSSTRINGS
 - contents 1874
 - SYSSYNONYMS 1877
 - SYSTABAUTH
 - contents 1878
 - table authorizations 2002
 - updated by CREATE VIEW statement 2001
 - view authorizations 2002
 - SYSTABCONST
 - contents 1881
 - SYSTABLEAPART
 - partition order 1999
 - SYSTABLEPART
 - contents 1882
 - SYSTABLEPART_HIST
 - contents 1887
 - SYSTABLES
 - contents 1890
 - rows maintained 1999
 - updated by COMMENT ON statement 2006
 - updated by CREATE VIEW statement 2001
 - SYSTABLES_HIST
 - contents 1905
 - SYSTABLESPACE
 - contents 1896
 - SYSTABLESPACESTATS
 - contents 1901
 - SYSTABSTATS
 - contents 1907
 - SYSTABSTATS_HIST
 - contents 1908
 - SYSTRIGGERS 1909, 2005
 - SYSUSERAUTH 1911
 - SYSVIEWDEP
 - contents 1914
 - updated by CREATE VIEW statement 2002
 - SYSVIEWS 1915
 - SYSVOLUMES 1917
 - SYSXMLRELS 1918
 - SYSXMLSTRINGS 1919
 - table space 1679
 - USERNAMES 1920
- catalog, DB2
 - constraint information 2004
 - database design 1998, 2007
 - retrieving information from 1998
 - tables 1677
- CCSID
 - clause of CREATE DATABASE statement 913
 - clause of CREATE FUNCTION statement 921, 944, 961, 976
 - clause of CREATE GLOBAL TEMPORARY TABLE statement 984
 - clause of CREATE TABLE statement 1113
 - clause of CREATE TABLESPACE statement 1142
 - clause of CREATE TYPE statement 1179
 - clause of DECLARE GLOBAL TEMPORARY TABLE statement 1210
 - column of SYSPARMS catalog table 1826
- CCSID (coded character set identifier)
 - definition 38
 - Definition 43
- CCSID (coded character set identifier) (*continued*)
 - description 38
- CCSID (Coded Character Set Identifier)
 - of strings 43
- CCSID clause
 - ALTER DATABASE statement 698
 - ALTER TABLESPACE statement 843
 - CREATE PROCEDURE (external) statement 1021
 - CREATE PROCEDURE (SQL - external) statement 1039
- CCSID column
 - SYS_COLUMNS catalog table 1723
 - SYSKEYTARGETS catalog table 1793
- CCSID_ENCODING function 300
- CD-ROM, books on 2033
- CEIL function 301
- CEILING function 301
- CHAR
 - data type 74
- CHAR function 302
- CHAR LARGE OBJECT data type 74, 1089
- CHAR VARYING data type 74, 1088
- character 47
- character conversion
 - ASCII 38
 - assignment rules 111
 - character set 38
 - code page 38
 - code point 38
 - coded character set 38
 - comparison rules 122
 - concatenation rules 666
 - contracting conversion 46
 - description 38
 - EBCDIC 38
 - encoding scheme 38
 - expanding conversion 46
 - set operations rules 666
 - substitution character 38
 - SYSIBM.SYSSTRINGS catalog table 1874
 - Unicode 38
 - UTF-16 38
 - UTF-8 38
- Character conversion
 - Coded character sets and ccsids 43
- CHARACTER data type
 - CREATE TABLE statement 1088
 - description 74
- character large object (CLOB) 85
- CHARACTER LARGE OBJECT data type 74, 1089
- character set 38
- character string
 - assignment 109
 - comparison 116
 - constants 128
 - description 73
 - empty 73
- CHARACTER VARYING data type 74, 1088
- CHARACTER_LENGTH function 310
- Characteristics of SQL statements in DB2 1600
- CHARSET column
 - SYSDBRM catalog table 1754
 - SYSENVIRONMENT catalog table 1759
 - SYSPACKAGE catalog table 1810
- CHECK
 - clause of CREATE TABLE statement 1102
 - column of SYSVIEWS catalog table 1915

- CHECK clause
 - ALTER TABLE statement 817
- check constraint
 - defining
 - ALTER TABLE statement 817
 - SYSCHECKDEP catalog table 1710
- check constraints 24
- check pending status
 - retrieving catalog information 2003
- CHECKCONDITION column
 - SYSCHECKS catalog table 1711
- CHECKEXISTINGDATA column
 - SYSRELS catalog table 1841
- CHECKFLAG column
 - SYSTABLEPART catalog table 1882
 - SYSTABLES catalog table 1890
- CHECKNAME column
 - SYSCHECKDEP catalog table 1710
 - SYSCHECKS catalog table 1711
 - SYSCHECKS2 catalog table 1712
- CHECKRID5B column
 - SYSTABLEPART catalog table 1882
 - SYSTABLES catalog table 1890
- CHECKS column
 - SYSTABLES catalog table 1890
- CHILDREN column of SYSTABLES catalog table 1890
- CLASS column
 - SYSJARCONTENTS catalog table 1786
 - SYSROUTINES catalog table 1848
- CLASS_SOURCE column
 - SYSJARCLASS_SOURCE catalog table 1785
 - SYSJARCONTENTS catalog table 1786
- CLASS_SOURCE_ROWID column
 - SYSJARCONTENTS catalog table 1786
- CLI (Call Level Interface) 3
- CLOB (character large object)
 - description 74, 85, 1089
 - file reference 165
 - function 312
 - host variable 163
 - locator 163
- CLONE column
 - SYSTABLESPACE catalog table 1896
- clone table
 - naming convention 52
- CLOSE
 - clause of ALTER INDEX statement 728
 - clause of CREATE INDEX statement
 - description 1007
 - clause of CREATE TABLESPACE statement
 - description 1142
 - statement
 - description 885
 - example 885
- CLOSE clause
 - ALTER TABLESPACE statement 844
- closed state of cursor 1402
- CLOSERULE column
 - SYSINDEXES catalog table 1764
 - SYSTABLESPACE catalog table 1896
- CLUSTER clause
 - ALTER INDEX statement 734
 - CREATE INDEX statement 999
- CLUSTERED column of SYSINDEXES catalog table
 - description 1764
- CLUSTERING column
 - SYSINDEXES_HIST catalog table 1769
- CLUSTERING column of SYSINDEXES catalog table
 - description 1764
- CLUSTERRATIO column
 - SYSINDEXES catalog table 1764
 - SYSINDEXSTATS catalog table 1782
- CLUSTERRATIOF column
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSINDEXSTATS catalog table 1782
 - SYSINDEXSTATS_HIST catalog table 1784
- CLUSTERTYPE column of SYSTABLES catalog table 1890
- CNAME column
 - SYSPKSYSTEM catalog table 1830
 - SYSPLSYSTEM catalog table 1840
- COALESCE function 119, 315, 387
- COBOL application program
 - host structure 171
 - host variable
 - description 160
 - EXECUTE IMMEDIATE statement 1280
 - host-variable-arrays 172
 - INCLUDE SQLCA 1652
 - varying-length string 74
- COBOL_STRING_DELIMITER session variable 168
- code page 38
- code point 38
- coded character set 38
- CODEUNITS16 76
- CODEUNITS32 76
- COLCARDATA column of SYSCOLSTATS catalog table 1721
- COLCARDF column
 - SYSCOLUMNS catalog table 1723
 - SYSCOLUMNS_HIST catalog table 1732
- COLCOUNT column
 - SYSINDEXES catalog table 1764
 - SYSRELS catalog table 1841
 - SYSTABCONST catalog table 1881
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
- COLGROUPCOLNO column
 - SYSCOLDIST catalog table 1715
 - SYSCOLDIST_HIST catalog table 1719
 - SYSCOLDISTSTATS catalog table 1717
- COLLATION_KEY function 317
- collection-id
 - naming convention 52
- collection, package
 - granting privileges 1336
 - revoking privileges 1437
 - SET CURRENT PACKAGESET statement 1492
- COLLID
 - column of SYSSEQUENCEAUTH catalog table 1863
- COLLID clause
 - ALTER FUNCTION statement 714
 - ALTER PROCEDURE (external) statement 747
 - ALTER PROCEDURE (SQL - external) statement 754
 - CREATE FUNCTION statement 933, 954
 - CREATE PROCEDURE (external) statement 1028
 - CREATE PROCEDURE (SQL - external) statement 1041
- COLLID column
 - SYSCOLAUTH catalog table 1713
 - SYSPACKAGE catalog table 1810
 - SYSPACKAUTH catalog table 1818
 - SYSPACKLIST catalog table 1822
 - SYSPACKSTMT catalog table 1823
 - SYSPKSYSTEM catalog table 1830

- COLLID column (*continued*)
 - SYSROUTINEAUTH catalog table 1846
 - SYSROUTINES catalog table 1848
 - SYSTABAUTH catalog table 1878
- COLNAME column
 - SYSAXRELS catalog table 1709
 - SYSCHECKDEP catalog table 1710
 - SYSFOREIGNKEYS catalog table 1763
 - SYSKEYCOLUSE catalog table 1791
 - SYSKEYS catalog table 1792
 - SYSXMLRELS catalog table 1918
- COLNAME column of SYSCOLAUTH catalog table 1713
- COLNO column
 - SYSOLUMNS_HIST catalog table 1732
 - SYSFIELDS catalog table 1761
 - SYSFOREIGNKEYS catalog table 1763
 - SYSKEYCOLUSE catalog table 1791
 - SYSKEYS catalog table 1792
 - SYSKEYTARGETS catalog table 1793
- COLNO column of SYSCOLUMNS catalog table 1723
- colon
 - host variable in SQL 160
- COLSEQ column
 - SYSFOREIGNKEYS catalog table 1763
 - SYSKEYCOLUSE catalog table 1791
 - SYSKEYS catalog table 1792
- COLSTATUS column of SYSCOLUMNS catalog table 1723
- COLTYPE column
 - SYSOLUMNS_HIST catalog table 1732
- COLTYPE column of SYSCOLUMNS catalog table 1723
- column
 - derived
 - CREATE VIEW statement 1186
 - DELETE statement 1229
 - functions 259
 - INSERT statement 1370
 - string comparison 122
 - UPDATE statement 1526
 - name
 - ambiguous reference 155
 - correlated reference 157
 - in a result 635
 - undefined reference 155
 - naming convention 52
 - retrieving
 - catalog information 2000
 - comments 2006
- COLUMN clause
 - COMMENT statement 890
 - LABEL statement 1385
- COLVALUE column
 - SYSOLDIST catalog table 1715
 - SYSOLDIST_HIST catalog table 1719
 - SYSOLDISTSTATS catalog table
 - description 1717
- COMMA
 - column of SYSDBRM catalog table 1754
 - column of SYSPACKAGE catalog table 1810
 - option of precompiler 251
- comment
 - adding 887
 - replacing 887
 - SQL 48
- COMMENT ON statement
 - column name qualification 154
 - examples 2006
 - storing 2006
- COMMENT statement
 - description 887
 - example 894
- comments
 - SQL statements 695
- commit
 - description 25
- COMMIT ON RETURN clause
 - ALTER PROCEDURE (external) statement 749
 - ALTER PROCEDURE (SQL - external) statement 756
 - AUTONOMOUS clause
 - CREATE PROCEDURE (SQL - native) statement 766
 - CREATE PROCEDURE (external) statement 1031
 - CREATE PROCEDURE (SQL - external) statement 1043
 - CREATE PROCEDURE (SQL - native) statement 766, 1053
- commit processing 31
- COMMIT statement
 - description 896
 - example 898
- COMMIT_ON_RETURN column
 - SYSROUTINES catalog table 1848
- common table expression 670
- communications database 16
- COMPARE_DECFLOAT function 320
- comparison
 - compatibility rules 102
 - datetime values 117
 - distinct type values 117
 - numbers 114
 - row ID values 117
 - strings 116
 - XML values 117
- compatibility
 - data types 102
 - rules 102
- COMPILE_OPTS column
 - SYSROUTINES_OPTS catalog table 1859
- COMPONENT column
 - SYSIBM.XSRCOMPONENT table 1921
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSRPROPERTY table 1928
- composite key 6
- compound statement
 - example 1554, 1620
 - order of statements in 1554, 1620
 - SQL procedure 1554, 1620
- COMPRESS
 - clause of CREATE TABLESPACE statement 1142
 - column of SYSTABLEPART catalog table 1882
- COMPRESS clause
 - ALTER TABLESPACE statement 844
- COMPRESS column
 - SYSINDEXES catalog table 1764
- COMPRESS NO clause
 - ALTER INDEX statement 735
 - CREATE INDEX statement 1005
- COMPRESS YES clause
 - ALTER INDEX statement 735
 - CREATE INDEX statement 1005
- CONCAT
 - function 322
 - operator 189
- concatenation
 - CONCAT function 322
 - operator 189
- concurrency
 - application 25

- concurrency (*continued*)
 - LOCK TABLE statement 1386
- condition
 - naming convention 55
- CONNECT
 - option of precompiler 248
 - statement 899
- connectable and connected state 36
- connectable and unconnected state 36
- connected state 34
- connection
 - application process states 34, 36
 - definition of 31
 - initial state in distributed unit of work 32
 - management in distributed unit of work 32
 - management in remote unit of work 36
 - SQL state
 - in a distributed unit of work 33
 - state transitions 32
 - when ended in a distributed unit of work 34
- connection exit routine
 - description 148
- connection state
 - SET CONNECTION statement 1475
- consistency, point 25
- constant
 - binary string 129
 - character string 128
 - datetime 129
 - decimal 127
 - decimal floating point 127
 - floating-point 127
 - graphic string 130
 - hexadecimal 128
 - integer 127
- CONSTNAME column
 - SYSKEYCOLUSE catalog table 1791
 - SYSTABCONST catalog table 1881
- constraint
 - description 20
 - naming convention 52
 - unique 20
- CONSTRAINT clause
 - ALTER TABLE statement 814, 815, 817
 - CREATE TABLE statement 1091, 1099, 1100
- CONSTRAINT
 - clause of CREATE TABLE statement 1102
- CONTAINS function 323
- CONTAINS SQL clause
 - ALTER FUNCTION statement 710, 723
 - ALTER PROCEDURE (external) statement 747
 - ALTER PROCEDURE (SQL - external) statement 753
 - ALTER PROCEDURE (SQL - native) statement 764
 - CREATE FUNCTION statement 928, 950, 979
 - CREATE PROCEDURE (external) statement 1026
 - CREATE PROCEDURE (SQL - external) statement 1041
 - CREATE PROCEDURE (SQL - native) statement 1051
- context-name
 - naming convention 52
- context-name clause
 - ALTER TRUSTED CONTEXT statement 857
 - CREATE TRUSTED CONTEXT statement 1169
- CONTEXTID column
 - SYSCONTEXT catalog table 1737
 - SYSCONTEXTAUTHIDS catalog table 1739
 - SYSCTXITRUSTATTRS catalog table 1746
- CONTINUE
 - clause of WHENEVER statement 1539
- CONTINUE AFTER FAILURE clause
 - ALTER FUNCTION statement 716
 - ALTER PROCEDURE (external) statement 750
 - ALTER PROCEDURE (SQL - external) statement 756
 - CREATE FUNCTION statement 935, 956
 - CREATE PROCEDURE (external) statement 1030
 - CREATE PROCEDURE (SQL - external) statement 1044
- CONTINUE handler
 - SQL procedure 1554, 1620
- CONTOKEN
 - column of SYSSEQUENCEAUTH catalog table 1863
- CONTOKEN column
 - SYSCOLAUTH catalog table 1714
 - SYSPACKAGE catalog table 1810
 - SYSPACKSTMT catalog table 1823
 - SYSPKSYSTEM catalog table 1830
 - SYSROUTINEAUTH catalog table 1846
 - SYSROUTINES catalog table 1848
 - SYSTABAUTH catalog table 1878
- control character 48
- control statement 1541, 1610
- conversion of numbers
 - errors 1472
 - precision 106
 - scale 106
- CONVERT TO clause
 - ALTER INDEX statement 725
- CONVLIMIT column of LUMODES catalog table
 - description 1704
- COPY
 - clause of ALTER INDEX statement 728
 - clause of CREATE INDEX statement 1009
 - column of SYSINDEXES catalog table 1764
- COPY privilege
 - GRANT statement 1345
 - REVOKE statement 1447
- COPYAUTH column of SYSPACKAUTH catalog table 1818
- COPYCHANGES column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- COPYLASTTIME column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- COPYLRN column of SYSINDEXES catalog table 1764
- COPYPAGESF column of SYSCOPY catalog table 1740
- COPYUPDATEDPAGES column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- COPYUPDATELRN column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- COPYUPDATETIME column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- correlated reference
 - correlation name
 - defining 154
 - FROM clause of subselect 638
 - naming convention 52
 - qualifying a column name 154
 - description 157
 - HAVING clause 651
 - WHERE clause 648
- CORRELATION
 - function 270

- correlation name 643
- COS function 326
- COSH function 327
- COUNT function 271
- COUNT_BIG function 272
- COVARIANCE or COVARIANCE_SAMP function 274
- CPAGESF column of SYSCOPY catalog table 1740
- CREATE ALIAS statement
 - description 905
 - example 907
- CREATE AUXILIARY TABLE statement
 - description 908
 - example 911
- CREATE DATABASE statement
 - description 12, 912
 - example 914
- CREATE FUNCTION (external scalar) statement
 - description 916
 - example 938
- CREATE FUNCTION (external table) statement
 - description 940
 - example 957
- CREATE FUNCTION (sourced) statement
 - description 958
 - example 970
- CREATE FUNCTION (SQL scalar) statement
 - description 972
 - example 981
- CREATE FUNCTION statement 915
- CREATE GLOBAL TEMPORARY TABLE statement
 - description 982
 - example 987
- CREATE IN privilege
 - binding a package 67
 - GRANT statement 1336
 - REVOKE statement 1437
- CREATE INDEX statement
 - description 988
 - example 1012
- CREATE PROCEDURE (external) statement
 - description 1016
 - example 1034
- CREATE PROCEDURE (SQL - external) statement
 - description 1035
 - example 1045
- CREATE PROCEDURE (SQL - native) statement
 - description 1046
 - examples 1063
- CREATE PROCEDURE statement 1015
 - assignment statement 1548, 1613
 - SQL procedure body 1546, 1612
- CREATE ROLE statement
 - description 1065
 - example 1065
- CREATE SEQUENCE
 - use 30
- CREATE SEQUENCE statement
 - description 1066
 - example 1073
- CREATE STOGROUP statement 1074
 - example 1076
- CREATE SYNONYM statement
 - description 1077
 - example 1077
- CREATE TABLE statement
 - description 1079

- CREATE TABLE statement (*continued*)
 - example 1124
 - materialized query table 1079
- CREATE TABLESPACE statement
 - description 1128
 - example 1149
- CREATE TRIGGER statement
 - description 1151
 - example 1164
- CREATE TRUSTED CONTEXT statement
 - description 1167
 - example 1176
 - usage notes 1175
- CREATE TYPE statement
 - description 1177
 - example 1183
- CREATE VIEW statement
 - description 1184
 - example 1190
- CREATEALIAS privilege
 - GRANT statement 1352
 - REVOKE statement 1456
- CREATEALIASAUTH column of SYSUSERAUTH catalog table 1911
- CREATEDBA privilege
 - GRANT statement 1352
 - REVOKE statement 1456
- CREATEDBAAUTH column of SYSUSERAUTH catalog table 1911
- CREATEDBC privilege
 - GRANT statement 1353
 - REVOKE statement 1456
- CREATEDBCAUTH column of SYSUSERAUTH catalog table 1911
- CREATEDBY column
 - SYSDATABASE catalog table 1747
 - SYSDATATYPES catalog table 1749
 - SYSIBM.XSOBJECTS table 1922
 - SYSINDEXES catalog table 1764
 - SYSROUTINES catalog table 1848
 - SYSSEQUENCES catalog table 1865
 - SYSSTOGROUP catalog table 1872
 - SYSSYNONYMS catalog table 1877
 - SYSTABLES catalog table 1890
 - SYSTABLESPACE catalog table 1896
 - SYSTRIGGERS catalog table 1909
- CREATEDTS column
 - SYSCOLUMNS catalog table 1723
 - SYSCONTEXT catalog table 1737
 - SYSCONTEXTAUTHIDS catalog table 1739
 - SYSCTXTRUSTATTRS catalog table 1746
 - SYSDATABASE catalog table 1747
 - SYSDATATYPES catalog table 1749
 - SYSIBM.XSOBJECTCOMPONENTS table 1924
 - SYSIBM.XSOBJECTS table 1922
 - SYSINDEXES catalog table 1764
 - SYSINDEXPART catalog table 1771
 - SYSJAROBJECTS catalog table 1788
 - SYSKEYTARGETS catalog table 1793
 - SYSROLES catalog table 1845
 - SYSROUTINES catalog table 1848
 - SYSSEQUENCES catalog table 1865
 - SYSSTOGROUP catalog table 1872
 - SYSSYNONYMS catalog table 1877
 - SYSTABCONST catalog table 1881
 - SYSTABLEPART catalog table 1882
 - SYSTABLES catalog table 1890

CREATEDTS column *(continued)*
 SYSTABLESPACE catalog table 1896
 SYSTRIGGERS catalog table 1909
 SYSXMLRELS catalog table 1918
 CREATEIN privilege
 GRANT statement 1349
 REVOKE statement 1451
 CREATEINAUTH column of SYSSCHEMAAUTH catalog table 1862
 CREATESG privilege
 GRANT statement 1353
 REVOKE statement 1456
 CREATESGAUTH column of SYSUSERAUTH catalog table 1911
 CREATESTMT column
 SYSROUTINES_SRC catalog table 1861
 CREATETAB privilege
 GRANT statement 1337
 REVOKE statement 1439
 CREATETABAUTH column of SYSDBAUTH catalog table 1751
 CREATETMTAB privilege
 GRANT statement 1353
 REVOKE statement 1456
 CREATETMTABAUTH column
 SYSUSERAUTH catalog table 1911
 CREATETS privilege
 GRANT statement 1337
 REVOKE statement 1439
 CREATETSAUTH column of SYSDBAUTH catalog table 1751
 CREATOR column
 SYSCHECKS catalog table 1711
 SYSCOLAUTH catalog table 1713
 SYSDATABASE catalog table 1747
 SYSFOREIGNKEYS catalog table 1763
 SYSINDEXES catalog table 1764
 SYSINDEXES_HIST catalog table 1769
 SYSINDEXSPACESTATS catalog table 1777
 SYSPACKAGE catalog table 1810
 SYSPLAN catalog table 1832
 SYSRELS catalog table 1841
 SYSTOGROUP catalog table 1872
 SYSSYNONYMS catalog table 1877
 SYSTABCONST catalog table 1881
 SYSTABLES catalog table 1890
 SYSTABLES_HIST catalog table 1905
 SYSTABLESPACE catalog table 1896
 SYSVIEWS catalog table 1915
 CREATOR TYPE column
 SYSDATABASE catalog table 1747
 SYSPLAN catalog table 1832
 SYSTOGROUP catalog table 1872
 SYSSYNONYMS catalog table 1877
 SYSTABCONST catalog table 1881
 SYSTABLESPACE catalog table 1896
 CURRENCY function 2014
 CURRENT
 clause of RELEASE statement 1424
 CURRENT APPLICATION ENCODING SCHEME special register 134
 CURRENT clause
 FETCH statement 1295
 CURRENT CLIENT_ACCTNG special register 135
 CURRENT CLIENT_APPLNAME special register 136
 CURRENT CLIENT_USERID special register 136
 CURRENT CLIENT_WRKSTNNAME special register 137
 current connection state 33
 CURRENT DATA clause
 ALTER PROCEDURE (SQL - native) statement 767
 CREATE PROCEDURE (SQL - native) statement 1054
 CURRENT DATE special register 137
 CURRENT DEBUG MODE special register 137, 1478
 CURRENT DECFLOAT ROUNDING MODE special register 138, 1480
 CURRENT DEGREE special register
 assigning a value 1482
 description 139
 setting 1482
 CURRENT LC_CTYPE special register
 description 140
 CURRENT LOCALE LC_CTYPE special register
 assigning a value 1483
 description 140
 CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register
 description 140
 CURRENT MEMBER
 description 141
 CURRENT OPTIMIZATION HINT special register
 assigning a value 1487
 description 141
 CURRENT PACKAGE PATH clause
 SET PATH statement 1507
 CURRENT PACKAGE PATH special register
 description 141
 CURRENT PACKAGESET special register
 assigning a value 1492
 description 142
 stored procedures 1493
 CURRENT PATH clause
 SET PATH statement 1507
 CURRENT PATH special register
 assigning a value 1507
 description 142
 CURRENT PRECISION special register
 assigning a value 1494
 description 143
 CURRENT REFRESH AGE special register
 description 144
 CURRENT ROUTINE VERSION special register 145, 1497
 CURRENT ROWSET clause
 FETCH statement 1300
 CURRENT RULES special register
 assigning a value 1499
 description 145
 CURRENT SCHEMA special register
 assigning a value 1510
 description 147
 CURRENT SERVER special register
 description 147
 CURRENT SQLID special register
 assigning a value 1500
 description 148
 initial value 64
 CURRENT TIME special register
 description 148
 CURRENT TIMESTAMP special register
 description 149
 CURRENT TIMEZONE special register 149
 CURRENT_SCHEMA column
 SYSENVIRONMENT catalog table 1759
 CURRENT_VERSION
 column of SYSTABLESPACE catalog table 1896

- CURRENT_VERSION column
 - SYSINDEXES catalog table 1764
- CURRENTSERVER
 - column of SYSPLAN catalog table 1832
- cursor
 - ASENSITIVE 1193
 - closed state 1402
 - closing
 - CLOSE statement 885
 - CONNECT statement 899
 - error in FETCH 1307
 - error in UPDATE 1530
 - DYNAMIC 1193
 - INSENSITIVE 1193, 1408
 - naming convention 52
 - NO SCROLL 1193, 1410
 - open state 1307
 - opening
 - errors 1402
 - OPEN statement 1400
 - rowset positioning 1196
 - rowset-positioning 1410
 - SCROLL 1193, 1410
 - SENSITIVE 1193
 - SENSITIVE DYNAMIC 1409
 - SENSITIVE STATIC 1409
 - STATIC 1194
 - using
 - current row 1307
 - DECLARE CURSOR statement 1191
 - FETCH statement 1290
 - positions 1307
- cursor-name clause
 - DECLARE CURSOR statement 1192
 - FETCH statement 1302
- CYCLE
 - clause of ALTER SEQUENCE statement 783
- CYCLE clause
 - ALTER TABLE statement 804
 - CREATE SEQUENCE statement 1069
- CYCLE column of SYSSEQUENCES catalog table 1865

D

- DATA CAPTURE clause
 - ALTER TABLE statement 828
 - CREATE TABLE statement 1112
- data compression
 - COMPRESS clause
 - ALTER TABLESPACE statement 844
 - CREATE TABLESPACE statement 1142
- data structures
 - databases 12
 - described 4
 - hierarchy 4
 - index spaces 15
 - indexes 5
 - keys 6
 - table spaces 14
 - types 4
 - views 8
- data type
 - built-in 69
 - casting between 96
 - character string 73
 - compatibility matrix 102
 - CREATE TABLE statement 1087

- data type (*continued*)
 - datetime 87
 - distinct 94
 - graphic string 83
 - list of built-in types 69
 - name, unqualified 58
 - naming convention
 - built-in 51
 - distinct type 52
 - numeric 70
 - promotion 95
 - result column 636
 - results of an operation 119
 - row ID 92
 - unqualified name 58
 - XML values 93
- DATA TYPE clause 1271
- DATA_FORMAT column
 - SYSENVIRONMENT catalog table 1759
- DATA_SHARING_GROUP_NAME session variable 168
- database
 - altering
 - ALTER DATABASE statement 698
 - creating 12, 912
 - default database 12, 56
 - description 12
 - designing
 - using catalog 1998
 - dropping 1259
 - DSNDB04 (default database) 56
 - DSNXSR (XML schema repository) 1921, 1922, 1924, 1925, 1926, 1927, 1928
 - implementing a design 2007
 - limits 1588
 - naming convention 52
 - operations that lock 12
 - privileges
 - granting 1337
 - revoking 1439
 - reasons for defining 12
 - starting and stopping as unit 12
 - TEMP for declared temporary tables 12
 - users who need their own 12
- DATABASE clause
 - ALTER DATABASE statement 698
 - DROP statement 1259
- database descriptor 17
 - contents 17
- database request module (DBRM) 27
- DATABASE
 - clause of GRANT statement 1338
 - clause of REVOKE statement 1440
- databases
 - use of term 12
- DATACAPTURE column of SYSTABLES catalog table 1890
- DATACLAS clause
 - CREATE STOGROUP statement 787, 1075
- DATACLAS column
 - SYSSTOGROUP catalog table 1872
- DATAPEATFACTORF column
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSINDEXESSTATS catalog table 1782
 - SYSINDEXESSTATS_HIST catalog table 1784
- DATASIZE column
 - SYSTABLESPACESTATS catalog table 1901

- DATATYPEID column
 - DATATYPES catalog table 1749
 - SYSOLUMNS catalog table 1723
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSPARMS catalog table 1826
 - SYSSEQUENCES catalog table 1865
- date
 - arithmetic 194
 - data type 87
 - duration 192
 - strings 89, 92
- DATE
 - data type
 - CREATE TABLE statement 1090
 - function 328
- DATE FORMAT clause
 - ALTER PROCEDURE (SQL - native) statement 773
 - CREATE PROCEDURE (SQL - native) statement 1061
- DATE FORMAT field of panel DSNTIP4 255
- date routine
 - CHAR function 302
- DATE_FORMAT session variable 168
- DATE_LENGTH session variable 168
- DATE
 - data type
 - description 87
- datetime
 - arithmetic 193
 - constants 129
 - data types
 - description 87
 - string representation 89
 - EUR (IBM European standard) 89
 - format
 - setting through the CHAR function 302
 - ISO (International Standards Organization) 89
 - JIS (Japanese Industrial Standard) 89
 - LOCAL 89
 - string formats 89
 - USA 89
- datetime host variables
 - data type
 - description 89
- Datetime operands 121
- DAY function 330
- day of week calculation 335
- DAYNAME function 2016
- DAYOFMONTH function 331
- DAYOFWEEK function 332
- DAYOFWEEK_ISO function 333
- DAYOFYEAR function 334
- DAYS function 335
- DB2 books online 2033
- DB2 catalog 1692
- DB2 Information Center for z/OS solutions 2033
- DB2 private protocol access
 - authorization ID 67
- DB2 subsystem
 - local 31
- DB2 version identification, current server 901
- DBADM authority
 - GRANT statement 1337
 - REVOKE statement 1439
- DBADMAUTH column of SYSDBAUTH catalog table 1751
- DBALIAS column
 - LOCATIONS catalog table 1701
- DBCLOB
 - function 336
- DBCLOB (double-byte character large object)
 - data type 84, 1089
 - description 85
 - file reference 165
 - host variable 163
 - locator 163
- DBCS (double-byte character set)
 - ASCII 74
 - EBCDIC 74
 - SQL ordinary identifier 47, 49
 - Unicode 74
- DBCS_CCSID column
 - SYSDATABASE catalog table 1747
 - SYSTABLESPACE catalog table 1896
- DBCTRL authority
 - GRANT statement 1337
 - REVOKE statement 1439
- DBCTRLAUTH column of SYSDBAUTH catalog table 1751
- DBD01 directory table space
 - contents 17
- DBID
 - column of SYSCHECKS catalog table 1711
 - column of SYSDATABASE catalog table 1747
 - column of SYSINDEXES catalog table 1764
 - column of SYSTABLES catalog table 1890
 - column of SYSTABLESPACE catalog table 1896
 - column of SYSTRIGGERS catalog table 1909
- DBID column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- DBINFO
 - clause of ALTER FUNCTION statement 714
 - clause of CREATE FUNCTION statement 932, 953
 - column of SYSROUTINES catalog table 1848
- DBINFO clause
 - ALTER PROCEDURE (external) statement 747
 - CREATE PROCEDURE (external) statement 1028
- DBMAINT authority
 - GRANT statement 1337
 - REVOKE statement 1439
- DBMAINTAUTH column of SYSDBAUTH catalog table 1751
- DBNAME column
 - SYS COPY catalog table 1740
 - SYSINDEXES catalog table 1764
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSLOBSTATS catalog table 1807
 - SYSLOBSTATS_HIST catalog table 1808
 - SYSTABAUTH catalog table 1878
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
 - SYSTABLESPACE catalog table 1896
 - SYSTABLESPACESTATS catalog table 1901
 - SYSTABSTATS catalog table 1907
 - SYSTABSTATS_HIST catalog table 1908
- DBPROTOCOL column
 - SYS PACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- DBRMLIB column of SYSJAVAOPTS catalog table 1789
- DCLGEN subcommand of DSN
 - description 87
- DCOLLID column of SYSPACKDEP catalog table 1820
- DCOLNAME column
 - SYSDEPENDENCIES catalog table 1756

- DCOLNAME column of SYSSEQUENCEDEP catalog table 1867
- DCOLNO column
 - SYSDEPENDENCIES catalog table 1756
- DCONSTNAME column of SYSCONSTDEP catalog table 1736
- DCONTOKEN column of SYSPACKDEP catalog table 1820
- DCREATOR column
 - SYSSEQUENCEDEP catalog table 1867
 - SYSVIEWDEP catalog table 1914
- DDCS (data definition control support) database 19
- DDL (Data Definition Language) 1
- deadlock
 - locks and uncommitted changes 25
- DEBUG_MODE column
 - SYSROUTINES catalog table 1848
 - SYSROUTINES_OPTS catalog table 1859
- DEBUGSESSION privilege
 - GRANT statement 1353
 - REVOKE statement 1456
- DEBUGSESSIONAUTH column
 - SYSUSERAUTH catalog table 1911
- DEC function 344
- DEC15 precompiler option 183
- DEC31
 - column of SYSDBRM catalog table 1754
 - column of SYSPACKAGE catalog table 1810
 - precompiler option 183
- DECFLOAT
 - arithmetic 186
 - data type 71
 - CREATE TABLE statement 1088
 - rounding mode 251
- DECFLOAT function 340
- DECFLOAT_SORTKEY function 342
- decimal
 - constants 127
 - numbers 71
- DECIMAL
 - data type 71
 - CREATE TABLE statement 1088
 - function
 - description 344
- DECIMAL clause
 - ALTER PROCEDURE (SQL - native) statement 773
 - CREATE PROCEDURE (SQL - native) statement 1061
- decimal floating point
 - constants 127
 - numbers 71
- decimal floating-point operands 186
- DECIMAL POINT IS field of panel DSNTIPF 251
- decimal point precompiler option 251
- DECIMAL_ARITHMETIC column
 - SYSENVIRONMENT catalog table 1759
- DECIMAL_ARITHMETIC session variable 168
- DECIMAL_POINT column
 - SYSENVIRONMENT catalog table 1759
- DECIMAL_POINT session variable 168
- DECLARE CURSOR statement
 - description 1191
 - example 1199
- declare default element namespace clause
 - CREATE INDEX statement 997
- DECLARE GLOBAL TEMPORARY TABLE statement
 - description 1202
 - example 1214
- declare namespace clause
 - CREATE INDEX statement 996
- DECLARE STATEMENT statement
 - description 1216
 - example 1216
- DECLARE TABLE statement
 - description 1217
 - example 1219
- DECLARE VARIABLE statement
 - description 1221
 - example 1223
- DECOMPOSITION column
 - SYSIBM.XSROBJECTS table 1922
- DECOMPOSITION_VERSION column
 - SYSIBM.XSROBJECTS table 1922
- decrementing time 196
- DECRYPT_BINARY function 346
- DECRYPT_BIT function 346
- DECRYPT_CHAR function 346
- DECRYPT_DB function 346
- DEFAULT clause
 - ALTER TABLE statement 800
- DEFAULT column
 - SYSCOLUMNS catalog table 1723
- default database (DSNDB04)
 - defining 12
 - implicit specification 56
- DEFAULT REGISTERS clause
 - ALTER PROCEDURE (external) statement 750
 - ALTER PROCEDURE (SQL - external) statement 756
 - ALTER PROCEDURE (SQL - native) statement 766
 - CREATE PROCEDURE (SQL - external) statement 1044
 - CREATE PROCEDURE (SQL - native) statement 1054
- DEFAULT ROLE clause
 - ALTER TRUSTED CONTEXT statement 858
 - CREATE TRUSTED CONTEXT statement 1169
- DEFAULT SECURITY LABEL clause
 - ALTER TRUSTED CONTEXT statement 859
 - CREATE TRUSTED CONTEXT statement 1170
- DEFAULT SPECIAL REGISTERS clause
 - ALTER FUNCTION statement 717
 - CREATE FUNCTION statement 935, 956
 - CREATE PROCEDURE (external) statement 1031
- DEFAULT_DECFLOAT_ROUND_MODE session variable 168
- DEFAULT_DEFAULT_SSID session variable 168
- DEFAULT_LANGUAGE session variable 168
- DEFAULT_LOCALE_LC_CTYPE session variable 168
- DEFAULTROLE column
 - SYSCONTEXT catalog table 1737
- DEFAULTSECURITYLABEL column
 - SYSCONTEXT catalog table 1737
- DEFAULTVALUE column of SYSCOLUMNS catalog table 1723
- DEFER
 - clause of CREATE INDEX statement 1007
- DEFER PREPARE clause
 - ALTER PROCEDURE (SQL - native) statement 766
 - CREATE PROCEDURE (SQL - native) statement 1054
- DEFERPREP column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- DEFERPREPARE column of SYSPACKAGE catalog table 1810
- deferred embedded SQL 3
- define behavior for dynamic SQL statements 64
- DEFINE clause
 - CREATE INDEX statement 1004
 - CREATE TABLESPACE statement 1136

- DEFINER column
 - SYSCONTEXT catalog table 1737
 - SYSOBJROLEDEP catalog table 1809
 - SYSROLES catalog table 1845
- DEFINERTYPE column
 - SYSCONTEXT catalog table 1737
 - SYSOBJROLEDEP catalog table 1809
 - SYSROLES catalog table 1845
- DEFINITION ONLY clause
 - CREATE TABLE statement 1124
 - DECLARE GLOBAL TEMPORARY TABLE statement 1214
- DEGREE
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPLAN catalog table 1832
- DEGREE clause
 - ALTER PROCEDURE (SQL - native) statement 767
 - CREATE PROCEDURE (SQL - native) statement 1055
- DEGREES function 350
- DELETE
 - clause of TRIGGER statement 1153
 - statement
 - description 1224
 - example 1235
- DELETE privilege
 - GRANT statement 1355
 - REVOKE statement 1458
- delete rules 1232
- DELETEAUTH column of SYSTABAUTH catalog table 1878
- DELETERULE column of SYSRELS catalog table 1841
- deleting
 - all rows from a table 1517
 - rows from a table 1224
 - SQL objects 1256
- delimited identifier in SQL 50
- delimiter
 - SQL 50
- DENSE_RANK expression 212
- DENSERANK expression 212
- dependency
 - of objects on each other 1269
- dependent row 20
- dependent tables 20
- DERIVED_FROM column
 - SYSKEYTARGETS catalog table 1793
- DESC clause
 - ALTER TABLE statement 820
 - CREATE INDEX statement 995
 - CREATE TABLE statement 1109
 - select-statement 654
- DESCRIBE CURSOR statement
 - description 1238
 - example 1239
- DESCRIBE INPUT statement
 - prepared statement 1240
- DESCRIBE OUTPUT statement 1243
- DESCRIBE PROCEDURE statement
 - description 1250
 - example 1251
- DESCRIBE statement 1237
 - variables 1243, 1254
- DESCRIBE TABLE statement 1253
- descriptor
 - naming convention 52
- DETERMINISTIC clause
 - ALTER FUNCTION statement 709, 722
 - ALTER PROCEDURE (external) statement 746
 - ALTER PROCEDURE (SQL - external) statement 753
- DETERMINISTIC clause (*continued*)
 - ALTER PROCEDURE (SQL - native) statement 764
 - CREATE FUNCTION statement 928, 949, 977
 - CREATE PROCEDURE (external) statement 1027
 - CREATE PROCEDURE (SQL - external) statement 1041
 - CREATE PROCEDURE (SQL - native) statement 1051
- DETERMINISTIC column of SYSROUTINES catalog table 1848
- DEVTYPE column of SYSCOPY catalog table 1740
- DFSMSHsm (Data Facility Hierarchical Storage Manager)
 - dropping an index or table space 1269
- DIFFERENCE function 351
- digit, description in DB2 47
- DIGITS function 352
- directory
 - description 17
 - table space names 17
- disability xviii
- DISABLE clause
 - ALTER TRUSTED CONTEXT statement 858
 - CREATE TRUSTED CONTEXT statement 1170
- DISABLE DEBUG MODE clause
 - ALTER PROCEDURE (external) statement 750
 - ALTER PROCEDURE (SQL - native) statement 764
 - CREATE PROCEDURE (external) statement 1023
 - CREATE PROCEDURE (SQL - native) statement 1052
- DISALLOW DEBUG MODE clause
 - ALTER PROCEDURE (external) statement 750
 - ALTER PROCEDURE (SQL - native) statement 764
 - CREATE PROCEDURE (external) statement 1023
 - CREATE PROCEDURE (SQL - native) statement 1052
- DISALLOW PARALLEL clause
 - ALTER FUNCTION statement 714
 - CREATE FUNCTION statement 932, 953
- DISCONNECT
 - column of SYSPLAN catalog table 1832
- DISPLAY privilege
 - GRANT statement 1353
 - REVOKE statement 1456
- DISPLAYAUTH column of SYSUSERAUTH catalog table 1911
- DISPLAYDB privilege
 - GRANT statement 1337
 - REVOKE statement 1440
- DISPLAYDBAUTH column of SYSDBAUTH catalog table 1751
- DISTINCT
 - clause of subselect 633
 - keyword
 - AVG function 269
 - COUNT function 271
 - COUNT_BIG function 272
 - MAX function 275
 - MIN function 276
 - STDDEV function 277
 - STDDEV_SAMP function 277
 - SUM function 278
 - VARIANCE function 279
 - VARIANCE_SAMP function 279
- DISTINCT predicate 230
- distinct type 30
 - assignment of values 113
 - casting 96
 - comparison of values 117
 - CREATE TABLE statement 1090
 - creating 1177
 - description 94

- distinct type *(continued)*
 - dropping 1265
 - granting privileges 1359
 - name, unqualified 52, 58
 - naming convention 52
 - promotion 95
 - revoking privileges 1461
 - unqualified name 58
- distributed access
 - restriction 51
- distributed data
 - CONNECT statement 899
 - CURRENT SERVER special register 147
 - RELEASE (connection) statement 1424
 - SET CONNECTION statement 1475
- distributed relational database
 - definition of 31
- Distributed Relational Database Architecture (DRDA) 31
- distributed unit of work
 - connection management 32
 - definition of 32
- DISTRIBUTED_SQL_STRING_DELIMITER session
 - variable 168
- DLOCATION column of SYSPACKDEP catalog table 1820
- DML (Data Manipulation Language) 1
- DNAME column
 - SYSDEPENDENCIES catalog table 1756
 - SYSOBJROLEDEP catalog table 1809
 - SYSPACKDEP catalog table 1820
 - SYSPLANDEP catalog table 1839
 - SYSSEQUENCEDEP catalog table 1867
 - SYSVIEWDEP catalog table 1914
- dormant connection state 33
- DOUBLE data type
 - CREATE TABLE statement 1088
 - description 71
- DOUBLE function 353
- DOUBLE or DOUBLE_PRECISION
 - function 353
- DOUBLE PRECISION data type
 - CREATE TABLE statement 1088
 - description 71
- double precision floating-point number 71
- DOUBLE_PRECISION function 353
- double-byte character
 - LABEL statement 1385
 - truncated during assignment 110
- double-byte character large object (DBCLOB) 85
- DOWNER column
 - SYSDEPENDENCIES catalog table 1756
 - SYSVIEWDEP catalog table 1914
- DOWNER column of SYSPACKDEP catalog table 1820
- DOWNERTYPE column
 - SYSDEPENDENCIES catalog table 1756
 - SYSPACKDEP catalog table 1820
- DRDA access
 - authorization ID 67
 - mixed environment 1601
- DROP ATTRIBUTES clause
 - ALTER TRUSTED CONTEXT statement 861
- DROP CHECK clause
 - ALTER TABLE statement 819
- DROP CLONE clause
 - ALTER TABLE statement 830
- DROP CONSTRAINT clause
 - ALTER TABLE statement 819
- DROP FOREIGN KEY clause
 - ALTER TABLE statement 819
- DROP MATERIALIZED QUERY clause
 - ALTER TABLE statement 828
- DROP PRIMARY KEY clause
 - ALTER TABLE statement 819
- DROP privilege
 - GRANT statement 1338
 - REVOKE statement 1440
- DROP RESTRICT ON DROP clause
 - ALTER TABLE statement 830
- DROP statement
 - description 1256
 - example 1272
- DROP STORAGE clause
 - TRUNCATE statement 1518
- DROP UNIQUE clause
 - ALTER TABLE statement 819
- DROP USE FOR clause
 - ALTER TRUSTED CONTEXT statement 864
- DROP VERSION clause
 - ALTER PROCEDURE (SQL - native) statement 763
- DROPAUTH column of SYSDBAUTH catalog table 1751
- DROPIN privilege
 - GRANT statement 1349
 - REVOKE statement 1451
- DROPINAUTH column of SYSSCHEMAAUTH catalog
 - table 1862
- DSHEMA column
 - SYSDEPENDENCIES catalog table 1756
 - SYSOBJROLEDEP catalog table 1809
- DSHEMA column of SYSSEQUENCEDEP catalog table 1867
- DSN_DET_COST_TABLE
 - columns 1943
- DSN_FILTER_TABLE
 - columns 1949
- DSN_PGRANGE_TABLE
 - columns 1955
- DSN_PGROUPEP_TABLE
 - columns 1957
- DSN_PREDICATE_TABLE
 - columns 1961
- DSN_PTASK_TABLE
 - columns 1966
- DSN_QUERY_TABLE
 - columns 1973
- DSN_QUERYINFO_TABLE
 - columns 1969
- DSN_SORT_TABLE
 - columns 1978
- DSN_SORTKEY_TABLE
 - columns 1975
- DSN_STRUCT_TABLE
 - columns 1989
- DSN_VIEWREF_TABLE
 - columns 1992
- DSN_XMLVALIDATE function 355, 357
- DSNAME
 - column of SYSCOPY catalog table 1740
- DSNDB04 default database 12
- DSNHDECP_NAME session variable 168
- DSNUM column
 - SYSCOPY catalog table 1740
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887

- DSNXSR database
 - SYSIBM.XSRCOMPONENT table 1921
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSROBJECTGRAMMAR table 1925
 - SYSIBM.XSROBJECTHIERARCHIES table 1926
 - SYSIBM.XSROBJECTPROPERTY table 1927
 - SYSIBM.XSROBJECTS table 1922
 - SYSIBM.XSRPROPERTY table 1928
- DSSIZE
 - clause of CREATE TABLESPACE statement 1137
 - column of SYSTABLESPACE catalog table 1896
- DSVOLSER column of SYSCOPY catalog table 1740
- DTBCREATOR column of SYSCONSTDEP catalog table 1736
- DTBNAME column of SYSCONSTDEP catalog table 1736
- DTBOWNER column
 - SYSCONSTDEP catalog table 1736
- DTYPE column
 - SYSCONSTDEP catalog table 1736
 - SYSDEPENDENCIES catalog table 1756
 - SYSOBJROLEDEP catalog table 1809
 - SYSPACKDEP catalog table 1820
 - SYSVIEWDEP catalog table 1914
- DTYPE column of SYSSEQUENCEDEP catalog table 1867
- dual logging
 - description 17
- duplicate rows, UNION clause 662
- duration
 - date 192
 - labeled 192
 - time 192
 - timestamp 192
- DYNAMIC clause
 - DECLARE CURSOR statement 1193
- DYNAMIC RESULT SET clause
 - ALTER PROCEDURE (external) statement 743
 - ALTER PROCEDURE (SQL - native) statement 764
 - CREATE PROCEDURE (SQL - external) statement 1040
 - CREATE PROCEDURE (SQL - native) statement 1052
- DYNAMIC RESULT SETS clause
 - ALTER PROCEDURE (SQL - external) statement 753
 - CREATE PROCEDURE (external) statement 1023
- dynamic SQL
 - description 3, 688
 - DYNAMICRULES bind option 64
 - EXECUTE IMMEDIATE statement 1280
 - EXECUTE statement 1275
 - execution 690
 - INTO clause
 - DESCRIBE statement 1243
 - PREPARE statement 1407
 - invocation of SELECT statement 692
 - preparation 690
 - SQLDA 1656
 - statements allowed 1601
- DYNAMIC_RULES session variable 168
- DYNAMICRULES
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPLAN catalog table 1832
 - dynamic SQL authorization 64
 - option 58
 - unqualified names 58
- DYNAMICRULES behavior 64
- DYNAMICRULES clause
 - ALTER PROCEDURE (SQL - native) statement 767
 - CREATE PROCEDURE (SQL - native) statement 1055

E

- EBCDIC
 - definition 38
 - effect on MBCS and DBCS characters 74
- EBCDIC CODED CHAR SET field of panel DSNTIPF 254
- EBCDIC_CHR function 361
- EBCDIC_STR function 362
- edit routine
 - named in CREATE TABLE statement 1111
 - specified by EDITPROC option 1111
- EDITPROC clause
 - CREATE TABLE statement 1111
- EDPROC column of SYSTABLES catalog table 1890
- ENABLE
 - column of SYSPKSYSTEM catalog table 1830
 - column of SYSPLSYSTEM catalog table 1840
- ENABLE clause
 - ALTER TRUSTED CONTEXT statement 858
 - CREATE TRUSTED CONTEXT statement 1170
- ENABLE column
 - SYSVIEWS catalog table 1915
- ENABLE QUERY OPTIMIZATION clause
 - ALTER TABLE statement 817
 - CREATE TABLE statement 1102
- ENABLED column
 - SYSCONTEXT catalog table 1737
- encoding scheme 38
 - of strings 43
- ENCODING_CCSID column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- ENCODING_SCHEME column
 - SYSDATABASE catalog table 1747
 - SYSDATATYPES catalog table 1749
 - SYSPARMS catalog table 1826
 - SYSTABLES catalog table 1890
 - SYSTABLESPACE catalog table 1896
- ENCODING_SCHEME session variable 168
- ENCRYPT function 363
- ENCRYPT_TDES function 363
- encryption 1171
- ENCRYPTION clause
 - ALTER TRUSTED CONTEXT statement 860
 - CREATE TRUSTED CONTEXT statement 1171
- encryption password 1502
- ENCRYPTION PASSWORD special register 150
- ENCRYPTPSWDS column of LUNAMES catalog table 1705
- END DECLARE SECTION statement
 - description 1273
 - example 1273
- ENDING AT clause
 - ALTER INDEX statement 737
 - ALTER TABLE statement 820, 822
 - CREATE INDEX statement 1005
 - CREATE TABLE statement 1109
- ENFORCED clause
 - ALTER TABLE statement 817
 - CREATE TABLE statement 1102
- ENFORCED column
 - SYSRELS catalog table 1841
- enforcing business rules 20
 - check constraints 24
 - referential constraints 20
 - triggers 24
- entities
 - integrity 20

- ENVID column
 - SYSENVIRONMENT catalog table 1759
 - SYSINDEXES catalog table 1764
 - SYSTRIGGERS catalog table 1909
- EPOCH column of SYSTABLEPART catalog table 1882
- ERASE clause
 - ALTER INDEX statement 732
 - ALTER TABLESPACE statement 850
 - CREATE INDEX statement 1002
 - CREATE TABLESPACE statement 1133
- ERASERULE column
 - SYSINDEXES catalog table 1764
 - SYSTABLESPACE catalog table 1896
- error
 - arithmetic expression 1472
 - closes cursor 1402
 - during FETCH 1307
 - during update 1530
 - numeric conversion 1472
 - signaling 1516
- ERRORBYTE column of SYSSTRINGS catalog table 1874
- ESCAPE clause
 - LIKE predicate 237
- evaluation order 197
- EXCEPT clause 662
- EXCEPTION clause 1330
- EXCHANGE statement
 - description 1274
 - example 1274
- EXCLUDING COLUMN DEFAULTS clause
 - CREATE TABLE statement 1107
 - DECLARE GLOBAL TEMPORARY TABLE statement 1209
- EXCLUDING IDENTITY COLUMN ATTRIBUTES clause
 - CREATE TABLE statement 1106
 - DECLARE GLOBAL TEMPORARY TABLE statement 1209
- EXCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES clause
 - CREATE TABLE statement 1106
- EXCLUSIVE
 - option of LOCK TABLE statement 1387
- exclusive dependence 1434
- executable statement 688, 689
- EXECUTE IMMEDIATE statement
 - description 1280
 - example 1282
- EXECUTE privilege
 - GRANT statement 1341, 1345, 1348
 - REVOKE statement 1443, 1447, 1449
- EXECUTE statement
 - description 1275
 - example 1279
- EXECUTEAUTH column
 - SYSPACKAUTH catalog table 1818
 - SYSPLANAUTH catalog table 1837
 - SYSROUTINEAUTH catalog table 1846
- EXISTS predicate 232
- EXIT handler
 - SQL procedure 1554, 1620
- exit routine
 - named in ALTER TABLE statement 831
 - named in CREATE TABLE statement 1098
- EXITPARM column of SYSFIELDS catalog table 1761
- EXITPARML column of SYSFIELDS catalog table 1761
- EXP function 366
- EXPLAIN
 - column of SYSPACKAGE catalog table 1810
- EXPLAIN (*continued*)
 - statement
 - description 1283
 - example 1288
- EXPLAIN tables 1969
 - DSN_DETCOST_TABLE 1943
 - DSN_FILTER_TABLE 1949
 - DSN_FUNCTION_TABLE 1952
 - DSN_PGRANGE_TABLE 1955
 - DSN_PGROUPE_TABLE 1957
 - DSN_PREDICAT_TABLE 1961
 - DSN_PTASK_TABLE 1966
 - DSN_QUERY_TABLE 1973
 - DSN_SORT_TABLE 1978
 - DSN_SORTKEY_TABLE 1975
 - DSN_STATEMENT_CACHE_TABLE 1981
 - DSN_STATEMNT_TABLE 1985
 - DSN_STRUCT_TABLE 1989
 - DSN_VIEWREF_TABLE 1992
 - overview 1929
 - PLAN_TABLE 1930
- EXPLAINABLE column
 - SYSPACKSTMT catalog table 1823
 - SYSTMT catalog table 1868
- explainable statement
 - description 1283
 - EXPLAIN statement 1285
- EXPLAN column of SYSPLAN catalog table 1832
- exposed name 158
- EXPREDICATE column of SYSPLAN catalog table 1832
- expression
 - arithmetic operators 182
 - CASE 199
 - CAST specification 202
 - concatenation operator 189
 - datetime operands 192
 - decimal floating-point operands 186
 - decimal operands 183
 - DENSE_RANK expression 212
 - DENSERANK expression 212
 - distinct type operands 188
 - floating-point operands 186
 - integer operands 183
 - NEXT VALUE expression 217
 - nextval-expression 217
 - OLAP-specification 212
 - precedence of operation 197
 - PREVIOUS VALUE expression 217
 - prevval-expression 217
 - RANK expression 212
 - ROW CHANGE TIMESTAMP expression 216
 - ROW CHANGE TOKEN expression 216
 - ROW_NUMBER expression 212
 - row-value 222
 - ROWNUMBER expression 212
 - subselect statement 633
 - without operators 182
- expressions 180
- EXTENTS column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLESPACESTATS catalog table 1901
- EXTERNAL ACTION clause
 - ALTER FUNCTION statement 711, 723

- EXTERNAL ACTION clause (*continued*)
 - CREATE FUNCTION statement 929, 950, 978
- EXTERNAL clause
 - ALTER PROCEDURE (external) statement 743
 - CREATE FUNCTION statement 925, 948
 - CREATE PROCEDURE (external) statement 1024
- EXTERNAL NAME clause
 - ALTER FUNCTION statement 707
 - ALTER PROCEDURE (SQL - external) statement 753
 - CREATE PROCEDURE (SQL - external) statement 1040
- EXTERNAL_ACTION column of SYSROUTINES catalog table 1848
- EXTERNAL_NAME column of SYSROUTINES catalog table 1848
- EXTERNAL_SECURITY column
 - SYSROUTINES catalog table 1848
- external-java-routine-name clause
 - ALTER FUNCTION statement 707
 - ALTER PROCEDURE (external) statement 743
 - CREATE FUNCTION statement 925
 - CREATE PROCEDURE (external) statement 1024
- external-program
 - naming convention 52
- external-program-name clause
 - CREATE FUNCTION statement 925
- EXTRACT function 367

- FARINDREF column
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
- FAROFFPOSF column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
- FENCED
 - clause of CREATE FUNCTION statement 928, 949
 - column of SYSROUTINES catalog table 1848
- FENCED clause
 - CREATE PROCEDURE (external) statement 1023
 - CREATE PROCEDURE (SQL - external) statement 1040
- FETCH FIRST clause
 - select-statement 655
- FETCH FIRST n ROWS ONLY clause
 - SELECT INTO statement 1473
- FETCH statement
 - description 1290
 - example 1314
- field description 806
- field procedure
 - comparisons 116
 - named in ALTER TABLE statement 806
 - named in CREATE TABLE statement 1098
- FIELDPROC clause
 - ALTER TABLE statement 806
 - CREATE TABLE statement 1098
- file reference
 - LOB 165
- FILESEQNO column of SYSCOPY catalog table 1740
- FINAL CALL clause
 - ALTER FUNCTION statement 713, 931
 - CREATE FUNCTION statement 952
- FINAL_CALL column of SYSROUTINES catalog table 1848
- FIRST clause
 - FETCH statement 1295
- FIRST ROWSET clause
 - FETCH statement 1299

- FIRSTKEYCARD column
 - SYSINDEXSTATS catalog table 1782
- FIRSTKEYCARDF column
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSINDEXSTATS catalog table 1782
 - SYSINDEXSTATS_HIST catalog table 1784
- fixed-length binary strings 85
- FLDPROC column
 - SYSCOLUMNS catalog table 1723
 - SYSFIELDS catalog table 1761
- FLDTYPE column of SYSFIELDS catalog table 1761
- FLOAT
 - data type
 - CREATE TABLE statement 1088
 - description 71
- FLOAT function 353
- FLOAT_FORMAT column
 - SYSENVIRONMENT catalog table 1759
- floating-point
 - constants 127
 - double precision number 71
 - single precision number 71
- FLOOR function 370
- FOLD column
 - SYSENVIRONMENT catalog table 1759
- FOR
 - clause of CREATE ALIAS statement 906
 - clause of CREATE SYNONYM statement 1077
 - clause of CREATE TABLE statement 1088
 - clause of CREATE TYPE statement 1179
 - clause of EXPLAIN statement 1285
- FOR EACH ROW clause of TRIGGER statement 1156
- FOR EACH ROW ON UPDATE AS ROW CHANGE
 - TIMESTAMP clause
 - ALTER TABLE statement 806
 - CREATE TABLE statement 1095
- FOR EACH STATEMENT clause of TRIGGER statement 1156
- FOR FETCH ONLY clause 674
- FOR host-variable or integer constant clause
 - FETCH statement 1304
- FOR MULTIPLE ROWS clause
 - PREPARE statement 1411
- FOR n ROWS clause
 - EXECUTE statement 1278
 - INSERT statement 1374
- FOR READ ONLY clause 674
- FOR RESULT SET clause of ALLOCATE CURSOR
 - statement 696
- FOR ROW n OF ROWSET clause
 - DELETE statement 1231
 - UPDATE statement 1529
- FOR SINGLE ROW clause
 - PREPARE statement 1411
- FOR statement
 - example 1563
 - SQL procedure 1563
- FOR UPDATE clause
 - NOFOR precompiler option 256
 - select-statement 673
- FOR UPDATE CLAUSE OPTIONAL clause
 - ALTER PROCEDURE (SQL - native) statement 773
 - CREATE PROCEDURE (SQL - native) statement 1061
- FOR UPDATE CLAUSE REQUIRED clause
 - ALTER PROCEDURE (SQL - native) statement 773
 - CREATE PROCEDURE (SQL - native) statement 1061

- FOREIGN KEY clause
 - ALTER TABLE statement 815
 - CREATE TABLE statement 1100
- foreign keys 6
- FOREIGNKEY column of SYSCOLUMNS catalog table 1723
- FORMAT column
 - SYSTABLEPART catalog table 1882
- Fortran application program
 - host variable 160
 - INCLUDE SQLCA 1652
 - varying-length string 74
- FREE LOCATOR statement
 - description 1316
 - example 1316
- free space
 - index 1003
 - table space 847
- FREEPAGE
 - clause of ALTER INDEX statement
 - description 733
 - clause of CREATE INDEX statement
 - description 1003
 - clause of CREATE TABLESPACE statement
 - description 1134
 - column of SYSINDEXPART catalog table 1771
 - column of SYSTABLEPART catalog table 1882
- FREEPAGE clause
 - ALTER TABLESPACE statement
 - description 847
- FREESPACE column
 - SYSLOBSTATS catalog table 1807
 - SYSLOBSTATS_HIST catalog table 1808
- FREQUENCYF column
 - SYSOLDIST catalog table 1715
 - SYSOLDIST_HIST catalog table 1719
 - SYSOLDISTSTATS catalog table 1717
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
- FROM clause
 - DELETE statement 1228
 - PREPARE statement 1412
 - REVOKE statement 1433
 - subselect 638
- FULL OUTER JOIN
 - description 645
 - example 657
 - FROM clause of subselect 645
- FULLKEYCARD column of SYSINDEXSTATS catalog table 1782
- FULLKEYCARDF column
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSINDEXSTATS catalog table 1782
 - SYSINDEXSTATS_HIST catalog table 1784
- fullselect
 - CREATE VIEW statement 1186
 - description 662
 - example 664
 - INSERT statement 1372
- function 392
 - aggregate 174
 - AVG 269
 - column name 153
 - CORRELATION 270
 - COUNT 271
 - COUNT_BIG 272

- function (continued)
 - aggregate (continued)
 - COVARIANCE or COVARIANCE_SAMP 274
 - description 267
 - example 267
 - MAX 275
 - MIN 276
 - STDDEV 277
 - STDDEV_SAMP 277
 - SUM 278
 - VARIANCE or VAR 279
 - VARIANCE_SAMP or VAR_SAMP 279
 - XMLAGG 281
 - built-in 173
 - cast function 174
 - column 174
 - CORRELATION function 270
 - DENSE_RANK function 212
 - DENSERANK function 212
 - description 173, 259
 - invocation 178
 - maximum number in select 1588
 - name, unqualified 58
 - RANK function 212
 - resolution 175
 - ROW_NUMBER function 212
 - ROWNUMBER function 212
 - scalar 174
 - ABS 284
 - ACOS 285
 - ADD_MONTHS 286
 - ASCII 288
 - ASCII CHR 289
 - ASCII_STR 290
 - ASIN 291
 - ATAN 292
 - ATAN2 294
 - ATANH 293
 - BIGINT 295
 - BINARY 297
 - BLOB 299
 - CCSID_ENCODING 300
 - CEIL or CEILING 301
 - CHAR 302
 - CHARACTER_LENGTH 310
 - CLOB 312
 - COALESCE 315
 - COLLATION_KEY 317
 - COMPARE_DECFLOAT 320
 - CONCAT 322
 - COS 326
 - COSH 327
 - DATE 328
 - DAY 330
 - DAYOFMONTH 331
 - DAYOFWEEK 332
 - DAYOFWEEK_ISO 333
 - DAYOFYEAR 334
 - DAYS 335
 - DBCLOB 336
 - DECFLOAT 340
 - DECFLOAT_SORTKEY 342
 - DECIMAL or DEC 344
 - DECRYPT_BINARY 346
 - DECRYPT_BIT 346
 - DECRYPT_CHAR 346
 - DEGREES 350

function (continued)

scalar (continued)

DIFFERENCE 351
 DIGITS 352
 DOUBLE or DOUBLE_PRECISION 353
 DSN_XMLVALIDATE 355, 357
 EBCDIC_CHR 361
 EBCDIC_STR 362
 ENCRYPT_TDES 363
 EXP 366
 EXTRACT 367
 FLOOR 370
 GENERATE_UNIQUE 371
 GETHINT 373
 GETVARIABLE 374
 GRAPHIC 376
 HEX 380
 HOUR 381
 IDENTITY_VAL_LOCAL 382
 IFNULL 387
 INSERT 388
 JULIAN_DAY 394
 LAST_DAY 395
 LCASE 410
 LCASE function 396
 LEFT 397
 LENGTH 401
 LN 402
 LOCATE 403
 LOCATE_IN_STRING 406
 LOG 402
 LOG10 409
 LOWER 410
 LPAD 413
 LTRIM 415
 MAX 416
 MICROSECOND 417
 MIDNIGHT_SECONDS 418
 MIN 419
 MINUTE 420
 MOD 421
 MONTH 423
 MONTHS_BETWEEN 424
 MQPUBLISH 426
 MQPUBLISHXML 429
 MQREADXML 435
 MQRECEIVEXML 441
 MQSENDXML 445
 MQSENDXMLFILE 447
 MQSENDXMLFILECLOB 449
 MQSUBSCRIBE 451
 MQUNSUBSCRIBE 453
 MULTIPLY_ALT 455
 NEXT_DAY 456
 NORMALIZE_DECFLOAT 458
 NORMALIZE_STRING 459
 NULLIF 461
 OVERLAY 462
 POSITION 466
 POSSTR 469
 POWER 471
 QUANTIZE 472
 QUARTER 474
 RADIANS 475
 RAISE_ERROR 476
 RAND 477
 REAL 478

function (continued)

scalar (continued)

REPEAT 480
 REPLACE 482
 RID 485
 RIGHT 486
 ROUND 488
 ROUND_TIMESTAMP 490
 ROWID 493
 RPAD 494
 RTRIM 496
 SECOND 501
 SIGN 502
 SIN 503
 SINH 504
 SMALLINT 505
 SOAPHTTPC and SOAPHTTPV 508
 SOAPHTTPNC and SOAPHTTPNV 509
 SOUNDEX 507
 SPACE 511
 SQRT 512
 STRIP 513
 SUBSTR 515
 SUBSTRING 517
 TAN 522
 TANH 523
 TIME 524
 TIMESTAMP 525
 TIMESTAMP_FORMAT 529
 TIMESTAMP_ISO 534
 TIMESTAMPADD 527
 TIMESTAMPDIF 535
 TO_CHAR 538, 564
 TO_DATE 529, 539
 TOTALORDER 540
 TRANSLATE 541
 TRUNC_TIMESTAMP 546
 TRUNCATE 544
 UCASE 549, 552
 UNICODE 550
 UNICODE_STR 551
 UPPER 552
 VALUE 315
 VARBINARY 556
 VARCHAR 558
 VARCHAR_FORMAT 564
 VARGRAPHIC 569
 WEEK 573
 WEEK_ISO 574
 XMLATTRIBUTES 575
 XMLCOMMENT 576
 XMLCONCAT 577
 XMLDOCUMENT 578
 XMLELEMENT 579
 XMLFOREST 584
 XMLNAMESPACES 587
 XMLPARSE 589
 XMLPI 591
 XMLQUERY 592
 XMLSERIALIZE 596
 XMLTEXT 599
 YEAR 600

string units 76

table 174

ADMIN_TASK_LIST function 601
 ADMIN_TASK_STATUS function 606
 description 600

- function (continued)
 - table (continued)
 - MQREAD function 431
 - MQREADALL function 609
 - MQREADALLCLOB function 611
 - MQREADALLXML 613
 - MQREADCLOB function 433
 - MQRECEIVE function 437
 - MQRECEIVEALL function 615
 - MQRECEIVEALLCLOB function 618
 - MQRECEIVEALLXML 621
 - MQRECEIVECLOB function 439
 - MQSEND function 443
 - types 173
 - unqualified name 58
- FUNCTION clause
 - COMMENT statement 891
 - DROP statement 1259
- function resolution 175
- function table 1283
- FUNCTION_TYPE column
 - SYSROUTINES catalog table 1848
- function, built-in
 - nesting 283
 - scalar
 - description 283
 - example 283
- functions 28
 - best fit 177
 - casting
 - XMLCAST 210
 - CONTAINS 323
 - SCORE 498
 - table
 - XMLTABLE 624
 - VALUE 555
- FUNCTIONS column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832

G

- GBPCACHE clause
 - ALTER INDEX statement 734
 - ALTER TABLESPACE statement 850
 - CREATE INDEX statement 1003
 - CREATE TABLESPACE statement 1135
- GBPCACHE column
 - SYSINDEXPART catalog table 1771
 - SYSTABLEPART catalog table 1882
- general-use programming information, described 2042
- GENERATE KEY USING clause
 - CREATE INDEX statement 996
- GENERATE_UNIQUE function 371
- GENERATED clause
 - ALTER TABLE statement 802
 - CREATE TABLE statement 1095
 - DECLARE GLOBAL TEMPORARY TABLE statement 1207
- GENERIC column of LUNAMES catalog table 1705
- GET DIAGNOSTICS statement
 - description 1317
 - SQL procedure 1565, 1626
- GETHINT function 373
- GETVARIABLE function 374
- GMT (Greenwich Mean Time) 132
- GO TO clause of WHENEVER statement 1539
- GOTO statement
 - example 1627
 - examples 1566
 - SQL procedure 1566, 1627
- GRAMMAR column
 - SYSIBM.XSROBJECTGRAMMAR table 1925
 - SYSIBM.XSROBJECTS table 1922
- GRANT statement
 - collection privileges 1336
 - database privileges 1337
 - description 1333
 - function privileges 1340
 - package privileges 1345
 - plan privileges 1348
 - procedure privileges 1340
 - schema privileges 1349
 - sequence privileges 1351
 - system privileges 1352
 - table privileges 1355
 - USAGE privilege 1359
 - use privileges 1361
 - view privileges 1355
- GRANTEDTS
 - column of SYSSEQUENCEAUTH catalog table 1863
- GRANTEDTS column
 - SYSCOLAUTH catalog table 1714
 - SYSDBAUTH catalog table 1751
 - SYSPLANAUTH catalog table 1837
 - SYSRESAUTH catalog table 1843
 - SYSROUTINEAUTH catalog table 1846
 - SYSSCHEMAAUTH catalog table 1862
 - SYSTABAUTH catalog table 1878
 - SYSUSERAUTH catalog table 1911
- GRANTEE
 - column of SYSSEQUENCEAUTH catalog table 1863
- GRANTEE column
 - SYSCOLAUTH catalog table 1713
 - SYSDBAUTH catalog table 1751
 - SYSPACKAUTH catalog table 1818
 - SYSPLANAUTH catalog table 1837
 - SYSRESAUTH catalog table 1843
 - SYSROUTINEAUTH catalog table 1846
 - SYSSCHEMAAUTH catalog table 1862
 - SYSTABAUTH catalog table 1878
 - SYSUSERAUTH catalog table 1911
- GRANTEETYPE
 - column of SYSSEQUENCEAUTH catalog table 1863
- GRANTEETYPE column
 - SYSCOLAUTH catalog table 1713
 - SYSDBAUTH catalog table 1751
 - SYSPACKAUTH catalog table 1818
 - SYSPLANAUTH catalog table 1837
 - SYSRESAUTH catalog table 1843
 - SYSROUTINEAUTH catalog table 1846
 - SYSSCHEMAAUTH catalog table 1862
 - SYSTABAUTH catalog table 1878
 - SYSUSERAUTH catalog table 1911
- GRANTOR column
 - SYSCOLAUTH catalog table 1713
 - SYSDBAUTH catalog table 1751
 - SYSPACKAUTH catalog table 1818
 - SYSPLANAUTH catalog table 1837
 - SYSRESAUTH catalog table 1843
 - SYSROUTINEAUTH catalog table 1846
 - SYSSCHEMAAUTH catalog table 1862
 - SYSTABAUTH catalog table 1878
 - SYSUSERAUTH catalog table 1911

- GRANTORS
 - column of SYSSEQUENCEAUTH catalog table 1863
- GRANTORTYPE column
 - SYSCOLAUTH catalog table 1714
 - SYSDBAUTH catalog table 1751
 - SYSPACKAUTH catalog table 1818
 - SYSPLANAUTH catalog table 1837
 - SYSRESAUTH catalog table 1843
 - SYSROUTINEAUTH catalog table 1846
 - SYSSCHEMAAUTH catalog table 1862
 - SYSSEQUENCEAUTH catalog table 1863
 - SYSTABAUTH catalog table 1878
 - SYSUSERAUTH catalog table 1911
- GRANULARITY column of SYSTRIGGERS catalog table 1909
- GRAPHIC
 - data type
 - CREATE TABLE statement 1089
 - description 83
 - function 376
 - option of precompiler 254
- graphic string
 - constants 130
 - description 83
- GREATEST function 416
- Greenwich Mean Time (GMT) 132
- group buffer pools
 - described 18
- GROUP BY clause
 - cannot join view 1189
 - subselect
 - description 649
 - results 633
- GROUP_MEMBER column
 - SYS COPY catalog table 1740
 - SYS DATABASE catalog table 1747
 - SYS PACKAGE catalog table 1810
 - SYS PLAN catalog table 1832
- grouping column 649
- GUI symbols 2042

H

- handler
 - SQL procedure 1554, 1620
- handling errors
 - SQL procedure 1554, 1620
- HAVING clause of subselect
 - description 651
 - results 633
- held connection state 33
- HEX function 380
- hexadecimal constant 128
- HIDDEN column of SYSCOLUMNS catalog table 1723
- high encryption 1171
- HIGH2KEY column
 - SYS COLSTATS catalog table 1721
 - SYS COLUMNS catalog table 1723
 - SYS COLUMNS_HIST catalog table 1732
 - SYS KEYTARGETS catalog table 1793
 - SYS KEYTARGETS_HIST catalog table 1798
 - SYS KEYTARGETSTATS catalog table 1796
- HIGHDSNUM column of SYSCOPY catalog table 1740
- HIGHKEY column
 - SYS KEYTARGETSTATS catalog table 1796
- HIGHKEY column of SYS COLSTATS catalog table 1721
- HIGHVALUE column
 - SYS COLDIST catalog table 1715

- HIGHVALUE column (*continued*)
 - SYS COLDIST_HIST catalog table 1719
 - SYS COLDISTSTATS catalog table 1717
 - SYS KEYTGTDIST catalog table 1801
 - SYS KEYTGTDIST_HIST catalog table 1805
 - SYS KEYTGTDISTSTATS catalog table 1803
- HOLD LOCATOR statement
 - description 1363
 - example 1363
- host identifier 50
- host label
 - naming convention 53
- host structure
 - description 171
- host variable
 - colon 160
 - description 160
 - EXECUTE IMMEDIATE statement 1280
 - EXPLAIN statement 1285
 - FETCH statement 1303
 - input 160
 - naming convention 53
 - output 160
 - PREPARE statement 1412
 - SELECT INTO statement 1472
 - substitution for parameter markers 1276
 - XML 164
- HOST_LANGUAGE column
 - SYS ENVIRONMENT catalog table 1759
- host-variable-arrays
 - description 172
- HOSTLANG column
 - SYS DBRM catalog table 1754
 - SYS PACKAGE catalog table 1810
- HOURL function 381
- HPJCOMPILE_OPTS column
 - SYS JAVA_OPTS catalog table 1789
- HTYPE column
 - SYS IBM.XSROBJECTHIERARCHIES table 1926

I

- I/O processing
 - CURRENT DEGREE special register 139
- IBMREQD
 - column of SYSSEQUENCEAUTH catalog table 1863
- IBMREQD column
 - IPLIST catalog table 1697
 - IPNAMES catalog table 1698
 - LOCATIONS catalog table 1701
 - LULIST catalog table 1703
 - LUMODES catalog table 1704
 - LUNAMES catalog table 1705
 - MODESELECT catalog table 1708
 - release dependency indicators 1677
 - SYS AUXRELS catalog table 1709
 - SYS CHECKDEP catalog table 1710
 - SYS CHECKS catalog table 1711
 - SYS CHECKS2 catalog table 1712
 - SYS COLDIST catalog table 1715
 - SYS COLDIST_HIST catalog table 1719
 - SYS COLDISTSTATS catalog table 1717
 - SYS COLSTATS catalog table 1721
 - SYS COLUMNS catalog table 1723
 - SYS COLUMNS_HIST catalog table 1732
 - SYS CONSTDEP catalog table 1736
 - SYS CONTEXT catalog table 1737

IBMREQD column *(continued)*

SYSCONTEXTAUTHIDS catalog table 1739
 SYSCOPY catalog table 1740
 SYSCTXTTRUSTATTRS catalog table 1746
 SYSDATABASE catalog table 1747
 SYSDATATYPES catalog table 1749
 SYSDBAUTH catalog table 1751
 SYSDBRM catalog table 1754
 SYSDEPENDENCIES catalog table 1756
 SYSDDUMMY1 catalog table 1758
 SYSENVIRONMENT catalog table 1759
 SYSFIELDS catalog table 1761
 SYSFOREIGNKEYS catalog table 1763
 SYSINDEXES catalog table 1764
 SYSINDEXES_HIST catalog table 1769
 SYSINDEXPART catalog table 1771
 SYSINDEXPART_HIST catalog table 1775
 SYSINDEXSPACESTATS catalog table 1777
 SYSINDEXSTATS catalog table 1782
 SYSINDEXSTATS_HIST catalog table 1784
 SYSJARCONTENTS catalog table 1786
 SYSJAROBJECTS catalog table 1788
 SYSJAVAOPTS catalog table 1789
 SYSJAVAPATHS catalog table 1790
 SYSKEYCOLUSE catalog table 1791
 SYSKEYS catalog table 1792
 SYSKEYTARGETS catalog table 1793
 SYSKEYTARGETS_HIST catalog table 1798
 SYSKEYTARGETSTATS catalog table 1796
 SYSKEYTGTDIST catalog table 1801
 SYSKEYTGTDIST_HIST catalog table 1805
 SYSKEYTGTDISTSTATS catalog table 1803
 SYSLOBSTATS catalog table 1807
 SYSLOBSTATS_HIST catalog table 1808
 SYSOBJROLEDEP catalog table 1809
 SYSPACKAGE catalog table 1810
 SYSPACKAUTH catalog table 1818
 SYSPACKDEP catalog table 1820
 SYSPACKLIST catalog table 1822
 SYSPACKSTMT catalog table 1823
 SYSPARMS catalog table 1826
 SYSPKSYSTEM catalog table 1830
 SYSPLAN catalog table 1832
 SYSPLANAUTH catalog table 1837
 SYSPLANDEP catalog table 1839
 SYSPLSYSTEM catalog table 1840
 SYSRELS catalog table 1841
 SYSRESAUTH catalog table 1843
 SYSROLES catalog table 1845
 SYSROUTINEAUTH catalog table 1846
 SYSROUTINES catalog table 1848
 SYSROUTINES_OPTS catalog table 1859
 SYSROUTINES_SRC catalog table 1861
 SYSSCHEMAAUTH catalog table 1862
 SYSSEQUENCEDEP catalog table 1867
 SYSSEQUENCES catalog table 1865
 SYSSTMT catalog table 1868
 SYSSTOGROUP catalog table 1872
 SYSSTRINGS catalog table 1874
 SYSSYNONYMS catalog table 1877
 SYSTABAUTH catalog table 1878
 SYSTABCONST catalog table 1881
 SYSTABLEPART catalog table 1882
 SYSTABLEPART_HIST catalog table 1887
 SYSTABLES catalog table 1890
 SYSTABLES_HIST catalog table 1905
 SYSTABLESPACE catalog table 1896

IBMREQD column *(continued)*

SYSTABLESPACESTATS catalog table 1901
 SYSTABSTATS catalog table 1907
 SYSTABSTATS_HIST catalog table 1908
 SYSTRIGGERS catalog table 1909
 SYSUSERAUTH catalog table 1911
 SYSVIEWDEP catalog table 1914
 SYSVIEWS catalog table 1915
 SYSVOLUMES catalog table 1917
 SYSXMLRELS catalog table 1918
 SYSXMLSTRINGS catalog table 1919
 USERNAMES catalog table 1920
 IBMREQD column of SYSCOLAUTH catalog table 1713
 ICBACKUP column of SYSCOPY catalog table 1740
 ICTYPE column of SYSCOPY catalog table 1740
 ICUNIT column of SYSCOPY catalog table 1740
 identifier in SQL
 delimited 50
 ordinary 49
 identity column
 ALTER TABLE statement 803
 CREATE TABLE statement 1095
 IDENTITY_VAL_LOCAL function 382
 IF statement
 example 1568, 1629
 SQL procedure 1568, 1629
 IFNULL function 387
 IGNORE DELETE TRIGGERS clause
 TRUNCATE statement 1518
 IMAGCOPY privilege
 GRANT statement 1338
 REVOKE statement 1440
 IMAGCOPYAUTH column of SYSDBAUTH catalog
 table 1751
 IMMEDIATE clause
 TRUNCATE statement 1518
 IMMEDIATEWRITE column
 SYSPACKAGE catalog table 1810
 SYSPLAN catalog table 1832
 IMPLICIT column
 SYSDATABASE catalog table 1747
 IMPLICIT column of SYSTABLESPACE catalog table 1896
 IMPLICITLY HIDDEN clause
 ALTER TABLE statement 806
 CREATE TABLE statement 1099
 IN
 clause of CREATE AUXILIARY TABLE statement 909
 clause of CREATE TABLE statement 1108
 clause of CREATE TABLESPACE statement 1131
 predicate 119, 234
 IN clause
 ALTER PROCEDURE (SQL - native) statement 763
 CREATE PROCEDURE (external) statement 1021
 CREATE PROCEDURE (SQL - external) statement 1038
 CREATE PROCEDURE (SQL - native) statement 1050
 IN EXCLUSIVE MODE clause of LOCK TABLE
 statement 1387
 IN SHARE MODE clause of LOCK TABLE statement 1386
 INCCSID column of SYSSTRINGS catalog table 1874
 INCLUDE clause
 DELETE statement 1229
 INSERT statement 1371
 MERGE statement 1393
 UPDATE statement 1526
 INCLUDE statement
 assembler declarations 1652
 description 1365

- INCLUDE statement (*continued*)
 - example 1366
 - SQLCA
 - C 1652
 - COBOL 1652
 - Fortran 1652
 - SQLDA
 - assembler 1670
 - C 1670
 - C++ 1670
 - COBOL 1670
 - PL/I 1652, 1670
- INCLUDING COLUMN DEFAULTS clause
 - CREATE TABLE statement 1107
 - DECLARE GLOBAL TEMPORARY TABLE statement 1209
- INCLUDING IDENTITY COLUMN ATTRIBUTES clause
 - CREATE TABLE statement 1106
 - DECLARE GLOBAL TEMPORARY TABLE statement 1209
- INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES clause
 - CREATE TABLE statement 1106
- INCLUSIVE clause
 - ALTER INDEX statement 738
 - ALTER TABLE statement 821, 823, 824, 825
 - CREATE INDEX statement 1007
 - CREATE TABLE statement 1110
- INCREMENT BY
 - clause of ALTER SEQUENCE statement 782
- INCREMENT BY clause
 - CREATE SEQUENCE statement 1068
- INCREMENT column of SYSSEQUENCES catalog table 1865
- incrementing time 196
- index
 - accelerators table 1996
 - altering
 - ALTER INDEX statement 725
 - catalog information about 2001, 2003
 - catalog table 1679
 - creating with CREATE INDEX statement 988
 - dropping 1261
 - name, unqualified 58
 - naming convention 53
 - partitioning 1005
 - renaming with RENAME statement 1428
 - space
 - description 14
 - types
 - changing 725
 - primary 2003
 - unqualified name 58
- INDEX clause
 - ALTER INDEX statement 725
 - COMMENT statement 892
 - CREATE INDEX statement 993
 - DROP statement 1261
- INDEX privilege
 - GRANT statement 1355
 - REVOKE statement 1458
- index spaces 15
- INDEXAUTH column of SYSTABAUTH catalog table 1878
- INDEXBP
 - clause of CREATE DATABASE statement 913
 - column of SYSDATABASE catalog table 1747
- INDEXBP clause
 - ALTER DATABASE statement 698
- indexes
 - described 5
- INDEXSPACE column
 - SYSINDEXSPACESTATS catalog table 1777
- INDEXSPACE column of SYSINDEXES catalog table 1764
- INDEXTYPE column of SYSINDEXES catalog table 1764
- indicator array 171
- indicator variable
 - description 160
 - string expression 1280
- infix operators 182
- INHERIT SPECIAL REGISTERS clause
 - ALTER FUNCTION statement 717
 - ALTER PROCEDURE (external) statement 750
 - ALTER PROCEDURE (SQL - external) statement 756
 - ALTER PROCEDURE (SQL - native) statement 766
 - CREATE FUNCTION statement 935, 956
 - CREATE PROCEDURE (external) statement 1031
 - CREATE PROCEDURE (SQL - external) statement 1044
 - CREATE PROCEDURE (SQL - native) statement 1053
- INITIAL_INSTS column of SYSROUTINES catalog table 1848
- INITIAL_IOS column of SYSROUTINES catalog table 1848
- INNER JOIN
 - description 645
 - example 657
 - FROM clause of subselect 645
- INOUT clause
 - ALTER PROCEDURE (SQL - native) statement 763
 - CREATE PROCEDURE (external) statement 1021
 - CREATE PROCEDURE (SQL - external) statement 1039
 - CREATE PROCEDURE (SQL - native) statement 1050
- input host variable 160
- INPUT SEQUENCE clause
 - ORDER BY clause of subselect 652
- INSENSITIVE clause
 - DECLARE CURSOR statement 1193
 - FETCH statement 1292, 1293
- INSERT clause of CREATE TRIGGER statement 1153
- INSERT function 388
- INSERT privilege
 - GRANT statement 1356
 - REVOKE statement 1458
- insert rule 1375
- INSERT statement
 - description 1367
 - example 1380
- INSERTAUTH column of SYSTABAUTH catalog table 1878
- inserting
 - declaration in a program 1365
 - rows in a table 1367, 1388
- INSTANCE column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACE catalog table 1896
 - SYSTABLESPACESTATS catalog table 1901
- INSTS_PER_INVOC column of SYSROUTINES catalog table 1848
- INT function 392
- INTEGER 392
 - data type
 - CREATE TABLE statement 1087
 - large 71
 - small 71
- integer constants 127
- INTEGER or INT 392
- integrated catalog facility
 - CREATE INDEX statement 1003
- identifier 51
- interactive SQL 3, 692
- INTERSECT clause 662

- INTO clause
 - DESCRIBE CURSOR statement 1238
 - DESCRIBE INPUT statement 1240
 - DESCRIBE PROCEDURE statement 1250
 - DESCRIBE statement 1243, 1254
 - FETCH statement 1303
 - INSERT statement 1370
 - MERGE statement 1392
 - PREPARE statement 1407
 - SELECT INTO statement 1472
 - VALUES INTO statement 1537
- INTO DESCRIPTOR clause
 - FETCH statement 1303, 1305
- INTO host-variable-array clause
 - FETCH statement 1304
- invoke behavior for dynamic SQL statements 64
- IOS_PER_INVOC column of SYSROUTINES catalog table 1848
- IPADDR column
 - IPLIST catalog table 1697
- IPADDR column of IPNAMES catalog table 1698
- IPREFIX column
 - SYSINDEXPART catalog table 1771
 - SYSTABLEPART catalog table 1882
- IS clause
 - COMMENT statement 894
 - LABEL statement 1385
- IS DISTINCT FROM predicate 230
- ISOBID column of SYSINDEXES catalog table 1764
- ISOLATION
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPACKSTMT catalog table 1823
 - column of SYSPLAN catalog table 1832
 - column of SYSSTMT catalog table 1868
- ISOLATION column
 - SYSVIEWS catalog table 1915
- isolation level
 - control by SQL statement
 - DELETE statement 1231
 - INSERT statement 1373
 - SELECT INTO statement 1473
 - select-statement 676
 - UPDATE statement 1529
- ISOLATION LEVEL clause
 - ALTER PROCEDURE (SQL - native) statement 770
 - CREATE PROCEDURE (SQL - native) statement 1057
- isolation-clause
 - DELETE statement 1231
 - INSERT statement 1373
 - SELECT INTO statement 1473
 - UPDATE statement 1529
- ITERATE statement
 - example 1631
 - examples 1569
 - SQL procedure 1569, 1631
- IX_EXTENSION_TYPE column
 - SYSINDEXES catalog table 1764
- IXCREATOR column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSKEYS catalog table 1792
 - SYSTABLEPART catalog table 1882
- IXNAME column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSKEYS catalog table 1792
 - SYSKEYTARGETS catalog table 1793

- IXNAME column (*continued*)
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSKEYTARGETSTATS catalog table 1796
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
 - SYSTABCONST catalog table 1881
 - SYSTABLEPART catalog table 1882
- IXNAME column of SYSRELS catalog table 1841
- IXOWNER column
 - SYSRELS catalog table 1841
 - SYSTABCONST catalog table 1881
- IXSCHEMA column
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSKEYTARGETSTATS catalog table 1796
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803

J

- JAR file
 - unqualified name 58
- JAR file privileges
 - granting 1359
 - revoking 1461
- JAR_DATA column
 - SYSJARDATA catalog table 1787
 - SYSJAROBJECTS catalog table 1788
- JAR_DATA_ROWID column
 - SYSJAROBJECTS catalog table 1788
- JAR_ID column
 - SYSJARCONTENTS catalog table 1786
 - SYSJAROBJECTS catalog table 1788
 - SYSJAVA_OPTS catalog table 1789
 - SYSJAVAPATHS catalog table 1790
 - SYSROUTINES catalog table 1848
- JARSCHEMA column
 - SYSROUTINES catalog table 1848
- JARSCHEMA column
 - SYSJARCONTENTS catalog table 1786
 - SYSJAROBJECTS catalog table 1788
 - SYSJAVA_OPTS catalog table 1789
 - SYSJAVAPATHS catalog table 1790
- JAVA_SIGNATURE column
 - SYSROUTINES catalog table 1848
- JDBC 4
- JOBNAME clause
 - ALTER TRUSTED CONTEXT statement 860
 - CREATE TRUSTED CONTEXT statement 1172
- JOBNAME column of SYSCOPY catalog table 1740
- join operation
 - example 657
 - FROM clause of subselect 647
 - FULL OUTER JOIN
 - FROM clause of subselect 645
 - INNER JOIN
 - FROM clause of subselect 645
 - joining tables 645
 - LEFT OUTER JOIN
 - FROM clause of subselect 645
 - RIGHT OUTER JOIN
 - FROM clause of subselect 645
 - summary of results 647
- JULIAN_DAY function 394

K

- Katakana character 48
- KATAKANA value for EBCDIC CODED CHAR SET 48
- KEEPDYNAMIC column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- key
 - foreign
 - catalog information 2003
 - length
 - maximum 1588
 - partitioning index 737, 1005, 1526
 - primary
 - catalog information 2002
 - defining on a single column 1091
- key-expression clause
 - CREATE INDEX statement 995
- KEYCOLUMNS column of SYSTABLES catalog table 1890
- KEYCOUNT column of SYSINDEXSTATS catalog table 1782
- KEYCOUNTF column
 - SYSINDEXSTATS catalog table 1782
 - SYSINDEXSTATS_HIST catalog table 1784
- KEYGROUPKEYNO column
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
- KEYOBID column of SYSTABLES catalog table 1890
- keys
 - composite key 6
 - defined 6
 - foreign keys 6
 - parent keys 6
 - primary keys 6
 - unique keys 6
- KEYSEQ column
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSKEYTARGETSTATS catalog table 1796
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
- KEYSEQ column of SYSCOLUMNS catalog table 1723
- KEYTARGET_COUNT column
 - SYSINDEXES catalog table 1764
- KEYVALUE column
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
- keywords, reserved 1596

L

- label
 - naming convention 53
- LABEL
 - column of SYSTABLES catalog table 1890
- LABEL column
 - SYSCOLUMNS catalog table 1723
- LABEL statement
 - description 1384
 - example 1385
- labeled duration 192
- labels 1544
- LABELS
 - USING clause of DESCRIBE statement 1244, 1254
 - USING clause of PREPARE statement 1407

- LANGUAGE
 - clause of ALTER FUNCTION statement 708
 - clause of CREATE FUNCTION statement 927, 948
- LANGUAGE clause
 - ALTER PROCEDURE (external) statement 745
 - CREATE PROCEDURE (external) statement 1026
 - CREATE PROCEDURE (SQL - external) statement 1040
- LANGUAGE column
 - SYSROUTINES catalog table 1848
- LANGUAGE SQL clause
 - ALTER FUNCTION statement 722
 - CREATE FUNCTION statement 977
 - CREATE PROCEDURE (SQL - native) statement 1051
- LARGE clause
 - CREATE TABLESPACE statement 1130
- large object (LOB)
 - description 85
- LAST ROWSET clause
 - FETCH statement 1300
- LAST_DAY function 395
- LASTUSED column
 - SYSINDEXSPACESTATS catalog table 1777
- LCASE function 396, 410
- LEAFDIST column
 - SYSINDEXPART_HIST catalog table 1775
- LEAFDIST column of SYSINDEXPART catalog table
 - description 1771
- LEAFFAR column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
- LEAFNEAR column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
- LEAST function 419
- LEAVE statement
 - example 1571, 1632
 - SQL procedure 1571, 1632
- LEFT function 397
- LEFT OUTER JOIN
 - example 657
 - FROM clause of subselect 645
- length attribute of column 74
- LENGTH column
 - SYSCOLUMNS catalog table 1723
 - SYSCOLUMNS_HIST catalog table 1732
 - SYSDATATYPES catalog table 1749
 - SYSFIELDS catalog table 1761
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSPARMS catalog table 1826
- LENGTH function 401
- LENGTH2 column
 - SYSCOLUMNS catalog table 1723
 - SYSCOLUMNS_HIST catalog table 1732
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
- letter, description in DB2 47
- library 2033
- LIKE clause
 - CREATE GLOBAL TEMPORARY TABLE statement 984
 - CREATE TABLE statement 1103
 - DECLARE GLOBAL TEMPORARY TABLE statement 1207
- LIKE predicate 237
- LIMITKEY column
 - SYSINDEXPART catalog table 1771
 - SYSTABLEPART catalog table 1882

- LIMITKEY_INTERNAL column
 - SYSTABLEPART catalog table 1882
- limits, DB2 1588
- LINK_OPTS column
 - SYSROUTINES_OPTS catalog table 1859
- LINKNAME column
 - IPLIST catalog table 1697
 - IPNAMES catalog table 1698
 - LOCATIONS catalog table 1701
 - LULIST catalog table 1703
 - USERNAMES catalog table 1920
- literal 126
- LN function 402
- LOAD privilege
 - GRANT statement 1338
 - REVOKE statement 1440
- LOADAUTH column of SYSDBAUTH catalog table 1751
- LOADLASTTIME column
 - SYSTABLESPACESTATS catalog table 1901
- LOADRLASTTIME column
 - SYSINDEXSPACESTATS catalog table 1777
- LOB
 - restrictions 86
- LOB (large object)
 - clause of CREATE TABLESPACE statement 1131
 - description 85
 - file reference 165
 - host variable 86, 163
 - locator 86, 163
 - retrieving catalog information 2004
- LOBCOLUMNS column of SYSROUTINES catalog table 1848
- local DB2 subsystem 31
- locale
 - CURRENT LOCALE LC_CTYPE special register 140
- LOCATE function 403
- LOCATE_IN_STRING function 406
- location
 - naming convention 53
- LOCATION
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPACKAUTH catalog table 1818
 - column of SYSPACKLIST catalog table 1822
 - column of SYSPACKSTMT catalog table 1823
 - column of SYSPKSYSTEM catalog table 1830
 - column of SYSTABLES catalog table 1890
- LOCATION column
 - LOCATIONS catalog table 1701
- locator
 - LOB 86, 163
 - result set 167
- LOCATOR column of SYSPARMS catalog table 1826
- locator variable
 - freeing 1316
 - holding beyond a unit of work 1363
- lock
 - ALTER TABLESPACE statement 845
 - CREATE TABLESPACE statement 1141
 - description 25
 - during update 1530
 - LOCK TABLE statement 1386
 - object
 - table space (table) 1386
- LOCK TABLE statement
 - description 1386
 - example 1387
- LOCKMAX clause
 - ALTER TABLESPACE statement
 - description 844
 - CREATE TABLESPACE statement
 - description 1141
- LOCKMAX column
 - SYSTABLESPACE catalog table 1896
- LOCKPART
 - clause of ALTER TABLESPACE statement 854
- LOCKPART clause
 - CREATE TABLESPACE statement 1148
- LOCKRULE column of SYSTABLESPACE catalog table 1896
- LOCKSIZE clause
 - ALTER TABLESPACE statement
 - description 845
 - CREATE TABLESPACE statement
 - description 1141
- LOG
 - column of SYSTABLESPACE catalog table 1896
 - function 402
- LOG NO
 - clause of ALTER TABLESPACE statement 854
 - clause of CREATE TABLESPACE statement 1148
- log range directory 17
- LOG YES
 - clause of ALTER TABLESPACE statement 854
 - clause of CREATE TABLESPACE statement 1148
- LOG10 function 409
- LOGGED clause
 - ALTER TABLESPACE statement 845
 - CREATE TABLESPACE statement 1136
- LOGGED column
 - SYSCOPY catalog table 1740
- logical operator 247
- LOGICAL_PART column
 - SYSCOPY catalog table 1740
 - SYSTABLEPART catalog table 1882
- logs
 - described 17
- long column string 84
- LONG VARCHAR data type 1120
 - description 74
- LONG VARGRAPHIC data type 1120
 - description 84
- LOOP statement
 - example 1573, 1633
 - SQL procedure 1573, 1633
- low encryption 1171
- LOW2KEY column
 - SYSCOLSTATS catalog table 1721
 - SYSCOLUMNS catalog table 1723
 - SYSCOLUMNS_HIST catalog table 1732
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSKEYTARGETSTATS catalog table 1796
- LOWDSNUM column of SYSCOPY catalog table 1740
- LOWER function 410
- lowercase character folded to uppercase 48
- LOWKEY column
 - SYSKEYTARGETSTATS catalog table 1796
- LOWKEY column of SYSCOLSTATS catalog table 1721
- LOWVALUE column
 - SYSCOLDIST catalog table 1715
 - SYSCOLDIST_HIST catalog table 1719
 - SYSCOLDISTSTATS catalog table 1717
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805

LOWVALUE column (*continued*)
 SYSKEYTGTDISTSTATS catalog table 1803
 LPAD function 413
 LTRIM function 415
 LUNAME
 column of LULIST catalog table 1703
 column of LUMODES catalog table 1704
 column of LUNAMES catalog table 1705
 column of MODESELECT catalog table 1708

M

MAINTENANCE column
 SYSVIEWS catalog table 1915
 mappings from SQL to XML 257
 materialized-query-definition
 CREATE TABLE statement 1114
 MAX
 aggregate function 275
 scalar function 416
 MAX_FAILURE column
 SYSROUTINES catalog table 1848
 MAXASSIGNEDVAL column of SYSSEQUENCES catalog table 1865
 MAXPARTITIONS clause
 ALTER TABLESPACE statement 847
 CREATE TABLESPACE statement 1138
 MAXPARTITIONS column
 SYSTABLESPACE catalog table 1896
 MAXROWS
 clause of CREATE TABLESPACE statement 1143
 column of SYSTABLESPACE catalog table 1896
 MAXROWS clause
 ALTER TABLESPACE statement 846
 MAXVALUE
 clause of ALTER SEQUENCE statement 783
 clause of CREATE TABLE statement 1097
 MAXVALUE clause
 ALTER TABLE statement 804
 CREATE SEQUENCE statement 1069
 MAXVALUE column of SYSSEQUENCES catalog table 1865
 MEMBER CLUSTER clause
 CREATE TABLESPACE statement 1139
 MERGE statement
 description 1388
 examples 1398
 usage 1397
 message
 precompiler processing of DECLARE TABLE statement 1219
 METATYPE column of SYSDATATYPES catalog table 1749
 MGMTCLAS clause
 CREATE STOGROUP statement 787, 1075
 MGMTCLAS column
 SYSSTOGROUP catalog table 1872
 MICROSECOND function 417
 MIDNIGHT_SECONDS function 418
 MIN
 aggregate function 276
 scalar function 419
 MIN_DIVIDE_SCALE column
 SYSENVIRONMENT catalog table 1759
 MINUTE function 420
 MINVALUE
 clause of ALTER SEQUENCE statement 782
 clause of CREATE TABLE statement 1096

MINVALUE clause
 ALTER TABLE statement 803
 CREATE SEQUENCE statement 1068
 MINVALUE column of SYSSEQUENCES catalog table 1865
 MIXED column
 SYSDBRM catalog table 1754
 SYSPACKAGE catalog table 1810
 mixed data
 convention xxii
 description 74
 in string assignments 110
 LIKE predicate 237
 MIXED DATA
 field of panel DSNTIPF 73, 254
 MIXED_CCSID column
 SYSDATABASE catalog table 1747
 SYSTABLESPACE catalog table 1896
 MIXED_DATA column
 SYSENVIRONMENT catalog table 1759
 MIXED_DATA session variable 168
 MOD function 421
 MODE SQL clause of TRIGGER statement 1157
 MODENAME column
 LUMODES catalog table 1704
 MODESELECT catalog table 1708
 MODESELECT column of LUNAMES catalog table 1705
 MODIFIES SQL DATA clause
 ALTER FUNCTION statement 710
 ALTER PROCEDURE (external) statement 747
 ALTER PROCEDURE (SQL - external) statement 753
 ALTER PROCEDURE (SQL - native) statement 764
 CREATE FUNCTION statement 928
 CREATE PROCEDURE (external) statement 1026
 CREATE PROCEDURE (SQL - external) statement 1041
 CREATE PROCEDURE (SQL - native) statement 1051
 MON1AUTH column of SYSUSERAUTH catalog table 1911
 MON2AUTH column of SYSUSERAUTH catalog table 1911
 MONITOR1 privilege
 GRANT statement 1353
 REVOKE statement 1456
 MONITOR2 privilege
 GRANT statement 1353
 REVOKE statement 1456
 MONITORED STMTS clause
 EXPLAIN statement 1286
 MONTH function 423
 MONTHNAME function 2017
 MONTHS_BETWEEN function 424
 MQPUBLISH function 426
 MQPUBLISHXML function 429
 MQREAD function 431
 MQREADALL function 609
 MQREADALLCLOB function 611
 MQREADALLXML function 613
 MQREADCLOB function 433
 MQREADXML function 435
 MQRECEIVE function 437
 MQRECEIVEALL function 615
 MQRECEIVEALLCLOB function 618
 MQRECEIVEALLXML function 621
 MQRECEIVECLOB function 439
 MQRECEIVEXML function 441
 MQSEND function 443
 MQSENDXML function 445
 MQSENDXMLFILE function 447
 MQSENDXMLFILECLOB function 449
 MQSeries functions 259

MQSUBSCRIBE function 451
 MQUNSUBSCRIBE function 453
 multiple-row-fetch clause
 FETCH statement 1303
 MULTIPLY_ALT function 455

N

NACTIVE column
 SYSINDEXSPACESTATS catalog table 1777
 SYSTABLESPACE catalog table
 description 1896
 SYSTABLESPACESTATS catalog table 1901
 SYSTABSTATS catalog table 1907
 NACTIVEF column
 SYSTABLESPACE catalog table
 description 1896
 NAME
 column of SYSCOLDIST catalog table 1715
 column of SYSCOLDISTSTATS catalog table 1717
 column of SYSCOLSTATS catalog table 1721
 column of SYSCOLUMNS catalog table 1723
 column of SYSSEQUENCEAUTH catalog table 1863
 NAME clause
 CREATE FUNCTION statement 925
 CREATE PROCEDURE (external) statement 1024
 NAME column
 SYSCOLDIST_HIST catalog table 1719
 SYSCOLUMNS_HIST catalog table 1732
 SYSCONTEXT catalog table 1737
 SYSCXTXTTRUSTATTRS catalog table 1746
 SYSDATABASE catalog table 1747
 SYSDATATYPES catalog table 1749
 SYSDBAUTH catalog table 1751
 SYSDBRM catalog table 1754
 SYSFIELDS catalog table 1761
 SYSINDEXES catalog table 1764
 SYSINDEXES_HIST catalog table 1769
 SYSINDEXSPACESTATS catalog table 1777
 SYSINDEXSTATS catalog table 1782
 SYSINDEXSTATS_HIST catalog table 1784
 SYSLOBSTATS catalog table 1807
 SYSLOBSTATS_HIST catalog table 1808
 SYSPACKAGE catalog table 1810
 SYSPACKAUTH catalog table 1818
 SYSPACKLIST catalog table 1822
 SYSPACKSTMT catalog table 1823
 SYSPARMS catalog table 1826
 SYSPKSYSTEM catalog table 1830
 SYSPLAN catalog table 1832
 SYSPLANAUTH catalog table 1837
 SYSRESAUTH catalog table 1843
 SYSROLES catalog table 1845
 SYSROUTINES catalog table 1848
 SYSSEQUENCES catalog table 1865
 SYSSTMT catalog table 1868
 SYSSTOGROUP catalog table 1872
 SYSSYNONYMS catalog table 1877
 SYSTABLES catalog table 1890
 SYSTABLES_HIST catalog table 1905
 SYSTABLESPACE catalog table 1896
 SYSTABLESPACESTATS catalog table 1901
 SYSTABSTATS catalog table 1907
 SYSTABSTATS_HIST catalog table 1908
 SYSTRIGGERS catalog table 1909
 SYSVIEWS catalog table 1915
 names, prepared SQL statements 1216

NAMES
 USING clause of DESCRIBE statement 1244, 1254
 USING clause of PREPARE statement 1407
 naming convention
 SQL 51
 NEARINDREF column
 SYSTABLEPART catalog table 1882
 SYSTABLEPART_HIST catalog table 1887
 NEAROFFPOSF column
 SYSINDEXPART catalog table 1771
 SYSINDEXPART_HIST catalog table 1775
 nested table expressions 640
 new and changed tables 1692
 NEW AS clause of CREATE TRIGGER statement 1155
 new line control character 48
 NEW TABLE AS clause of CREATE TRIGGER statement 1155
 NEW TABLE clause 1164
 NEWAUTHID column of USERNAMES catalog table 1920
 NEWFUN session variable 168
 NEXT clause
 FETCH statement 1294
 NEXT ROWSET clause
 FETCH statement 1298
 NEXT VALUE expression
 definition 217
 NEXT_DAY function 456
 NLEAF column
 SYSINDEXES catalog table
 description 1764
 SYSINDEXES_HIST catalog table 1769
 SYSINDEXSPACESTATS catalog table 1777
 SYSINDEXSTATS catalog table 1782
 SYSINDEXSTATS_HIST catalog table 1784
 NLEVELS column
 SYSINDEXES catalog table
 description 1764
 SYSINDEXES_HIST catalog table 1769
 SYSINDEXSPACESTATS catalog table 1777
 SYSINDEXSTATS catalog table 1782
 SYSINDEXSTATS_HIST catalog table 1784
 NO ACTION delete rule
 CREATE TABLE statement 1101
 NO CACHE
 clause of ALTER SEQUENCE statement 784
 NO CACHE clause
 ALTER TABLE statement 805
 CREATE SEQUENCE statement 1070
 NO CASCADE BEFORE clause of CREATE TRIGGER
 statement 1153
 NO COLLID clause
 ALTER FUNCTION statement 714
 ALTER PROCEDURE (external) statement 747
 ALTER PROCEDURE (SQL - external) statement 754
 CREATE FUNCTION statement 933, 954
 NO CYCLE
 clause of ALTER SEQUENCE statement 783
 NO CYCLE clause
 ALTER TABLE statement 804
 CREATE SEQUENCE statement 1069
 NO DBINFO clause
 ALTER FUNCTION statement 714
 ALTER PROCEDURE (external) statement 747
 CREATE FUNCTION statement 932, 953
 CREATE PROCEDURE (external) statement 1028
 CREATE PROCEDURE (SQL - external) statement 1041
 NO DEFAULT ROLE clause
 ALTER TRUSTED CONTEXT statement 858

NO DEFAULT ROLE clause *(continued)*
 CREATE TRUSTED CONTEXT statement 1169
 no encryption 1171
 NO EXTERNAL ACTION clause
 ALTER FUNCTION statement 711, 723
 CREATE FUNCTION statement 929, 950, 978
 NO FINAL CALL clause
 ALTER FUNCTION statement 713, 931
 CREATE FUNCTION statement 952
 NO MAXVALUE
 clause of ALTER SEQUENCE statement 783
 clause of CREATE TABLE statement 1097
 NO MAXVALUE clause
 ALTER TABLE statement 804
 CREATE SEQUENCE statement 1069
 NO MINVALUE
 clause of ALTER SEQUENCE statement 782
 clause of CREATE TABLE statement 1096
 NO MINVALUE clause
 ALTER TABLE statement 803
 CREATE SEQUENCE statement 1068
 NO ORDER
 clause of ALTER SEQUENCE statement 784
 clause of CREATE TABLE statement 1098
 NO ORDER clause
 ALTER TABLE statement 805
 CREATE SEQUENCE statement 1070
 NO PACKAGE PATH clause
 ALTER FUNCTION statement 711
 ALTER PROCEDURE (external) statement 746
 CREATE FUNCTION statement 930, 951
 CREATE PROCEDURE (external) statement 1028
 NO SCRATCHPAD clause
 ALTER FUNCTION statement 712
 CREATE FUNCTION statement 930, 951
 NO SCROLL clause
 DECLARE CURSOR statement 1193
 NO SQL clause
 ALTER FUNCTION statement 710
 ALTER PROCEDURE (external) statement 747
 CREATE FUNCTION statement 928, 950
 CREATE PROCEDURE (external) statement 1026
 NOCACHE clause
 CREATE SEQUENCE statement 1073
 CREATE TABLE statement 1124
 NOCOLLID clause
 CREATE PROCEDURE (external) statement 1028
 CREATE PROCEDURE (SQL - external) statement 1041
 NOCYCLE clause
 CREATE SEQUENCE statement 1073
 CREATE TABLE statement 1124
 NODEFER PREPARE clause
 ALTER PROCEDURE (SQL - native) statement 766
 CREATE PROCEDURE (SQL - native) statement 1054
 NOFOR option
 precompiler 256
 NOGRAPHIC option of precompiler 254
 NOMAXVALUE clause
 CREATE SEQUENCE statement 1073
 CREATE TABLE statement 1124
 NOMINVALUE clause
 CREATE SEQUENCE statement 1073
 CREATE TABLE statement 1124
 nonexecutable statement 688, 689
 NOORDER clause
 CREATE SEQUENCE statement 1073
 CREATE TABLE statement 1124
 NORMALIZE_DECFLOAT function 458
 NORMALIZE_STRING function 459
 NOT ATOMIC clause
 compound statement of an SQL procedure 1554, 1620
 NOT ATOMIC CONTINUE ON SQLEXCEPTION clause
 INSERT statement 1374
 MERGE statement 1396
 PREPARE statement 1412
 NOT CLUSTER
 clause of ALTER INDEX statement 734
 NOT CLUSTER clause
 CREATE INDEX statement 999
 NOT DETERMINISTIC clause
 ALTER FUNCTION statement 709, 722
 ALTER PROCEDURE (external) statement 746
 ALTER PROCEDURE (SQL - external) statement 753
 ALTER PROCEDURE (SQL - native) statement 764
 CREATE FUNCTION statement 928, 949, 977
 CREATE PROCEDURE (external) statement 1027
 CREATE PROCEDURE (SQL - external) statement 1041
 CREATE PROCEDURE (SQL - native) statement 1051
 NOT ENFORCED clause
 ALTER TABLE statement 817
 CREATE TABLE statement 1102
 NOT FOUND clause of WHENEVER statement 1539
 NOT LOGGED clause
 ALTER TABLESPACE statement 845
 CREATE TABLESPACE statement 1136
 NOT NULL CALL clause
 CREATE FUNCTION statement 938, 956, 981
 NOT NULL clause
 ALTER TABLE statement 806
 CREATE GLOBAL TEMPORARY TABLE statement 984
 CREATE TABLE statement
 description 1091
 DECLARE GLOBAL TEMPORARY TABLE statement 1207
 NOT PADDED
 clause of ALTER INDEX statement 735
 NOT PADDED clause
 CREATE INDEX statement 999
 NOT VARIANT clause
 CREATE FUNCTION statement 938, 956, 981
 CREATE PROCEDURE (external) statement 1033
 CREATE PROCEDURE (SQL - external) statement 1045
 CREATE PROCEDURE (SQL - native) statement 1063
 NOT VOLATILE
 clause of CREATE TABLE statement 1113
 NOT VOLATILE clause
 ALTER TABLE statement 829
 NPAGES column
 SYSINDEXSPACESTATS catalog table 1777
 SYSTABLES catalog table
 description 1890
 SYSTABLESPACESTATS catalog table 1901
 SYSTABSTATS catalog table 1907
 SYSTABSTATS_HIST catalog table 1908
 NPAGESF column
 SYSCOPY catalog table 1740
 SYSTABLES catalog table 1890
 SYSTABLES_HIST catalog table 1905
 NTABLES column of SYSTABLESPACE catalog table 1896
 NULL
 CAST specification 202
 predicate 243
 NULL CALL clause
 CREATE FUNCTION statement 938, 956, 981
 CREATE PROCEDURE (external) statement 1033

- assigned to host variable 1472
 - assignment 102
 - description 70
 - duplicate rows 633
 - grouping columns 649
 - specified by indicator variable 160
 - NULL_CALL column of SYSROUTINES catalog table 1848
 - NULLIF function 461
 - NULLS column
 - SYSOLUMNS catalog table 1723
 - SYSOLUMNS_HIST catalog table 1732
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - NULLS LAST clause
 - ALTER TABLE statement 820
 - CREATE TABLE statement 1109
 - NUM_DEP_MQTS column
 - SYSROUTINES catalog table 1848
 - SYSTABLES catalog table 1890
 - numbers
 - data types
 - string representation 72
 - subnormal numbers 73
 - numbers in SQL 70
 - NUMCOLUMNS column
 - SYSOLDIST catalog table 1715
 - SYSOLDIST_HIST catalog table 1719
 - SYSOLDISTSTATS catalog table 1717
 - numeric
 - assignments 105
 - comparisons 114
 - conversion errors 1472
 - data type 70
 - NUMERIC data type
 - CREATE TABLE statement 1088
 - description 71
 - NUMKEYS column
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
 - NUMPARTS
 - clause of CREATE TABLESPACE statement 1139
- O**
- OBID
 - clause of CREATE TABLE statement 1112
 - column of SYSCHECKS catalog table 1711
 - column of SYSINDEXES catalog table 1764
 - column of SYSTABLES catalog table 1890
 - column of SYSTABLESPACE catalog table 1896
 - column of SYSTRIGGERS catalog table 1909
- object name, unqualified 58
- object ownership 60
- object table 154
- OBJECTOWNERTYPE column
 - SYSCONTEXT catalog table 1737
- OBTYPE column of SYSRESAUTH catalog table 1843
- OCTETS 76
- ODBC (Open Database Connectivity) 3
- OLAP-specification
 - expression 212
- OLD AS clause of TRIGGER statement 1154
- OLD TABLE AS clause of CREATE TRIGGER statement 1155
- OLD TABLE clause 1164

- OLDEST_VERSION
 - column of SYSTABLESPACE catalog table 1896
- OLDEST_VERSION column
 - SYSCOPY catalog table 1740
 - SYSINDEXES catalog table 1764
 - SYSINDEXPART catalog table 1771
 - SYSTABLEPART catalog table 1882
- ON clause
 - CREATE INDEX statement 994
 - CREATE TRIGGER statement 1154
 - joining tables 645
- ON COMMIT clause
 - DECLARE GLOBAL TEMPORARY TABLE statement 1210
- ON DELETE clause
 - ALTER TABLE statement 816
 - CREATE TABLE statement 1101
- ON ROLLBACK RETAIN CURSORS clause
 - SAVEPOINT statement 1468
- ON ROLLBACK RETAIN LOCKS clause
 - SAVEPOINT statement 1469
- ON search condition
 - MERGE statement 1394
- ON TABLE clause
 - GRANT statement 1356
 - REVOKE statement 1459
- one-phase commit 31
- online 2033
- online books 2033
- OPEN
 - statement
 - description 1400
 - example 1404
- open cursor 1307
- Open Database Connectivity (ODBC) 3
- operands
 - datetime 192
 - decimal 183
 - decimal floating-point 186
 - distinct type 188
 - floating-point 186
 - integer 183
 - XML 122
- operation
 - SQL
 - assignment 102
 - comparison 114
 - description 102
- operational form of SQL statements 1
- OPERATIVE column
 - SYPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- operator
 - arithmetic 182
- OPTHINT clause
 - ALTER PROCEDURE (SQL - native) statement 770
 - CREATE PROCEDURE (SQL - native) statement 1058
- OPTHINT column
 - SYPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- optimization hints 770, 1058
- OPTIMIZE FOR n ROWS clause 675
- OR truth table 247
- ORDER
 - clause of ALTER SEQUENCE statement 784
 - clause of CREATE TABLE statement 1098
- ORDER BY clause
 - subselect 652

- ORDER clause
 - ALTER TABLE statement 805
 - CREATE SEQUENCE statement 1070
- ORDER column of SYSSEQUENCES catalog table 1865
- ORDER OF clause
 - ORDER BY clause of subselect 652
- order of evaluation, operators 197
- order of statements in a compound statement 1554, 1620
- ORDERING column
 - SYSKEYTARGETS catalog table 1793
- ORDERING column of SYSKEYS catalog table 1792
- ORDINAL column
 - SYSJAVAPATHS catalog table 1790
- ORDINAL column of SYSPARMS catalog table 1826
- ordinary identifier in SQL 49
- ORGRATIO column
 - SYSLOBSTATS catalog table 1807
 - SYSLOBSTATS_HIST catalog table 1808
- ORIGIN column of SYSROUTINES catalog table 1848
- ORIGINAL_ENCODING_CCSID column
 - SYSENVIRONMENT catalog table 1759
- OTYPE column of SYSCOPY catalog table 1740
- OUT clause
 - ALTER PROCEDURE (SQL - native) statement 763
 - CREATE PROCEDURE (external) statement 1021
 - CREATE PROCEDURE (SQL - external) statement 1039
 - CREATE PROCEDURE (SQL - native) statement 1050
- OUTCCSID column of SYSSTRINGS catalog table 1874
- outer join
 - FULL OUTER JOIN
 - example 657
 - FROM clause of subselect 645
 - LEFT OUTER JOIN
 - example 657
 - FROM clause of subselect 645
 - RIGHT OUTER JOIN
 - example 657
 - FROM clause of subselect 645
- output host variable 160
- OVERLAY function 462
- OVERRIDING USER VALUE
 - clause of INSERT statement 1371
- OWNER
 - column of SYSDATATYPES catalog table 1749
 - column of SYSINDEXSTATS catalog table 1782
 - column of SYSINDEXSTATS_HIST catalog table 1784
 - column of SYSJAROBJECTS catalog table 1788
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPARMS catalog table 1826
 - column of SYSROUTINES catalog table 1848
 - column of SYSSEQUENCES catalog table 1865
 - column of SYSTABSTATS catalog table 1907
 - column of SYSTABSTATS_HIST catalog table 1908
 - column of SYSTRIGGERS catalog table 1909
- OWNER column
 - SYSINDEXES catalog table 1764
 - SYSJAVAPATHS catalog table 1790
 - SYSTABLES catalog table 1890
 - SYSVIEWS catalog table 1915
- OWNERTYPE column
 - SYSCONSTDEP catalog table 1736
 - SYSDATABASE catalog table 1749
 - SYSINDEXES catalog table 1764
 - SYSJAROBJECTS catalog table 1788
 - SYSPACKAGE catalog table 1810
 - SYSPARMS catalog table 1826
 - SYSROUTINES catalog table 1848

- OWNERTYPE column (*continued*)
 - SYSSEQUENCES catalog table 1865
 - SYSTABLES catalog table 1890
 - SYSTABLESPACE catalog table 1909
 - SYSVIEWDEP catalog table 1914
 - SYSVIEWS catalog table 1915

P

- PACKADM authority
 - GRANT statement 1336
 - REVOKE statement 1437
- package
 - binding
 - remote 67
 - dropping 1262
 - invalidated
 - ALTER TABLE statement 836
 - privileges
 - granting 1345
 - remote bind 67
 - revoking 1447
- PACKAGE clause
 - COMMENT statement 893
 - DROP statement 1262
- PACKAGE OWNER clause
 - ALTER PROCEDURE (SQL - native) statement 765
 - CREATE PROCEDURE (SQL - native) statement 1053
- PACKAGE PATH clause
 - ALTER FUNCTION statement 711
 - ALTER PROCEDURE (external) statement 746
 - CREATE FUNCTION statement 930, 951
 - CREATE PROCEDURE (external) statement 1028
- PACKAGE_NAME session variable 168
- PACKAGE_SCHEMA session variable 168
- PACKAGE_VERSION session variable 168
- package-name
 - naming convention 53
- PACKAGE
 - clause of GRANT statement 1345
 - clause of REVOKE statement 1447
- PACKAGEPATH column
 - SYSROUTINES catalog table 1848
- packages 27
- PAD_NUL_TERMINATED session variable 168
- PADDED clause
 - ALTER INDEX statement 735
 - CREATE INDEX statement 999
- PADDED column
 - SYSINDEXES catalog table 1764
- page set
 - description 14
- PAGESAVE column
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
- PARALLEL column of SYSROUTINES catalog table 1848
- parallel processing
 - SET CURRENT DEGREE statement 1482
- Parallel Sysplex
 - group buffer pool 18
- parameter
 - passing to stored procedure 877, 1550, 1615
- PARAMETER CCSID ASCII clause
 - ALTER PROCEDURE (SQL - native) statement 765
- PARAMETER CCSID clause
 - CREATE FUNCTION statement 924, 947, 964, 977
 - CREATE PROCEDURE (external) statement 1023

- PARAMETER CCSID clause *(continued)*
 - CREATE PROCEDURE (SQL - external) statement 1040
 - CREATE PROCEDURE (SQL - native) statement 1052
- PARAMETER CCSID EBCDIC clause
 - ALTER PROCEDURE (SQL - native) statement 765
- PARAMETER CCSID UNICODE clause
 - ALTER PROCEDURE (SQL - native) statement 765
- parameter marker
 - CAST specification 202
 - description 1413
 - EXECUTE statement 1276
 - EXPLAIN statement 1285
 - host variables in dynamic SQL 162
 - obtaining information with DESCRIBE INPUT 1240
 - OPEN statement 1401
 - PREPARE statement 1413
 - rules 1413
- PARAMETER STYLE clause
 - ALTER FUNCTION statement 708
 - ALTER PROCEDURE (external) statement 745
 - CREATE FUNCTION statement 927, 948
 - CREATE PROCEDURE (external) statement 1026
- PARAMETER STYLE DB2SQL clause
 - CREATE FUNCTION statement 938, 956
 - CREATE PROCEDURE (external) statement 1033
- PARAMETER VARCHAR clause
 - CREATE FUNCTION statement 924, 947
 - CREATE PROCEDURE (external) statement 1023
- PARAMETER_CCSID column
 - SYSROUTINES catalog table 1848
- PARAMETER_STYLE column of SYSROUTINES catalog table 1848
- PARAMETER_VARCHARFORM column
 - SYSROUTINES catalog table 1848
- parameter-name
 - naming convention 53
- parent keys 6, 20
- parent row 20
- parent table 20
- PARENTS column of SYSTABLES catalog table 1890
- PARAM_COUNT column of SYSROUTINES catalog table 1848
- PARAM_SIGNATURE column of SYSROUTINES catalog table 1848
- PARAM1 - PARAM30 columns of SYSROUTINES catalog table 1848
- PARMLIST column
 - SYSFIELDS catalog table 1761
- PARAMNAME column of SYSPARMS catalog table 1826
- PART
 - clause of CREATE AUXILIARY TABLE statement 910
- PART clause
 - CREATE INDEX statement 1012
 - CREATE TABLE statement 1124
 - CREATE TABLESPACE statement 1148
 - synonym for PARTITION clause 1387
- partition
 - maximum size 1137
- PARTITION
 - clause of ALTER INDEX statement 737
 - clause of CREATE INDEX statement 1005
 - clause of CREATE TABLESPACE statement 1140
 - clause of LOCK TABLE statement 1386
- PARTITION BY RANGE
 - clause of CREATE INDEX statement 1005
- PARTITION BY RANGE clause
 - ALTER TABLE statement 819
 - CREATE TABLE statement 1108
- PARTITION BY SIZE clause
 - CREATE TABLE statement 1111
- PARTITION clause
 - ALTER TABLE statement 820
 - CREATE TABLE statement 1109
- PARTITION column
 - SYSCOLDISTSTATS catalog table 1717
 - SYSVOLSTATS catalog table 1721
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSINDEXSTATS catalog table 1782
 - SYSINDEXSTATS_HIST catalog table 1784
 - SYSKEYTARGETSTATS catalog table 1796
 - SYSKEYTGTDISTSTATS catalog table 1803
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLESPACE catalog table 1896
 - SYSTABLESPACESTATS catalog table 1901
 - SYSTABSTATS catalog table 1907
 - SYSTABSTATS_HIST catalog table 1908
- PARTITION column of SYSAUXRELS catalog table 1709
- partition order
 - retrieving
 - catalog information 1999
- partition-by-clause
 - CREATE TABLE statement 1108
- partition-by-growth table space 1143
- PARTITIONED clause
 - CREATE INDEX statement 999
- partitioned table spaces
 - described 14
- PARTKEY_COLSEQ column
 - SYSOLUMNS catalog table 1723
- PARTKEY_ORDERING column
 - SYSOLUMNS catalog table 1723
- PARTKEYCOLNUM column
 - SYSTABLES catalog table 1890
- PASSWORD column
 - USERNAMES catalog table 1920
- password, encryption 1502
- PATH column
 - SYSJAROBJECTS catalog table 1788
- PATHSCHEMAS column
 - SYSCHECKS2 catalog table 1712
 - SYSENVIRONMENT catalog table 1759
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
 - SYSVIEWS catalog table 1915
- PCTFREE
 - clause of ALTER INDEX statement 733
 - clause of CREATE INDEX statement 1003
 - clause of CREATE TABLESPACE statement 1135
 - column of SYSINDEXPART catalog table 1771
 - column of SYSTABLEPART catalog table 1882
- PCTFREE clause
 - ALTER TABLESPACE statement 847
- PCTTIMESTAMP column of SYSPACKAGE catalog table 1810
- PCTPAGES column
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
 - SYSTABSTATS catalog table 1907
- PCTROWCOMP column
 - SYSTABLES catalog table
 - description 1890
 - SYSTABLES_HIST catalog table 1905
 - SYSTABSTATS catalog table 1907

- PDSNAME column
 - SYSDBRM catalog table 1754
 - SYSPACKAGE catalog table 1810
- PE_CLASS_PATTERN column
 - SYSJAVAPATHS catalog table 1790
- PE_JAR_ID column
 - SYSJAVAPATHS catalog table 1790
- PE_JARSHEMA column
 - SYSJAVAPATHS catalog table 1790
- PERCACTIVE column
 - SYSTABLEPART catalog table
 - description 1882
 - SYSTABLEPART_HIST catalog table 1887
- PERCDROP column
 - SYSTABLEPART catalog table
 - description 1882
 - SYSTABLEPART_HIST catalog table 1887
- PERIOD option of precompiler 251
- PGSIZE column
 - SYSINDEXES catalog table 1764
 - SYSTABLESPACE catalog table 1896
- PIECESIZE clause
 - ALTER INDEX statement 729
 - CREATE INDEX statement 1008
- PIECESIZE column of SYSINDEXES catalog table 1764
- PIT_RBA column of SYSCOPY catalog table 1740
- PKSIZE column of SYSPACKAGE catalog table 1810
- PL/I application program
 - host structure 171
 - host variable
 - description 160
 - host-variable-arrays 172
 - INCLUDE SQLCA 1652
 - INCLUDE SQLDA 1670
 - varying-length string 74
- PLAN
 - clause of EXPLAIN statement 1284
- PLAN clause
 - COMMENT statement 892
- plan element 1492
- plan table 1283
 - column descriptions 1930
 - creating 1930
 - format 1930
- PLAN_NAME session variable 168
- PLAN_TABLE
 - column descriptions 1930
- plan-name
 - naming convention 53
- PLAN
 - clause of GRANT statement 1348
 - clause of REVOKE statement 1449
- PLANNAME column
 - MODESELECT catalog table 1708
 - SYSPACKLIST catalog table 1822
- PLCREATOR column
 - SYSDBRM catalog table 1754
 - SYSSTMT catalog table 1868
- PLCREATOR TYPE column
 - SYSDBRM catalog table 1754
 - SYSSTMT catalog table 1868
- PLENTRIES column of SYSPLAN catalog table 1832
- PLNAME column
 - SYSDBRM catalog table 1754
 - SYSSTMT catalog table 1868
- PLSIZE column of SYSPLAN catalog table 1832
- POBJECT_LIB column
 - SYSJAVAOPTS catalog table 1789
- point of consistency 25
 - description 25
- PORT column
 - LOCATIONS catalog table 1701
- POSITION function 466
- POSSTR function 469
- POWER function 471
- PQTY column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
- precedence of operators 197
- PRECISION column
 - SYSSEQUENCES catalog table 1865
- precision of numbers
 - description 70
 - determined by SQLLEN variable 1666
 - in assignments 105
 - in comparisons 114
 - results of arithmetic operations 182
 - values for data types 70
- PRECOMPILE_OPTS column of SYSROUTINES_OPTS catalog table 1859
- precompiler
 - checks SQL statements 1219
 - DECLARE TABLE statement 1217
 - DECLARE VARIABLE statement 1221
 - escape character 50
 - options
 - COBOL decimal point 251
 - CONNECT 248
 - date 255
 - NOFOR 256
 - STDSQL 255
 - string delimiter 253
 - time 255
 - SET CURRENT APPLICATION ENCODING SCHEME statement 1477
 - using INCLUDE statements 1365
- PRECOMPTS column of SYSDBRM catalog table 1754
- predicate
 - basic 224
 - BETWEEN 229
 - description 222
 - DISTINCT 230
 - EXISTS 232
 - IN 234
 - LIKE 237
 - NULL 243
 - quantified 226
 - XML EXISTS 244
- prefix operator 182
- PRELINK_OPTS column
 - SYSROUTINES_OPTS catalog table 1859
- PREPARE statement
 - description 1405
 - example 1420
- prepared SQL statement
 - dynamically prepared by PREPARE 1405
 - executing 1275
 - identifying by DECLARE 1216
 - obtaining information
 - with DESCRIBE 1243
 - with DESCRIBE INPUT 1240

- prepared SQL statement (*continued*)
 - SQLDA provides information 1656
 - statements allowed 1601
- PREVIOUS VALUE expression
 - definition 217
- PRIMARY KEY clause
 - ALTER TABLE statement
 - description 814
 - CREATE TABLE statement 1091, 1100
- primary keys 6
- PRIOR clause
 - FETCH statement 1294
- PRIOR ROWSET clause
 - FETCH statement 1299
- PRIQTY clause
 - ALTER INDEX statement 730
 - ALTER TABLESPACE statement 849
 - CREATE INDEX statement 1001
 - CREATE TABLESPACE statement 1132
- privilege
 - granting 1333
 - revoking 1432
 - types 1333
- PRIVILEGE column of SYSCOLAUTH catalog table 1714
- privileges
 - object ownership 60
- procedure
 - creating with CREATE PROCEDURE statement 1015
- PROCEDURE clause
 - COMMENT statement 893
 - DROP statement 1262
- procedure, stored
 - naming convention 53
- procedures 29
 - creating
 - with CREATE PROCEDURE (SQL - native) statement 1046
 - inheriting special registers 151
- process
 - description 25
- product-sensitive programming information, described 2043
- program
 - naming convention 54
- PROGRAM clause 1271
- PROGRAM TYPE clause
 - ALTER FUNCTION statement 716
 - ALTER PROCEDURE (external) statement 749
 - ALTER PROCEDURE (SQL - external) statement 755
 - CREATE FUNCTION statement 934, 955
 - CREATE PROCEDURE (external) statement 1030
 - CREATE PROCEDURE (SQL - external) statement 1043
- PROGRAM_TYPE column of SYSROUTINES catalog table 1848
- programming interface information, described 2041
- promotion of data types 95
- PROPERTIES column
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSROBJECTPROPERTY table 1927
 - SYSIBM.XSROBJECTS table 1922
- PSEUDO_DEL_ENTRIES column
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
- PSID column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- PSID column of SYSTABLESPACE catalog table 1896
- PSPI symbols 2043

- PUBLIC clause
 - CREATE TRUSTED CONTEXT statement 1174
 - GRANT statement 1334
 - REVOKE statement 1433

Q

- QMF (Query Management Facility)
 - database for each user 12
- qualification of column names 154
- QUALIFIER
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPLAN catalog table 1832
 - column of SYSRESAUTH catalog table 1843
 - unqualified object names 58
- QUALIFIER clause
 - ALTER PROCEDURE (SQL - native) statement 765
 - CREATE PROCEDURE (SQL - native) statement 1052
- quantified predicate 226
- QUANTILENO column
 - SYSOLDIST catalog table 1715
 - SYSOLDIST_HIST catalog table 1719
 - SYSOLDISTSTATS catalog table 1717
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
- QUANTIZE function 472
- QUARTER function 474
- query 629
- Query Management Facility (QMF) 12
- QUERYNO clause
 - DELETE statement 1232
 - INSERT statement 1373
 - SELECT INTO statement 1473
 - select-statement 678
 - UPDATE statement 1530
- QUERYNO column
 - SYPACKSTMT catalog table 1823
 - SYSSTMT catalog table 1868
- question mark (?) 1276
- quotation mark 50, 253
- QUOTE
 - column of SYSDBRM catalog table 1754
 - column of SYSPACKAGE catalog table 1810
 - option of precompiler 253
- QUOTESQL option of precompiler 253

R

- RACF (Resource Access Control Facility)
 - security for remote execution 69
- RADIANS function 475
- RAISE_ERROR function 476
- RAND function 477
- range-partitioned table space 1143
- RANK expression 212
- RBA column of SYSCHECKS catalog table 1711
- RBA1 column of SYSTABLES catalog table 1890
- RBA2 column of SYSTABLES catalog table 1890
- READ SQL clause
 - ALTER PROCEDURE (external) statement 747
- READ SQL DATA clause
 - ALTER FUNCTION statement 710
- read-only
 - FOR FETCH ONLY clause 674
 - FOR READ ONLY clause 674

- read-only (*continued*)
 - view 1189
- READS SQL DATA clause
 - ALTER FUNCTION statement 723
 - ALTER PROCEDURE (SQL - external) statement 753
 - ALTER PROCEDURE (SQL - native) statement 764
 - CREATE FUNCTION statement 928, 950, 979
 - CREATE PROCEDURE (external) statement 1026
 - CREATE PROCEDURE (SQL - external) statement 1041
 - CREATE PROCEDURE (SQL - native) statement 1051
- REAL data type
 - CREATE TABLE statement 1088
 - description 71
- REAL function 478
- REBUILDLASTTIME column
 - SYSINDEXSPACESTATS catalog table 1777
- RECLENGTH column of SYSTABLES catalog table 1890
- RECOVER privilege
 - GRANT statement 1353
 - REVOKE statement 1456
- RECOVERAUTH column of SYSUSERAUTH catalog table 1911
- RECOVERDB privilege
 - GRANT statement 1338
 - REVOKE statement 1440
- RECOVERDBAUTH column of SYSDBAUTH catalog table 1751
- recovery
 - COMMIT statement 896
 - description
 - restoring data consistency 25
 - unit of 26
- REFCOLS column of SYSTABAUTH catalog table 1878
- REFERENCES clause
 - ALTER TABLE statement 815
- REFERENCES privilege
 - GRANT statement 1356
 - REVOKE statement 1459
- references to labels 1544
- REFERENCESAUTH column of SYSTABAUTH catalog table 1878
- REFERENCING clause of TRIGGER statement 1154
- referencing SQL parameters 1542
- referencing SQL variables 1542
- referential constraint
 - ALTER TABLE statement 815
 - CREATE TABLE statement 1100
- referential constraints 20
- referential constraints and referential integrity
 - defined 20
- referential integrity 20
- REFRESH column
 - SYSVIEWS catalog table 1915
- REFRESH TABLE statement
 - description 1422
- REFRESH_TIME column
 - SYSVIEWS catalog table 1915
- REFTBCREATOR column of SYSRELS catalog table 1841
- REFTBNAME column of SYSRELS catalog table 1841
- REGENERATE clause
 - ALTER INDEX statement 728
 - ALTER VIEW statement 867
- REGENERATE VERSION clause
 - ALTER PROCEDURE (SQL - native) statement 762
- RELATIVE clause
 - FETCH statement 1297
- RELBOUNDED column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- RELCREATED column
 - SYSAXXRELS catalog table 1709
 - SYSCHECKS catalog table 1711, 1712
 - SYSCOLUMNS catalog table 1723
 - SYSCONTEXT catalog table 1737
 - SYSCOPY catalog table 1740
 - SYSDATABASE catalog table 1747
 - SYSDATATYPES catalog table 1749
 - SYSDBRM catalog table 1754
 - SYSENVIRONMENT catalog table 1759
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSROBJECTHIERARCHIES table 1926
 - SYSIBM.XSROBJECTS table 1922
 - SYSINDEXES catalog table 1764
 - SYSKEYTARGETS catalog table 1793
 - SYSRELS catalog table 1841
 - SYSROLES catalog table 1845
 - SYSROUTINES catalog table 1848
 - SYSSEQUENCES catalog table 1865
 - SYSTOGROUP catalog table 1872
 - SYSNONONYMS catalog table 1877
 - SYSTABCONST catalog table 1881
 - SYSTABLEPART catalog table 1882
 - SYSTABLES catalog table 1890
 - SYSTABLESPACE catalog table 1896
 - SYSSTRICTURES catalog table 1909
 - SYSVIEWS catalog table 1915
 - SYSVOLUMES catalog table 1917
 - SYSXMLRELS catalog table 1918
- RELEASE
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPLAN catalog table 1832
- RELEASE (connection) statement
 - description 1424
 - example 1425
- RELEASE AT clause
 - ALTER PROCEDURE (SQL - native) statement 771
 - CREATE PROCEDURE (SQL - native) statement 1059
- release dependency indicators 1677
- release level identification, current server 901
- RELEASE SAVEPOINT statement
 - description 1427
 - example 1427
- release-pending connection state 33
- RELNAME column
 - SYSFOREIGNKEYS catalog table 1763
 - SYSRELS catalog table 1841
- RELOBID1 column of SYSRELS catalog table 1841
- RELOBID2 column of SYSRELS catalog table 1841
- REMARKS column
 - SYSCOLUMNS catalog table 1723
 - SYSCONTEXT catalog table 1737
 - SYSDATATYPES catalog table 1749
 - SYSIBM.XSROBJECTS table 1922
 - SYSINDEXES catalog table 1764
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
 - SYSROLES catalog table 1845
 - SYSROUTINES catalog table 1848
 - SYSSEQUENCES catalog table 1865
 - SYSTABLES catalog table 1890, 1999
 - SYSSTRICTURES catalog table 1909
- REMOTE column of SYSPACKAGE catalog table 1810

- remote database server
 - definition of 31
- Remote Recovery Data Facility (RRDF) 1112
- remote unit of work
 - connection management 36
 - definition of 35
- REMOVE VOLUMES clause of ALTER STOGROUP statement 787
- RENAME COLUMN clause
 - ALTER TABLE statement 813
- RENAME statement
 - description 1428
 - example 1431
- REOPT clause
 - ALTER PROCEDURE (SQL - native) statement 772
 - CREATE PROCEDURE (SQL - native) statement 1060
- REOPTVAR column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- REORDMASSDELETE column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- REORG privilege
 - GRANT statement 1338
 - REVOKE statement 1440
- REORG_LR_TS column
 - SYSTABLEPART catalog table 1882
- REORGAPPENDINSERT column
 - SYSINDEXSPACESTATS catalog table 1777
- REORGAUTH column of SYSDBAUTH catalog table 1751
- REORGDELETES column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- REORGDISORGLOB column
 - SYSTABLESPACESTATS catalog table 1901
- REORGFARINDREF column
 - SYSTABLESPACESTATS catalog table 1901
- REORGINSERTS column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- REORGLASTTIME column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- REORGLEAFFAR column
 - SYSINDEXSPACESTATS catalog table 1777
- REORGLEAFNEAR column
 - SYSINDEXSPACESTATS catalog table 1777
- REORGNEARINDREF column
 - SYSTABLESPACESTATS catalog table 1901
- REORGNUMLEVELS column
 - SYSINDEXSPACESTATS catalog table 1777
- REORGPSEUDODELETES column
 - SYSINDEXSPACESTATS catalog table 1777
- REORGUNCLUSTINS column
 - SYSTABLESPACESTATS catalog table
 - description 1901
- REORGUPDATES column
 - SYSTABLESPACESTATS catalog table
 - description 1901
- REPAIR privilege
 - GRANT statement 1338
 - REVOKE statement 1440
- REPAIRAUTH column of SYSDBAUTH catalog table 1751
- REPEAT function 480
- REPEAT statement
 - example 1575
 - SQL procedure 1575, 1635
- REPLACE function 482
- REPLACE USE FOR clause
 - ALTER TRUSTED CONTEXT statement 862
- REPLACE VERSION clause
 - ALTER PROCEDURE (SQL - native) statement 762
- reserved keywords 1596
- reserved schema names 1595
- RESET
 - clause of CONNECT statement 900
- RESET clause
 - ALTER TABLE statement 826
- RESIGNAL statement
 - example 1577, 1636
 - SQL procedure 1577, 1636
- resource limit facility (governor)
 - database 19
- RESTART WITH
 - clause of ALTER SEQUENCE statement 782
- RESTART WITH clause
 - ALTER TABLE statement 813
- RESTARTWITH column
 - SYSSEQUENCES catalog table 1865
- RESTRICT
 - delete rule
 - ALTER TABLE statement 816
 - CREATE TABLE statement 1101
- RESTRICT clause of REVOKE statement 1434, 1461
- RESTRICT WHEN DELETE TRIGGERS clause
 - TRUNCATE statement 1518
- result column
 - data type 636
 - names 635
- Result data types with numeric operands 119
- RESULT SET clause
 - CREATE PROCEDURE (external) statement 1033
 - CREATE PROCEDURE (SQL - external) statement 1045
 - CREATE PROCEDURE (SQL - native) statement 1063
- result set locator
 - description 167
- RESULT SETS clause
 - CREATE PROCEDURE (external) statement 1033
 - CREATE PROCEDURE (SQL - external) statement 1045
 - CREATE PROCEDURE (SQL - native) statement 1063
- RESULT_COLS column of SYSROUTINES catalog table 1848
- RESULT_SETS column
 - SYSROUTINES catalog table 1848
- RETURN statement
 - example 1639
 - examples 1580
 - SQL procedure 1580, 1639
- RETURN STATUS clause 1330
- RETURN_TYPE column of SYSROUTINES catalog table 1848
- RETURN-statement clause
 - CREATE FUNCTION statement 979
- RETURNS clause
 - CREATE FUNCTION statement 976
- RETURNS clause of CREATE FUNCTION statement 923, 963
- RETURNS NULL ON NULL INPUT clause
 - ALTER FUNCTION statement 710
 - CREATE FUNCTION statement 928, 949
- RETURNS TABLE clause of CREATE FUNCTION statement 946
- REUSE STORAGE clause
 - TRUNCATE statement 1518
- REVOKE statement
 - alternative syntax 1346, 1448
 - cascading effect 1434

- REVOKE statement (*continued*)
 - collection privileges 1437
 - database privileges 1439
 - description 1432
 - function privileges 1442
 - JAR file privileges 1461
 - package privileges 1447
 - plan privileges 1449
 - procedure privileges 1442
 - schema privileges 1451
 - sequence privileges 1453
 - system privileges 1455
 - table privileges 1458
 - type privileges 1461
 - use privileges 1463
 - view privileges 1458
- REXX
 - SQLCA 1654
 - SQLDA 1675
- REXX SQLCA 1654
- REXX SQLDA 1675
- RID function 485
- RIGHT function 486
- RIGHT OUTER JOIN
 - example 657
 - FROM clause of subselect 645
- role
 - defining 1065
 - naming convention 54
- ROLE AS OBJECT OWNER clause
 - ALTER TRUSTED CONTEXT statement 858
 - CREATE TRUSTED CONTEXT statement 1169
- ROLE clause
 - COMMENT statement 893
 - CREATE TRUSTED CONTEXT statement 1172, 1173
 - DROP statement 1263
 - GRANT statement 1334
 - REVOKE statement 1433
- ROLE column
 - SYSCONTEXTAUTHIDS catalog table 1739
- ROLENAME column
 - SYSOBJROLEDEP catalog table 1809
- rollback
 - description 25
- ROLLBACK statement
 - description 1465
 - example 1467
- ROTATE PARTITION FIRST TO LAST clause
 - ALTER TABLE statement 824
- ROUND function 488
- ROUND_TIMESTAMP function 490
- ROUNDING clause
 - ALTER PROCEDURE (SQL - native) statement 773
 - CREATE PROCEDURE (SQL - native) statement 1060
- ROUNDING column
 - SYSENVIRONMENT catalog table 1759
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- rounding mode
 - DECFLOAT values 251
- routine
 - description 28
- routine versions
 - naming convention 54
- ROUTINEID column
 - SYSPARMS catalog table 1826
 - SYSROUTINES catalog table 1848
- ROUTINENAME column
 - SYSROUTINES_OPTS catalog table 1859
 - SYSROUTINES_SRC catalog table 1861
- routines
 - inheriting special registers 151
- ROUTINETYPE column
 - SYSPARMS catalog table 1826
 - SYSROUTINEAUTH catalog table 1846
 - SYSROUTINES catalog table 1848
- row
 - deleting 1224
 - inserting 1367, 1388
 - selecting single row 1471
 - updating 1521
- ROW CHANGE TIMESTAMP
 - expression 216
- row change timestamp column
 - CREATE TABLE statement 1095
- row change timestamp columns
 - ALTER TABLE statement 806
- ROW CHANGE TIMESTAMP expression
 - definition 216
- ROW CHANGE TOKEN
 - expression 216
- ROW CHANGE TOKEN expression
 - definition 216
- row ID
 - assignment of values 113
 - comparison of values 117
 - data type 92, 1090
- Row ID
 - operands 122
- ROW_NUMBER expression 212
- row-positioned clause
 - FETCH statement 1293
- row-value expression 222
- ROWID
 - data type
 - CREATE TABLE statement 1090
 - description 92
 - function 493
- ROWID column
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSROBJECTS table 1922
- ROWNUMBER expression 212
- ROWSET STARTING AT clause
 - FETCH statement 1301
- rowset-positioned clause
 - FETCH statement 1298
- rowset-positioning clause
 - DECLARE CURSOR statement 1196
 - PREPARE statement 1410
- ROWTYPE column of SYSPARMS catalog table 1826
- RPAD function 494
- RRDF (Remote Recovery Data Facility)
 - altering a table for 828
 - creating a table for 1112
- RTRIM function 496
- run behavior for dynamic SQL statements 64
- RUN OPTIONS clause
 - ALTER FUNCTION statement 716
 - ALTER PROCEDURE (external) statement 749
 - ALTER PROCEDURE (SQL - external) statement 755
 - CREATE FUNCTION statement 935, 955
 - CREATE PROCEDURE (external) statement 1031
 - CREATE PROCEDURE (SQL - external) statement 1043

RUNOPTS column
SYSROUTINES catalog table 1848

S

sample user-defined functions 2007
savepoint
 naming convention 54
 releasing 1427
 setting 1468
SAVEPOINT statement
 description 1468
 example 1469
SBCS data
 description 74
SBCS_CCSID column
 SYSDATABASE catalog table 1747
 SYSTABLESPACE catalog table 1896
scalar 392
scalar-fullselect 191
SCALE column
 SYSCOLUMNS catalog table 1723
 SYSDATATYPES catalog table 1749
 SYSFIELDS catalog table 1761
 SYSKEYTARGETS catalog table 1793
 SYSKEYTARGETS_HIST catalog table 1798
 SYSPARMS catalog table 1826
scale of numbers
 assignments 106
 comparisons 114
 description 71
 results of arithmetic operations 184
schema
 naming convention 54
 privileges 1349, 1451
SCHEMA
 column of SYSSEQUENCEAUTH catalog table 1863
SCHEMA column
 SYSDATATYPES catalog table 1749
 SYSPARMS catalog table 1826
 SYSROUTINEAUTH catalog table 1846
 SYSROUTINES catalog table 1848
 SYSROUTINES_OPTS catalog table 1859
 SYSROUTINES_SRC catalog table 1861
 SYSSEQUENCES catalog table 1865
 SYSTRIGGERS catalog table 1909
schema names, reserved 1595
schema processor 1091
SCHEMALOCATION column
 SYSIBM.XSROBJECTCOMPONENTS table 1924, 1926
 SYSIBM.XSROBJECTS table 1922
SCHEMANAME column
 SYSSCHEMAAUTH catalog table 1862
schemas
 definition 10
SCORE function 498
SCRATCHPAD clause
 ALTER FUNCTION statement 712
 CREATE FUNCTION statement 930, 951
SCRATCHPAD column of SYSROUTINES catalog table 1848
SCRATCHPAD_LENGTH column of SYSROUTINES catalog table 1848
SCREATOR column of SYSTABAUTH catalog table 1878
SCROLL clause
 DECLARE CURSOR statement 1193
SCT02 table space
 description 17

search condition
 DELETE statement 1230
 description 247
 HAVING clause 651
 order of evaluation 247
 UPDATE statement 1528
 WHERE clause 648
SECLABEL session variable 168
SECOND function 501
SECQTY clause
 ALTER INDEX statement 731
 ALTER TABLESPACE statement 850
 CREATE INDEX statement 1001
 CREATE TABLESPACE statement 1133
SECQTYI column
 SYSINDEXPART catalog table 1771
 SYSINDEXPART_HIST catalog table 1775
 SYSTABLEPART catalog table 1882
 SYSTABLEPART_HIST catalog table 1887
SECTNO column
 SYSPACKSTMT catalog table 1823
 SYSSTMT catalog table 1868
SECTNOI column
 SYSPACKSTMT catalog table 1823
 SYSSTMT catalog table 1868
SECURE column
 LOCATIONS catalog table 1701
SECURITY clause
 ALTER FUNCTION statement 716
 ALTER PROCEDURE (external) statement 749
 ALTER PROCEDURE (SQL - external) statement 755
 CREATE FUNCTION statement 934, 955
 CREATE PROCEDURE (external) statement 1030
 CREATE PROCEDURE (SQL - external) statement 1043
security label
 naming convention 54
SECURITY LABEL clause
 CREATE TRUSTED CONTEXT statement 1173
SECURITY_IN column of LUNAMES catalog table 1705
SECURITY_LABEL column
 SYSTABLES catalog table 1890
SECURITY_OUT column
 IPNAMES catalog table 1698
 LUNAMES catalog table 1705
SECURITYLABEL column
 SYSCONTEXTAUTHIDS catalog table 1739
segmented table spaces
 described 14
SEGSIZE
 clause of CREATE TABLESPACE statement 1143
 column of SYSTABLESPACE catalog table 1896
SELECT
 clause as syntax component 633
SELECT INTO statement
 description 1471
 example 1474
SELECT privilege
 GRANT statement 1356
 REVOKE statement 1459
SELECT statement
 common table expression 670
 description 669, 1470
 dynamic invocation 692
 example 680
 SYSIBM.SYSCOLUMNS 2000
 SYSIBM.SYSINDEXES 2001
 SYSIBM.SYSTABAUTH 2002

- SELECT statement (*continued*)
 - example (*continued*)
 - SYSIBM.SYSTABLEPART 1999
 - SYSIBM.SYSTABLES 1999, 2006
 - fullselect 662
 - list
 - application 633
 - description 633
 - maximum number of elements 1588
 - notation 633
 - static invocation 691
 - subselect 632
- SELECTAUTH column of SYSTABAUTH catalog table 1878
- selecting
 - single row 1471
- self-referencing tables 20
- SENSITIVE clause
 - DECLARE CURSOR statement 1193
 - FETCH statement 1292
- SEQNO column
 - SYSPACKLIST catalog table 1822
 - SYSPACKSTMT catalog table 1823
 - SYSROUTINES_SRC catalog table 1861
 - SYSSTMT catalog table 1868
 - SYSTRIGGERS catalog table 1909
 - SYSVIEWS catalog table 1915
- SEQTYPE column of SYSSEQUENCES catalog table 1865
- sequence
 - ALTER SEQUENCE statement 781
 - catalog information 2006
 - CREATE SEQUENCE statement 1066
 - description 30
 - dropping 1263
 - granting privileges 1351
 - name, unqualified 58
 - naming convention 54
 - reference 217
 - revoking privileges 1453
 - unqualified name 58
- SEQUENCE
 - clause of ALTER SEQUENCE statement 781
- SEQUENCE clause
 - COMMENT statement 893
 - CREATE SEQUENCE statement 1068
 - DROP statement 1263
 - GRANT statement 1351
 - REVOKE statement 1453
- SEQUENCEID column of SYSSEQUENCES catalog table 1865
- SERVAUTH clause
 - ALTER TRUSTED CONTEXT statement 860
 - CREATE TRUSTED CONTEXT statement 1172
- server
 - naming convention 54
 - remote 31
- session variable
 - built-in 168
 - returning values 374
- session variable, built-in 168
- SESSION_USER 150
- SESSION_USER clause
 - SET PATH statement 1507
- SESSION_USER special register 150
- SET CACHE clause
 - ALTER TABLE statement 813
- SET clause
 - DELETE statement 1229
- SET clause of UPDATE statement 1526
- SET CONNECTION statement
 - description 1475
 - example 1476
- SET CURRENT APPLICATION ENCODING SCHEME
 - statement
 - description 1477
 - example 1477
- SET CURRENT DEBUG MODE statement
 - description 1478
 - example 1479
- SET CURRENT DECFLOAT ROUNDING MODE statement
 - description 1480
 - example 1481
- SET CURRENT DEGREE statement
 - description 1482
 - example 1482
- SET CURRENT LOCALE LC_CTYPE statement
 - description 1483
 - example 1484
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement
 - description 1485
 - example 1486
- SET CURRENT OPTIMIZATION HINT statement
 - description 1487
 - example 1487
- SET CURRENT PACKAGE PATH
 - statement
 - description 1488
 - example 1490
- SET CURRENT PACKAGESET statement
 - description 1492
 - example 1493
- SET CURRENT PRECISION statement
 - description 1494
 - example 1494
- SET CURRENT REFRESH AGE statement
 - description 1495
- SET CURRENT ROUTINE VERSION statement
 - description 1497
 - example 1498
- SET CURRENT RULES statement
 - description 1499
 - example 1499
- SET CURRENT SQLID statement
 - description 1500
 - example 1501
- SET CYCLE clause
 - ALTER TABLE statement 813
- SET ENCRYPTION PASSWORD statement 1502
- SET GENERATED clause
 - ALTER TABLE statement 812
- SET host-variable assignment statement
 - description 1504
 - example 1506
- SET INCREMENT BY clause
 - ALTER TABLE statement 813
- SET MAXVALUE clause
 - ALTER TABLE statement 813
- SET MINVALUE clause
 - ALTER TABLE statement 813
- SET NO CYCLE clause
 - ALTER TABLE statement 813
- SET NO MAXVALUE clause
 - ALTER TABLE statement 813
- SET NO MINVALUE clause
 - ALTER TABLE statement 813

- SET NO ORDER clause
 - ALTER TABLE statement 813
- SET NULL delete rule
 - ALTER TABLE statement 816
 - CREATE TABLE statement 1101
- set operators 662
- SET ORDER clause
 - ALTER TABLE statement 813
- SET PATH statement
 - description 1507
 - example 1509
- SET QUERYNO clause of EXPLAIN statement 1285
- SET SCHEMA statement
 - description 1510
- SET transition-variable assignment statement
 - description 1513
 - example 1515
- SGCREATOR column of SYSVOLUMES catalog table 1917
- SGNAME column of SYSVOLUMES catalog table 1917
- SHARE
 - option of LOCK TABLE statement 1386
- shift-in character
 - convention xxii
 - LABEL statement 1385
 - not truncated by assignments 110
- shift-out character
 - convention xxii
 - LABEL statement 1385
- short string column 74, 83
- shortcut keys
 - keyboard xviii
- SHRLEVEL
 - column of SYSCOPY catalog table 1740
- SIGN function 502
- sign-on exit routine
 - CURRENT SQLID special register 64, 148
- SIGNAL statement
 - description 1516
 - example 1582, 1641
 - SQL procedure 1641
 - SQL routine 1582
- SIGNATURE column
 - SYSVIEWS catalog table 1915
- SIMPLE CALL clause
 - CREATE PROCEDURE (external) statement 1033
- SIMPLE CALL WITH NULLS clause
 - CREATE PROCEDURE (external) statement 1033
- SIN function 503
- single logging 17
- single precision floating-point number 71
- single-row-fetch clause
 - FETCH statement 1302
- SINH function 504
- SKCT (skeleton cursor table)
 - description 17
- skeleton cursor table (SKCT) 17
- skeleton package table (SKPT) 17
- SKIP LOCKED DATA clause
 - DELETE statement 1232
 - PREPARE statement 1412
 - SELECT INTO statement 1473
 - select-statement 679
 - UPDATE statement 1529
- SKPT (skeleton package table)
 - description 17
- SMALLINT function 505
- SOAPHTTPC and SOAPHTTPV functions 508
- SOAPHTTPNC and SOAPHTTPNV functions 509
- softcopy publications 2033
- SOME quantified predicate 226
- sort-key
 - ORDER BY clause of subselect 653
- SOUNDEX function 507
- SOURCE clause of CREATE FUNCTION statement 964
- SOURCEDSN column
 - SYSROUTINES_OPTS catalog table 1859
- SOURCESCHEMA column
 - SYSDATATYPES catalog table 1749
 - SYSROUTINES catalog table 1848
- SOURCESPECIFIC column of SYSROUTINES catalog table 1848
- SOURCETYPE column of SYSDATATYPES catalog table 1749
- SOURCETYPEID column
 - DATATYPES catalog table 1749
 - SYSCOLUMNS catalog table 1723
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSPARMS catalog table 1826
 - SYSSEQUENCES catalog table 1865
- space character 48
- SPACE column
 - SYSINDEXES catalog table 1764
 - SYSINDEXPART catalog table 1771
 - SYSSTOGROUP catalog table 1872
 - SYSTABLEPART catalog table 1882
 - SYSTABLESPACE catalog table 1896
- SPACE column of SYSINDEXSPACESTATS catalog table 1777
- SPACE column of SYSTABLESPACESTATS catalog table 1901
- SPACE function 511
- SPACEF
 - column of SYSTABLESPACE catalog table 1896
- SPACEF column
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSSTOGROUP catalog table 1872
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
- special character 47
- special register
 - behavior in user-defined functions and stored procedures 151
 - CURRENT APPLICATION ENCODING SCHEME 134
 - CURRENT CLIENT_ACCTNG 135
 - CURRENT CLIENT_APPLNAME 136
 - CURRENT CLIENT_USERID 136
 - CURRENT CLIENT_WRKSTNNAME 137
 - CURRENT DATE 137
 - CURRENT DEBUG MODE 137
 - CURRENT DECFLOAT ROUNDING MODE 138
 - CURRENT DEGREE 139
 - CURRENT LOCALE LC_CTYPE 140
 - CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION 140
 - CURRENT MEMBER 141
 - CURRENT OPTIMIZATION HINT 141
 - CURRENT PACKAGE PATH 141
 - CURRENT PACKAGESET 142
 - CURRENT PATH 142
 - CURRENT PRECISION 143
 - CURRENT REFRESH AGE 144

- special register (*continued*)
 - CURRENT ROUTINE VERSION 145
 - CURRENT RULES 145
 - CURRENT SCHEMA 147
 - CURRENT SERVER 147
 - CURRENT SQLID 148
 - CURRENT TIME 148
 - CURRENT TIMESTAMP 149
 - CURRENT TIMEZONE 149
 - CURRENT_DATE 137
 - CURRENT_TIME 148
 - CURRENT_TIMESTAMP 149
 - description 131
 - ENCRYPTION PASSWORD 150
 - SESSION_USER 150
 - USER 150
 - values in trigger 1162
- SPECIAL REGS column
 - SYSROUTINES catalog table 1848
- specific
 - naming convention 55
- SPECIFIC clause
 - CREATE FUNCTION statement 924, 946, 964, 977
- SPECIFIC FUNCTION clause of ALTER FUNCTION statement 706, 722
- specific name
 - unqualified name 58
- specifications
 - XMLCAST 210
- SPECIFICNAME column
 - SYSPARMS catalog table 1826
 - SYSROUTINEAUTH catalog table 1846
 - SYSROUTINES catalog table 1848
- SPLIT_ROWS column
 - SYSTABLES catalog table 1890
- SPT01 table space 17
- SQL (Structured Query Language) (*continued*)
 - assignment operation 102
 - Call Level Interface (CLI) 3
 - character 47
 - CLI (Call Level Interface) 3
 - comparison operation 102
 - constants 126
 - data types
 - binary strings 85
 - casting 96
 - character strings 73
 - datetime 87
 - description 69
 - graphic strings 83
 - LOBs (large objects) 85
 - numbers 70
 - promotion 95
 - results of an operation 119
 - row ID 92
 - XML values 93
 - deferred embedded 3
 - delimited identifier 50
 - dynamic
 - description 3
 - statements allowed 1601
 - identifier 49
 - interactive 3
 - JDBC 4
 - keywords, reserved 1596
 - limits 1588
 - naming conventions 51
 - SQL (Structured Query Language) (*continued*)
 - null value 70
 - ODBC (Open Database Connectivity) 3
 - Open Database Connectivity (ODBC) 3
 - ordinary identifier 47
 - rules 145
 - schema names, reserved 1595
 - SQLJ 4
 - standard 255
 - standards xxiii
 - static
 - description 2
 - token 48
 - value 69
 - variable names 51
 - SQL comments 695
 - SQL condition
 - naming convention 55
 - SQL condition names 1543
 - SQL control statement
 - assignment statement 1548, 1613
 - CALL statement 1550, 1615
 - CASE statement 1552, 1617
 - compound statement 1554, 1620
 - CONTINUE handler 1554, 1620
 - EXIT handler 1554, 1620
 - FOR statement 1563
 - GET DIAGNOSTICS statement 1565, 1626
 - GOTO statement 1566, 1627
 - handler 1554, 1620
 - handling errors 1554, 1620
 - IF statement 1568, 1629
 - ITERATE statement 1569, 1631
 - LEAVE statement 1571, 1632
 - LOOP statement 1573, 1633
 - order of statements 1554, 1620
 - REPEAT statement 1575, 1635
 - RESIGNAL statement 1577, 1636
 - RETURN statement 1580, 1639
 - SIGNAL statement 1582, 1641
 - WHILE statement 1586, 1645
 - SQL cursor names 1543
 - SQL label
 - naming convention 55
 - SQL parameter
 - naming convention 55
 - SQL parameters 1542
 - SQL path 57, 175
 - SQL PATH clause
 - ALTER PROCEDURE (SQL - native) statement 771
 - CREATE PROCEDURE (SQL - native) statement 1058
 - SQL procedure
 - new line control character 48
 - statements allowed 1608
 - SQL scalar statements
 - ALTER FUNCTION 719
 - SQL statements
 - ALLOCATE CURSOR 696
 - ALTER DATABASE 698
 - ALTER FUNCTION 701
 - ALTER INDEX 725
 - ALTER PROCEDURE (external) 741
 - ALTER PROCEDURE (SQL - external) 752
 - ALTER PROCEDURE (SQL - native) statement 758
 - ALTER SEQUENCE 781
 - ALTER STOGROUP 786
 - ALTER TABLE 789

SQL statements (*continued*)

- ALTER TABLESPACE 841
- ALTER TRUSTED CONTEXT 855
- ALTER VIEW 867
- ASSOCIATE LOCATORS 868
- BEGIN DECLARE SECTION 872
- CALL 874
- catalog table restrictions 1689
- CLOSE 885
- COMMENT 887
- COMMIT 896
- CONNECT 899
- CONTINUE 1539
- CREATE ALIAS 905
- CREATE AUXILIARY TABLE 908
- CREATE DATABASE 912
- CREATE FUNCTION 915
- CREATE FUNCTION (external scalar) 916
- CREATE FUNCTION (external table) 940
- CREATE FUNCTION (sourced) 958
- CREATE FUNCTION (SQL scalar) 972
- CREATE GLOBAL TEMPORARY TABLE 982
- CREATE INDEX 988
- CREATE PROCEDURE 1015
- CREATE PROCEDURE (external) 1016
- CREATE PROCEDURE (SQL - external) 1035
- CREATE PROCEDURE (SQL - native) 1046
- CREATE ROLE 1065
- CREATE SEQUENCE 1066
- CREATE STOGROUP 1074
- CREATE SYNONYM 1077
- CREATE TABLE 1079
- CREATE TABLESPACE 1128
- CREATE TRIGGER 1151
- CREATE TRUSTED CONTEXT 1167
- CREATE TYPE 1177
- CREATE VIEW 1184
- DECLARE CURSOR
 - description 1191
 - example 1199
- DECLARE GLOBAL TEMPORARY TABLE 1202
- DECLARE STATEMENT 1216
- DECLARE TABLE 1217
- DECLARE VARIABLE 1221
- DELETE
 - description 1224
 - example 1235
- DESCRIBE 1237
- DESCRIBE CURSOR 1238
- DESCRIBE INPUT 1240
- DESCRIBE OUTPUT 1243
- DESCRIBE PROCEDURE 1250
- DESCRIBE TABLE 1253
- DROP 1256
- END DECLARE SECTION 1273
- EXCHANGE 1274
- EXECUTE 1275
- EXECUTE IMMEDIATE 1280
- EXPLAIN
 - description 1283
 - example 1288
- FETCH
 - description 1290
 - example 1314
- FOR 1285
- FREE LOCATOR 1316
- GET DIAGNOSTICS 1317

SQL statements (*continued*)

- GRANT 1333
- HOLD LOCATOR 1363
- INCLUDE
 - description 1365
 - example 1366
 - SQLCA 1652
 - SQLDA 1670
- INSERT
 - description 1367
 - example 1380
- invocation 688
- LABEL 1384
- LOCALE LC_CTYPE 1483
- LOCK TABLE 1386
- MERGE
 - description 1388
 - examples 1398
- OPEN
 - description 1400
 - example 1404
- PREPARE 1405
- REFRESH TABLE 1422
- RELEASE (connection) 1424
- RELEASE SAVEPOINT 1427
- remote execution
 - description 67
 - dynamic execution 68
 - static execution 68
- RENAME 1428
- REVOKE 1432
- ROLLBACK 1465
- SAVEPOINT 1468
- SELECT 1470
- SELECT INTO 1471
- SET CONNECTION 1475
- SET CURRENT APPLICATION ENCODING
 - SCHEME 1477
- SET CURRENT DEBUG MODE 1478
- SET CURRENT DECFLOAT ROUNDING MODE 1480
- SET CURRENT DEGREE 1482
- SET CURRENT MAINTAINED TABLE TYPES FOR
 - OPTIMIZATION 1485
- SET CURRENT OPTIMIZATION HINT 1487
- SET CURRENT PACKAGE PATH 1488
 - example 1490
- SET CURRENT PRECISION 1494
- SET CURRENT REFRESH AGE 1495
- SET CURRENT ROUTINE VERSION 1497
- SET CURRENT RULES 1499
- SET CURRENT SQLID 1500
- SET ENCRYPTION PASSWORD 1502
- SET host-variable assignment 1504
- SET PATH 1507
- SET SCHEMA 1510
- SET transition-variable assignment 1513
- SIGNAL 1516
- TRUNCATE 1517
- UPDATE
 - description 1521
 - example 1533
- VALUES 1536
- VALUES INTO 1537
- WHENEVER 1539
- SQL variable
 - naming convention 55
- SQL variables 1542

SQL_DATA_ACCESS column of SYSROUTINES catalog table 1848

SQL_STRING_DELIMITER column
SYSENVIRONMENT catalog table 1759

SQL_STRING_DELIMITER session variable 168

SQL-routine-body
ALTER PROCEDURE (SQL - native) statement 774
CREATE PROCEDURE (SQL - native) statement 1061

SQL/OLB 4

SQLCA
REXX 1654

SQLCA (SQL communication area)
contents 1646
entry changed by UPDATE 1530
INCLUDE statement 1365

SQLCABC field of SQLCA 1647

SQLCAID field of SQLCA 1647

SQLCODE
+100 694, 1372, 1400, 1471, 1539
description 694
field of SQLCA 1647

SQLD field of SQLDA 1246, 1658

SQLDA
header 1658
REXX 1675
unrecognized data types 1669

SQLDA (SQL descriptor area)
clause of INCLUDE statement 1365
contents 1656, 1658

SQLDABC field of SQLDA 1246, 1658

SQLDAID field of SQLDA 1245, 1658

SQLDATA field of SQLDA 1247, 1662

SQLDATAL field of SQLDA 1664

SQLDATALEN field of SQLDA 1664

SQLDATATYPE field of SQLDA 1247

SQLDATATYPE-NAME field of SQLDA 1664

SQLERRD(3) field of SQLCA 1233

SQLERRD(n) field of SQLCA 1647

SQLERRMC field of SQLCA 1647

SQLERRML field of SQLCA 1647

SQLERROR
clause of WHENEVER statement 1539
column of SYSPACKAGE catalog table 1810

SQLERRP field of SQLCA 1647

SQLIND field of SQLDA 1247, 1662

SQLJ 4

SQLLEN field of SQLDA 1247, 1662

SQLLONGL field of SQLDA 1664

SQLLONGLEN field of SQLDA 1247, 1664

SQLN field of SQLDA
description 1243, 1254, 1658

SQLNAME field of SQLDA 1247, 1662

SQLRULES
column of SYSPLAN catalog table 1832

SQLSTATE
'02000' 1372, 1400, 1471, 1539
description 694
field of SQLCA 1647
signaling 1516

SQLTNAME field of SQLDA 1664

SQLTYPE field of SQLDA
description 1662
values 1247, 1666

SQLVAR
base 1246
extended 1246

SQLVAR field of SQLDA 1246

SQLWARN6 field of SQLCA 194

SQLWARNING clause
WHENEVER statement 1539

SQLWARNn field of SQLCA 1647

SQRT function 512

SQTY column
SYSINDEXPART catalog table 1771
SYSTABLEPART catalog table 1882

SSID session variable 168

STANDARD CALL clause
CREATE PROCEDURE (external) statement 1033

STANDARD_SQL session variable 168

standard, SQL (ANSI/ISO)
description xxiii
SET CONNECTION statement 1475
SQL-style comments 48
STDSQL precompiler option 255

START column of SYSSEQUENCES catalog table 1865

START WITH clause
CREATE SEQUENCE statement 1068

START_RBA column of SYSCOPY catalog table 1740

STARTDB privilege
GRANT statement 1338
REVOKE statement 1440

STARTDBAUTH column of SYSDBAUTH catalog table 1751

state
application process 34, 36
SQL connection 33

statement
naming convention 55

STATEMENT clause of DECLARE STATEMENT
statement 1216

statement table
EXPLAIN statement 1283

STATIC clause
DECLARE CURSOR statement 1194

STATIC DISPATCH clause
ALTER FUNCTION statement 717, 724
CREATE FUNCTION statement 935, 956, 979

static SQL
description 2, 688
invocation of SELECT statement 691

STATS privilege
GRANT statement 1338
REVOKE statement 1440

STATS_FORMAT column
SYSCOLSTATS catalog table 1721
SYSCOLUMNS catalog table 1723
SYSCOLUMNS_HIST catalog table 1732
SYSKEYTARGETS catalog table 1793
SYSKEYTARGETS_HIST catalog table 1798
SYSKEYTARGETSTATS catalog table 1796

STATSAUTH column of SYSDBAUTH catalog table 1751

STATSDELETES column
SYSINDEXSPACESTATS catalog table 1777
SYSTABLESPACESTATS catalog table 1901

STATSINSERTS column
SYSINDEXSPACESTATS catalog table 1777
SYSTABLESPACESTATS catalog table 1901

STATSLASTTIME column
SYSINDEXSPACESTATS catalog table 1777
SYSTABLESPACESTATS catalog table 1901

STATSMASDELETE column
SYSTABLESPACESTATS catalog table 1901

STATSMASDELETES column
SYSINDEXSPACESTATS catalog table 1777

- STATSTIME column
 - SYSOLDIST catalog table 1715
 - SYSOLDIST_HIST catalog table 1719
 - SYSOLDISTSTATS catalog table 1717
 - SYSOLSTATS catalog table 1721
 - SYSOLUMNS catalog table 1723
 - SYSOLUMNS_HIST catalog table 1732
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSINDEXPART catalog table 1771
 - SYSINDEXPART_HIST catalog table 1775
 - SYSINDEXSTATS catalog table 1782
 - SYSINDEXSTATS_HIST catalog table 1784
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSKEYTARGETSTATS catalog table 1796
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
 - SYSLOBSTATS catalog table 1807
 - SYSLOBSTATS_HIST catalog table 1808
 - SYSSTOGROUP catalog table 1872
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
 - SYSTABLESPACE catalog table 1896
 - SYSTABSTATS catalog table 1907
 - SYSTABSTATS_HIST catalog table 1908
- STATSUPDATES column
 - SYSTABLESPACESTATS catalog table 1901
- STATUS column
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSROBJECTS table 1922
 - SYSPACKSTMT catalog table 1823
 - SYSSTMT catalog table 1868
 - SYSTABLES catalog table 1890
 - SYSTABLESPACE catalog table 1896
- STAY RESIDENT clause
 - ALTER FUNCTION statement 715
 - ALTER PROCEDURE (external) statement 748
 - ALTER PROCEDURE (SQL - external) statement 755
 - CREATE FUNCTION statement 934, 955
 - CREATE PROCEDURE (external) statement 1030
 - CREATE PROCEDURE (SQL - external) statement 1042
- STAYRESIDENT column
 - SYSROUTINES catalog table 1848
- STD SQL LANGUAGE field of panel DSNTIP4 255
- STDDEV
 - aggregate function 277
- STDDEV_POP function 277
- STDDEV_SAMP
 - aggregate function 277
- STDSQL option
 - precompiler 255
- STGROUP column of SYSDATABASE catalog table 1747
- STMT column of SYSPACKSTMT catalog table 1823
- STMTCACHE clause of EXPLAIN statement 1285
- STMTNO column
 - SYSPACKSTMT catalog table 1823
 - SYSSTMT catalog table 1868
- STMTNOI column
 - SYSPACKSTMT catalog table 1823
 - SYSSTMT catalog table 1868
- STNAME column of SYSTABAUTH catalog table 1878
- stogroup
 - naming convention 55
- STOGROUP
 - clause of ALTER INDEX statement 730, 733
 - clause of ALTER STOGROUP statement 786
 - clause of CREATE DATABASE statement 913
 - clause of CREATE INDEX statement 1000, 1003
 - clause of CREATE TABLESPACE statement 1132, 1134
- STOGROUP clause
 - ALTER DATABASE statement 698
 - ALTER TABLESPACE statement 848
 - DROP statement 1264
- STOGROUP privilege
 - GRANT statement 1361
 - REVOKE statement 1463
- STOP AFTER SYSTEM DEFAULT FAILURES clause
 - ALTER FUNCTION statement 716
 - ALTER PROCEDURE (external) statement 750
 - ALTER PROCEDURE (SQL - external) statement 756
 - CREATE FUNCTION statement 935, 956
 - CREATE PROCEDURE (external) statement 1030
 - CREATE PROCEDURE (SQL - external) statement 1044
- STOPALL privilege
 - GRANT statement 1353
 - REVOKE statement 1456
- STOPALLAUTH column of SYSUSERAUTH catalog table 1911
- STOPAUTH column of SYSDBAUTH catalog table 1751
- STOPDB privilege
 - GRANT statement 1338
 - REVOKE statement 1440
- storage
 - groups 11
- storage group, DB2
 - altering 786
 - creating 1074
 - dropping 1264
 - retrieving catalog information 1998
- storage structure 14
- STORCLAS clause
 - CREATE STOGROUP statement 787, 1075
- STORCLAS column
 - SYSSTOGROUP catalog table 1872
- stored procedure
 - altering
 - ALTER PROCEDURE (external) statement 741
 - with ALTER PROCEDURE (SQL - external) statement 752
 - with ALTER PROCEDURE (SQL - native) statement 758
- CALL statement 874
- creating
 - CREATE PROCEDURE (external) statement 1016
 - with CREATE PROCEDURE (SQL - external) statement 1035
 - with CREATE PROCEDURE (SQL - native) statement 1046
- CURRENT PACKAGESET special register 1493
- dropping 1262
- invoking 874
- name, unqualified 58
- naming convention 53
- privileges
 - granting 1340
 - revoking 1442
- statements allowed 1605
- unqualified name 58
- stored procedures
 - inheriting special registers 151

- STORES clause of CREATE AUXILIARY TABLE statement 909
- STORNAME column
 - SYSINDEXPART catalog table 1771
 - SYSTABLEPART catalog table 1882
- STORTYPE column
 - SYSINDEXPART catalog table 1771
 - SYSTABLEPART catalog table 1882
- STOSPACE privilege
 - GRANT statement 1353
 - REVOKE statement 1456
- STOSPACEAUTH column of SYSUSERAUTH catalog table 1911
- string
 - binary 85
 - CCSID 43
 - character 73
 - comparison 116
 - constant 128
 - conversion 38
 - datetime values 89
 - delimiter
 - COBOL 253
 - controlling representation 253
 - SQL 253
 - description 38
 - encoding scheme 43
 - fixed-length
 - description 74, 83
 - graphic 83
 - long column
 - description 84
 - limitations 634
 - numbers 72
 - short 74, 83
 - varying-length
 - description 74, 84
- string clause
 - CREATE PROCEDURE (external) statement 1024
- STRING column
 - SYSXMLSTRINGS catalog table 1919
- string delimiter precompiler option 253
- string unit 76
- STRING_DELIMITER column
 - SYSENVIRONMENT catalog table 1759
- STRINGID column
 - SYSXMLSTRINGS catalog table 1919
- STRIP function 496, 513
- strong typing 94
- structured query language (SQL)
 - binding 1
 - described 1
 - operational form of SQL statements 1
 - result table 1
- structures
 - data 4
- STYPE column of SYSCOPY catalog table 1740
- SUBBYTE column of SYSSTRINGS catalog table 1874
- subnormal numbers 73
- subquery
 - description 157
 - HAVING clause 651
 - ORDER BY clause 652
 - WHERE clause 648
- subselect
 - CREATE VIEW statement 632
 - description 632
- subselect (*continued*)
 - examples 657
 - INSERT statement 632
- substitution character 38, 111
- SUBSTR function 515
- SUBSTRING function 517
- SUBTYPE column
 - SYSDATATYPES catalog table 1749
 - SYSKEYTARGETS catalog table 1793
 - SYSPARMS catalog table 1826
- SUM function 278
- synonym
 - defining 1077
 - description 59
 - dropping 1264
 - naming convention 55
 - qualifying a column name 154
- SYNONYM clause
 - CREATE SYNONYM statement 1077
 - DROP statement 1264
- syntax diagram
 - how to read xx
- SYSADM authority
 - GRANT statement 1353
 - REVOKE statement 1457
- SYSADMAUTH column of SYSUSERAUTH catalog table 1911
- SYSCTRL authority
 - GRANT statement 1353
 - REVOKE statement 1457
- SYSCTRLAUTH column of SYSUSERAUTH catalog table 1911
- SYSENTRIES column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- SYSLGRNX directory table
 - table space
 - description 17
- SYSMODENAME column of LUNAMES catalog table 1705
- SYSOPR authority
 - GRANT statement 1354
 - REVOKE statement 1457
- SYSOPRAUTH column of SYSUSERAUTH catalog table 1911
- SYSTEM AUTHID clause
 - ALTER TRUSTED CONTEXT statement 857
- SYSTEM column
 - SYSPKSYSTEM catalog table 1830
 - SYSPLSYSTEM catalog table 1840
- system objects
 - described 16
- SYSTEM PATH clause
 - SET PATH statement 1507
- system structures
 - active log 17
 - archive log 17
 - bootstrap data set (BSDS) 18
 - buffer pools 18
 - catalog 16
 - catalog tables 16
 - SYSTEM_ASCII_CCSID session variable 168
 - SYSTEM_EBCDIC_CCSID session variable 168
 - SYSTEM_NAME session variable 168
 - SYSTEM_UNICODE_CCSID session variable 168
 - system, limits 1588
 - SYSTEMAUTHID column
 - SYSCONTEXT catalog table 1737
- SYSUTILX directory table space 17

T

table

- altering
 - ALTER TABLE statement 789
- creating
 - CREATE AUXILIARY TABLE statement 908
 - CREATE GLOBAL TEMPORARY TABLE statement 982
 - CREATE TABLE statement 1079
 - DECLARE GLOBAL TEMPORARY TABLE statement 1202
- designator 155
- dropping
 - DROP statement 1264
- joining 645
- naming convention 55
- privileges 1355
 - revoking 1458
- renaming with RENAME statement 1428
- result table 1402
- retrieving
 - catalog information 1999
 - comments 2006
 - temporary copy 1402
- Table
 - expressions, common 670
 - expressions, nested 670
- TABLE
 - column of SYSPARMS catalog table 1826
- table check constraint
 - catalog information 2004
 - defining
 - CREATE TABLE statement 1102
 - deleting rows 1233
 - inserting rows 1375
 - updating rows 1530
- TABLE clause
 - COMMENT statement 893
 - DROP statement 1264
- table function reference 640
- TABLE LIKE clause
 - CREATE FUNCTION statement 922, 945, 962
 - CREATE PROCEDURE (external) statement 1022
 - CREATE PROCEDURE (SQL - external) statement 1039
 - CREATE PROCEDURE (SQL - native) statement 1050
- table locator variable 640
- table name
 - qualifying a column name 154
 - unqualified 58
- table space
 - accelerators table 1996
 - altering with ALTER TABLESPACE statement 841
 - catalog table 1679
 - creating
 - CREATE TABLESPACE statement 1128
 - implicitly 1108
 - description 14
 - dropping 1265
 - naming convention 56
 - partition-by-growth 1143
 - range-partitioned 1143
 - universal 1143
- table spaces
 - described 14
 - description 14
 - large object 14
 - partitioned 14

table spaces (continued)

- partitioned table spaces
 - described 14
- segmented 14
- segmented table spaces
 - described 14
- simple 14
- universal 14
- TABLE_COLNO column of SYSPARMS catalog table 1826
- TABLE_LOCATION function 2018
- TABLE_NAME function 2020
- TABLE_SCHEMA function 2022
- table-name clause
 - EXCHANGE statement 1274
- TABLE
 - clause of DECLARE TABLE statement 1217
- TABLE◀
 - clause of LABEL statement 1384
- tables
 - defined 5
 - dependent 20
 - self-referencing tables 20
 - supplied by DB2
 - DSN_DETCOST_TABLE 1943
 - DSN_FILTER_TABLE 1949
 - DSN_FUNCTION_TABLE 1952
 - DSN_PGRANGE_TABLE 1955
 - DSN_PGROUPE_TABLE 1957
 - DSN_PREDICAT_TABLE 1961
 - PLAN_TABLE 1930
- TABLESPACE
 - clause of ALTER TABLESPACE statement 841
- TABLESPACE clause
 - DROP statement 1265
- TABLESPACE privilege
 - GRANT statement 1361
 - REVOKE statement 1463
- TABLESTATUS column of SYSTABLES catalog table 1890
- TAN function 522
- TANH function 523
- TARGETNAMESPACE column
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSROBJECTHIERARCHIES table 1926
 - SYSIBM.XSROBJECTS table 1922
- TBCREATOR column
 - SYSCOLUMNS catalog table 1723
 - SYSCOLUMNS_HIST catalog table 1732
 - SYSFIELDS catalog table 1761
 - SYSINDEXES catalog table 1764
 - SYSINDEXES_HIST catalog table 1769
 - SYSKEYCOLUSE catalog table 1791
 - SYSNONONYMS catalog table 1877
 - SYSTABCONST catalog table 1881
 - SYSTABLES catalog table 1890
- TBNAME column
 - SYSAXXRELS catalog table 1709
 - SYSCHKDEP catalog table 1710
 - SYSCHKES catalog table 1711
 - SYSCHKES2 catalog table 1712
 - SYSCLDIST catalog table 1715
 - SYSCLDIST_HIST catalog table 1719
 - SYSCLDISTSTATS catalog table 1717
 - SYSCLSTATS catalog table 1721
 - SYSCOLUMNS catalog table 1723
 - SYSCOLUMNS_HIST catalog table 1732
 - SYSFIELDS catalog table 1761
 - SYSFOREIGNKEYS catalog table 1763

TBNAME column (*continued*)

- SYSINDEXES catalog table 1764
- SYSINDEXES_HIST catalog table 1769
- SYSKEYCOLUSE catalog table 1791
- SYSRELS catalog table 1841
- SYSYNONYMS catalog table 1877
- SYSTABCONST catalog table 1881
- SYSTABLES catalog table 1890
- SYSTRIGGERS catalog table 1909
- SYSXMLRELS catalog table 1918

TBOWNER column

- SYSAUXRELS catalog table 1709
- SYSCHECKDEP catalog table 1710
- SYSCHECKS catalog table 1711
- SYSCHECKS2 catalog table 1712
- SYSCOLDIST catalog table 1715
- SYSCOLDIST_HIST catalog table 1719
- SYSCOLDISTSTATS catalog table 1717
- SYSCOLSTATS catalog table 1721
- SYSTRIGGERS catalog table 1909
- SYSXMLRELS catalog table 1918

TCREATOR column of SYSTABAUTH catalog table 1878

temporary

- copy of table 1402

temporary table

- creating 982, 1202

TEXT column

- SYSROUTINES catalog table 1848
- SYSROUTINESTEXT catalog table 1858
- SYSSTMT catalog table 1868
- SYSTRIGGERS catalog table 1909
- SYSVIEWS catalog table 1915

TEXT_ENVID column

- SYSROUTINES catalog table 1848

TEXT_ROWID column

- SYSROUTINES catalog table 1848

time

- arithmetic 196
- data type 88
- duration 192
- strings 89, 92

TIME

- data type
 - CREATE TABLE statement 1090
 - description 88
 - function 524

TIME FORMAT clause

- ALTER PROCEDURE (SQL - native) statement 774
- CREATE PROCEDURE (SQL - native) statement 1061

TIME FORMAT field of panel DSNTIP4 255

TIME_FORMAT column

- SYSENVIRONMENT catalog table 1759

TIME_FORMAT session variable 168

TIME_LENGTH session variable 168

timestamp

- arithmetic 197
- data type 88
- duration 192
- strings 89

TIMESTAMP

- column of SYSCHECKS catalog table 1711
- column of SYSCOPY catalog table 1740
- column of SYSDBRM catalog table 1754
- column of SYSPACKAGE catalog table 1810
- column of SYSPACKAUTH catalog table 1818
- column of SYSPACKLIST catalog table 1822
- column of SYSRELS catalog table 1841

TIMESTAMP (*continued*)

- data type
 - CREATE TABLE statement 1090
 - description 88
 - function 525
- TIMESTAMP_FORMAT function 529
- TIMESTAMP_ISO
 - function 534
- TIMESTAMPADD
 - function 527
- TIMESTAMPDIFF
 - function 535
- TNAME column of SYSCOLAUTH catalog table 1713
- TO
 - clause of CONNECT statement 899
- TO clause
 - GRANT statement 1334
- TO SAVEPOINT clause
 - ROLLBACK statement 1466
- TO_CHAR function 538, 564
- TO_DATE function 529, 539
- token in SQL 48
- TOTALENTRIES column
 - SYSINDEXSPACESTATS catalog table 1777
- TOTALORDER function 540
- TOTALROWS column
 - SYSTABLESPACESTATS catalog table 1901
- TPN column
 - LOCATIONS catalog table 1701
- TRACE privilege
 - GRANT statement 1354
 - REVOKE statement 1457
- TRACEAUTH column of SYSUSERAUTH catalog table 1911
- TRACKMOD
 - clause of CREATE TABLESPACE statement 1137
 - column of SYSTABLEPART catalog table 1882
- TRACKMOD clause
 - ALTER TABLESPACE statement 847
- TRANSLATE function 541
- TRANSPROC column of SYSSTRINGS catalog table 1874
- TRANSTAB column of SYSSTRINGS catalog table 1874
- TRANSTYPE column of SYSSTRINGS catalog table 1874
- TRIGEVENT column of SYSTRIGGERS catalog table 1909
- trigger
 - catalog information 2005
 - creating 1151
 - dropping 1265
 - name, unqualified 58
 - naming convention 56
 - SQL statements allowed in 1157
 - unqualified name 58
- TRIGGER clause
 - COMMENT statement 893
 - DROP statement 1265
- TRIGGER privilege
 - GRANT statement 1356
 - REVOKE statement 1459
- TRIGGERAUTH column
 - SYSTABAUTH catalog table 1878
- triggered-SQL-statement clause of TRIGGER statement 1157
- triggers 24
- TRIGTIME column of SYSTRIGGERS catalog table 1909
- TRUNC function 544
- TRUNC_TIMESTAMP function 546
- TRUNCATE function 544
- TRUNCATE statement
 - description 1517

- TRUNCATE statement (*continued*)
 - examples 1519
- truncation
 - numbers 105
- TRUSTED column
 - LOCATIONS catalog table 1701
- trusted context
 - altering 855
 - defining 1167
- TRUSTED CONTEXT clause
 - COMMENT statement 893
 - DROP statement 1265
- truth table 247
- truth valued logic 247
- TSNAME column
 - SYSCOPY catalog table 1740
 - SYSTABLEPART catalog table 1882
 - SYSTABLEPART_HIST catalog table 1887
 - SYSTABLES catalog table 1890
 - SYSTABLES_HIST catalog table 1905
 - SYSTABSTATS catalog table 1907
 - SYSTABSTATS_HIST catalog table 1908
- TNAME column of SYSTABAUTH catalog table 1878
- TTYPE column
 - SYSCOPY catalog table 1740
- two-phase commit 31
- TYPE clause
 - COMMENT statement 894
 - DROP statement 1265
- TYPE column
 - SYSCOLDIST catalog table 1715
 - SYSCOLDIST_HIST catalog table 1719
 - SYSCOLDISTSTATS catalog table 1717
 - SYSDATABASE catalog table 1747
 - SYSKEYTGTDIST catalog table 1801
 - SYSKEYTGTDIST_HIST catalog table 1805
 - SYSKEYTGTDISTSTATS catalog table 1803
 - SYSPACKAGE catalog table 1810
 - SYSTABCONST catalog table 1881
 - SYSTABLES catalog table 1890
 - SYSTABLESPACE catalog table 1896
 - USERNAMES catalog table 1920
- typed parameter marker 1413
- TYPENAME column
 - SYSCOLUMNS catalog table 1723
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSPARMS catalog table 1826
- TYPESHEMA column
 - SYSCOLUMNS catalog table 1723
 - SYSKEYTARGETS catalog table 1793
 - SYSKEYTARGETS_HIST catalog table 1798
 - SYSPARMS catalog table 1826

U

- UCASE function 549, 552
- UDF
 - catalog information 2005
- unary operation 182
- UNCOMPRESSED DATASIZE column
 - SYSTABLESPACESTATS catalog table 1901
- unconnectable and connected state 36
- unconnectable and unconnected state 36
- unconnected state 34
- underflow 73

- Unicode
 - definition 38
 - effect on MBCS and DBCS characters 74
- UNICODE function 550
- UNICODE_STR function 551
- UNION clause
 - duplicate rows 662
 - fullselect 662
- UNIQUE clause
 - ALTER TABLE statement 814
 - CREATE INDEX statement 993
 - CREATE TABLE statement 1091, 1100
 - SAVEPOINT statement 1468
- unique constraint 20
- unique indexes
 - described 6
- unique keys 6
- UNIQUE_COUNT column
 - SYSINDEXES catalog table 1764
- UNIQUERULE column of SYSINDEXES catalog table 1764
- unit of recovery 26
 - COMMIT statement 896
 - description 25
 - ROLLBACK statement 1465
- unit of work
 - closes cursors 1402
 - description 25
 - dynamic caching 1419
 - ending 25, 896, 1465
 - initiating 25
 - persistence of prepared statements 1419
 - referring to prepared statements 1405
- universal table space 1143
- universal time, coordinated (UTC) 132
- unqualified object names 58
- unsupported data types
 - SQLDA 1669
- untyped parameter marker 1413
- UPDATE
 - clause of TRIGGER statement 1153
 - statement
 - description 1521
 - example 1533
- UPDATE privilege
 - GRANT statement 1356
 - REVOKE statement 1459
- update rule 1530
- UPDATEAUTH column of SYSTABAUTH catalog table 1878
- UPDATECOLS column of SYSTABAUTH catalog table 1878
- UPDATES column
 - SYSCOLUMNS catalog table 1723
- UPDATESTATSTIME column
 - SYSINDEXSPACESTATS catalog table 1777
 - SYSTABLESPACESTATS catalog table 1901
- updating
 - rows in a table 1521
- UPPER function 552
- USAGE privilege
 - GRANT statement 1351, 1359
 - REVOKE statement 1453, 1461
- USEAUTH
 - column of SYSSEQUENCEAUTH catalog table 1863
- USEAUTH column of SYSRESAUTH catalog table 1843
- USER 150
- USER clause
 - SET PATH statement 1507
- USER special register 150

- user-defined function
 - altering with ALTER FUNCTION statement 701, 719
 - creating with CREATE FUNCTION statement 915, 916, 940, 958, 972
 - dropping 1259
 - privileges 1340
 - revoking 1442
 - statements allowed 1605
- user-defined function (UDF)
 - description 173
 - external functions 173
 - inheriting special registers 151
 - invocation 178
 - MQSeries functions 259
 - name, unqualified 58
 - naming convention 52
 - resolution 175
 - sample
 - ALTDAT 2009
 - ALTTIME 2012
 - CURRENCY 2014
 - DAYNAME 2016
 - MONTHNAME 2017
 - TABLE_LOCATION 2018
 - TABLE_NAME 2020
 - TABLE_SCHEMA 2022
 - WEATHER 2024
 - sourced functions 173
 - table functions 173
 - unqualified name 58
- user-defined types 30
- USERNAMES column
 - IPNAMES catalog table 1698
 - LUNAMES catalog table 1705
- USING clause
 - ALTER INDEX statement 730, 732
 - ALTER TABLESPACE statement 848
 - CREATE INDEX statement 1000, 1002
 - CREATE TABLESPACE statement 1131, 1133
 - DESCRIBE statement 1244, 1254
 - EXECUTE statement 1276
 - OPEN statement 1401
 - PREPARE statement 1407
- USING DESCRIPTOR clause
 - EXECUTE statement 1276, 1277
 - OPEN statement 1401
- USING host-variable-array clause
 - EXECUTE statement 1276
- USING TYPE DEFAULTS clause
 - CREATE TABLE statement 1107
 - DECLARE GLOBAL TEMPORARY TABLE statement 1210
- USING VALUES clause
 - MERGE statement 1393
- UTC (universal time, coordinated) 132
- UTF-16 38
- UTF-8 38

V

- VALID column
 - SYSPACKAGE catalog table 1810
 - SYSPLAN catalog table 1832
- VALIDATE
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPLAN catalog table 1832
- VALIDATE clause
 - ALTER PROCEDURE (SQL - native) statement 772

- VALIDATE clause (*continued*)
 - CREATE PROCEDURE (SQL - native) statement 1060
- validation procedure 831
- validation routine
 - VALIDPROC clause 831, 1111
- VALIDPROC clause
 - ALTER TABLE statement 831
 - CREATE TABLE statement 1111
- VALPROC column of SYSTABLES catalog table 1890
- value
 - SQL 69
- VALUE column
 - SYSCTXTTRUSTATTRS catalog table 1746
- VALUE function 119, 315, 387, 555
- VALUES clause
 - CREATE INDEX statement 1012
 - CREATE TABLE statement 1124
 - INSERT statement 1371, 1373
 - VALUES INTO statement 1537
 - VALUES statement 1536
- VALUES INTO statement
 - description 1537
 - example 1538
- VALUES statement
 - description 1536
 - example 1536
- VAR function 279
- VAR_POP function 279
- VAR_SAMP function 279
- VARBINARY
 - data type
 - description 85
 - function 556
- VARCHAR
 - data type
 - CREATE TABLE statement 1088
 - description 74
 - function 558
- VARCHAR_FORMAT function 564
- VARGRAPHIC
 - data type
 - CREATE TABLE statement 1089
 - description 84
 - function 569
- variable
 - built-in session
 - referencing 168
 - description 159
 - host
 - referencing 160
 - SQL syntax 160
 - referencing 159
 - SQL syntax 159
- VARIABLE clause
 - DECLARE VARIABLE statement 1221
- VARIANCE function 279
- VARIANCE_SAMP function 279
- VARIANT clause
 - CREATE FUNCTION statement 938, 956, 981
 - CREATE PROCEDURE (external) statement 1033
 - CREATE PROCEDURE (SQL - external) statement 1045
 - CREATE PROCEDURE (SQL - native) statement 1063
- VCAT
 - USING clause
 - ALTER INDEX statement 730
 - CREATE INDEX statement 732, 1000, 1002
 - CREATE TABLESPACE statement 1131, 1134

- VCAT clause
 - ALTER TABLESPACE statement 848
 - CREATE STOGROUP statement 1075
- VCATNAME column
 - SYSINDEXPART catalog table 1771
 - SYSTSTOGROUP catalog table 1872
 - SYSTABLEPART catalog table 1882
- VERSION
 - column of SYSDBRM catalog table 1754
 - column of SYSPACKAGE catalog table 1810
 - column of SYSPACKSTMT catalog table 1823
- VERSION clause
 - COMMENT statement 892
 - CREATE PROCEDURE (SQL - native) statement 1051
 - DROP statement 1262
- VERSION column
 - SYSINDEXES catalog table 1764
 - SYSPARMS catalog table 1826
 - SYSROUTINES catalog table 1848
 - SYSTABLES catalog table 1890
- version identification, current server 901
- VERSION session variable 168
- view
 - creating
 - CREATE VIEW statement 1184
 - dropping
 - description 1266
 - name, unqualified 58
 - naming convention 56
 - privileges 1458
 - granting 1355
 - regenerating
 - ALTER VIEW statement 867
 - unqualified name 58
 - using
 - adding comments 2006
 - read-only 1189
 - retrieving catalog information 2001
 - retrieving comments 2006
- VIEW clause
 - CREATE VIEW statement 1184
 - DROP statement 1266
- views
 - described 8
 - description 8
 - reasons for using 8
- VOLATILE
 - clause of CREATE TABLE statement 1113
- VOLATILE clause
 - ALTER TABLE statement 829
- VOLID column of SYSVOLUMES catalog table 1917
- VOLUMES clause
 - CREATE STOGROUP statement 1075
- VSAM (virtual storage access method)
 - catalog 1003

W

- WEATHER function 2024
- WEEK function 573
- WEEK_ISO function 574
- WHEN clause of TRIGGER statement 1157
- WHEN MATCHED clause
 - MERGE statement 1394
- WHEN NOT MATCHED clause
 - MERGE statement 1394

- WHENEVER statement
 - description 1539
 - example 1540
- WHERE clause
 - DELETE statement 1230
 - description 648
 - search condition 648
 - UPDATE statement 1528
- WHERE CURRENT OF clause
 - DELETE statement 1230
 - UPDATE statement 1528
- WHILE statement
 - example 1586, 1645
 - SQL procedure 1586, 1645
- WITH AUTHENTICATION clause
 - ALTER TRUSTED CONTEXT statement 864
 - CREATE TRUSTED CONTEXT statement 1173, 1174
- WITH CHECK OPTION clause of CREATE VIEW
 - statement 1186
- WITH clause
 - select-statement 676
- WITH common-table-expression clause
 - select-statement 670
- WITH common-table-expression clause of CREATE VIEW
 - statement 1186
- WITH EXPLAIN clause
 - ALTER PROCEDURE (SQL - native) statement 768
 - CREATE PROCEDURE (SQL - native) statement 1056
- WITH GRANT OPTION clause
 - GRANT statement 1334
- WITH HOLD clause of DECLARE CURSOR statement 1195
- WITH IMMEDIATE WRITE clause
 - ALTER PROCEDURE (SQL - native) statement 769
 - CREATE PROCEDURE (SQL - native) statement 1057
- WITH KEEP DYNAMIC clause
 - ALTER PROCEDURE (SQL - native) statement 770
 - CREATE PROCEDURE (SQL - native) statement 1057
- WITH PROCEDURE clause of ASSOCIATE LOCATORS
 - statement 868
- WITH RETURN clause of DECLARE CURSOR
 - statement 1196
- WITH RETURN clause of PREPARE statement 1410
- WITH ROWSET POSITIONING clause
 - DECLARE CURSOR statement 1196
 - PREPARE statement 1411
- WITH USE FOR clause
 - CREATE TRUSTED CONTEXT statement 1172
- WITHOUT AUTHENTICATION clause
 - ALTER TRUSTED CONTEXT statement 864
 - CREATE TRUSTED CONTEXT statement 1173, 1174
- WITHOUT EXPLAIN clause
 - ALTER PROCEDURE (SQL - native) statement 768
 - CREATE PROCEDURE (SQL - native) statement 1056
- WITHOUT HOLD clause of DECLARE CURSOR
 - statement 1195
- WITHOUT IMMEDIATE WRITE clause
 - ALTER PROCEDURE (SQL - native) statement 769
 - CREATE PROCEDURE (SQL - native) statement 1057
- WITHOUT KEEP DYNAMIC clause
 - ALTER PROCEDURE (SQL - native) statement 770
 - CREATE PROCEDURE (SQL - native) statement 1057
- WITHOUT RETURN clause of DECLARE CURSOR
 - statement 1196
- WITHOUT RETURN clause of PREPARE statement 1410
- WITHOUT ROWSET POSITIONING clause
 - DECLARE CURSOR statement 1196
 - PREPARE statement 1411

- WLM ENVIRONMENT clause
 - ALTER FUNCTION statement 715
 - ALTER PROCEDURE (external) statement 748
 - ALTER PROCEDURE (SQL - external) statement 754
 - CREATE FUNCTION statement 933, 954
 - CREATE PROCEDURE (external) statement 1029
 - CREATE PROCEDURE (SQL - external) statement 1042
- WLM ENVIRONMENT FOR DEBUG MODE clause
 - ALTER PROCEDURE (SQL - native) statement 766
 - CREATE PROCEDURE (SQL - native) statement 1054
- WLM_ENV_FOR_NESTED column of SYSROUTINES catalog table 1848
- WLM_ENVIRONMENT column of SYSROUTINES catalog table 1848
- work file database
 - creating 913
 - description 19
- WORKAREA column of SYSFIELDS catalog table 1761

X

- XML
 - assignment of values 113
 - comparison of values 117
 - data type
 - CREATE TABLE statement 1090
 - host variable 164
- XML operands 122
- XML pattern expression clause
 - CREATE INDEX statement 997
- XML schema repository tables
 - XSRCOMPONENT 1921
 - XSROBJECTCOMPONENTS 1924
 - XSROBJECTGRAMMAR 1925
 - XSROBJECTHIERARCHIES 1926
 - XSROBJECTPROPERTY 1927
 - XSROBJECTS 1922
 - XSRPROPERTY 1928
- XML values
 - data type 93
- XML-attribute
 - naming convention 56
- XML-element
 - naming convention 56
- XMLAGG function 281
- XMLATTRIBUTES function 575
- XMLCAST specification
 - description 210
- XMLCOMMENT function 576
- XMLCONCAT function 577
- XMLDOCUMENT function 578
- XMLELEMENT function 579
- XMLEXISTS
 - predicate 244
- XMLFOREST function 584
- XMLNAMESPACES function 587
- XMLPARSE function 589
- XMLPATTERN clause
 - CREATE INDEX statement 996
- XMLPI function 591
- XMLQUERY function 592
- XMLRELOBID column
 - SYSXMLRELS catalog table 1918
- XMLSERIALIZE function 596
- XMLTABLE table function
 - description 624

- XMLTBNAME column
 - SYSXMLRELS catalog table 1918
- XMLTBOWNER column
 - SYSXMLRELS catalog table 1918
- XMLTEXT function 599
- XSRCOMPONENTID column
 - SYSIBM.XSROBJECTCOMPONENTS table 1924
 - SYSIBM.XSROBJECTHIERARCHIES table 1926
- XSROBJECTID column
 - SYSIBM.XSROBJECTHIERARCHIES table 1926
 - SYSIBM.XSROBJECTS table 1922
- XSROBJECTNAME column
 - SYSIBM.XSROBJECTS table 1922
- XSROBJECTSCHEMA column
 - SYSIBM.XSROBJECTS table 1922

Y

- YEAR function 600



Product Number: 5635-DB2
5697-P12

Printed in USA

SC18-9854-10



Spine information:

DB2 Version 9.1 for z/OS

SQL Reference

