

DB2 Universal Database for OS/390



SQL Reference

Version 6

DB2 Universal Database for OS/390



SQL Reference

Version 6

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix H, "Notices" on page 1051.

Third Edition, Softcopy Only (May 2001)

This edition applies to Version 6 of DB2 Universal Database Server for OS/390, 5645-DB2, 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.

This softcopy version is based on the printed edition of the book and includes the changes indicated in the printed version by vertical bars. Additional changes made to this softcopy version of the manual since the hardcopy manual was published are indicated by the hash (#) symbol in the left-hand margin. Editorial changes that have no technical significance are not noted.

© Copyright International Business Machines Corporation 1982, 1999. All rights reserved.

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

Contents

	Chapter 1. Introduction	1
	Who should read this book	3
	How to use this book	3
#	Product terminology and citations	3
	SQL standards	4
	How to read the syntax diagrams	4
	Conventions for describing mixed data values	5
	How to use the DB2 library	6
	How to obtain DB2 information	8
	Summary of changes to DB2 UDB for OS/390 Version 6	10
	Summary of changes to this book	14
	 Chapter 2. DB2 concepts	17
	Structured query language	19
	Schemas	20
	Tables	21
	Indexes	22
	Keys	22
	Referential integrity	23
	Check constraints	25
	Triggers	25
	Storage structures	26
	Storage groups	26
	Databases	26
	Catalog	26
	Views	27
	Application processes, concurrency, and recovery	28
	Packages and application plans	31
	Distributed data	31
	Character conversion	38
	 Chapter 3. Language elements	43
	Characters	47
	Tokens	47
	Identifiers	48
	Naming conventions	50
	Schemas and the SQL path	57
	Aliases and synonyms	58
	Authorization IDs and authorization-names	59
	Data types	66
	Promotion of data types	81
	Casting between data types	83
	Assignment and comparison	84
	Rules for result data types	99
	Constants	101
	Special registers	104
	Column names	114
	Referencing host variables	120
	Host structures in PL/I, C, and COBOL	123
	Functions	125

Expressions	131
Predicates	150
Search conditions	163
Options affecting SQL	164
Chapter 4. Built-in functions	173
Column functions	178
Scalar functions	187
Chapter 5. Queries	307
Authorization	309
subselect	311
fullselect	327
select-statement	331
Chapter 6. Statements	337
How SQL statements are invoked	340
ALLOCATE CURSOR	346
ALTER DATABASE	348
ALTER FUNCTION	351
ALTER INDEX	364
ALTER PROCEDURE (external)	376
ALTER PROCEDURE (SQL)	384
ALTER STOGROUP	390
ALTER TABLE	393
ALTER TABLESPACE	412
ASSOCIATE LOCATORS	424
BEGIN DECLARE SECTION	427
CALL	428
CLOSE	436
COMMENT ON	438
COMMIT	444
CONNECT	446
CONNECT (Type 1)	449
CONNECT (Type 2)	454
CREATE ALIAS	457
CREATE AUXILIARY TABLE	459
CREATE DATABASE	462
CREATE DISTINCT TYPE	465
CREATE FUNCTION	472
CREATE FUNCTION (external scalar)	473
CREATE FUNCTION (external table)	492
CREATE FUNCTION (sourced)	508
CREATE GLOBAL TEMPORARY TABLE	520
CREATE INDEX	525
CREATE PROCEDURE (external)	541
CREATE PROCEDURE (SQL)	555
CREATE STOGROUP	565
CREATE SYNONYM	568
CREATE TABLE	570
CREATE TABLESPACE	597
CREATE TRIGGER	615
CREATE VIEW	627
DECLARE CURSOR	634

|
|

|
|
|
|
|
|
|
|

|

#	DECLARE GLOBAL TEMPORARY TABLE	639
	DECLARE STATEMENT	649
	DECLARE TABLE	650
	DELETE	653
	DESCRIBE (prepared statement or table)	659
	DESCRIBE CURSOR	666
	DESCRIBE INPUT	668
	DESCRIBE PROCEDURE	671
	DROP	674
	END DECLARE SECTION	687
	EXECUTE	689
	EXECUTE IMMEDIATE	692
	EXPLAIN	694
	FETCH	707
	FREE LOCATOR	710
	GRANT	711
	GRANT (collection privileges)	714
	GRANT (database privileges)	715
	GRANT (distinct type privileges)	718
	GRANT (function or procedure privileges)	720
	GRANT (package privileges)	725
	GRANT (plan privileges)	727
	GRANT (schema privileges)	728
	GRANT (system privileges)	730
	GRANT (table or view privileges)	733
	GRANT (use privileges)	736
	HOLD LOCATOR	738
	INCLUDE	740
	INSERT	742
	LABEL ON	749
	LOCK TABLE	751
	OPEN	753
	PREPARE	757
	RELEASE (connection)	765
#	RELEASE SAVEPOINT	768
	RENAME	769
	REVOKE	772
	REVOKE (collection privileges)	777
	REVOKE (database privileges)	778
	REVOKE (distinct type privileges)	781
	REVOKE (function or procedure privileges)	783
	REVOKE (package privileges)	788
	REVOKE (plan privileges)	790
	REVOKE (schema privileges)	791
	REVOKE (system privileges)	793
	REVOKE (table or view privileges)	796
	REVOKE (use privileges)	799
	ROLLBACK	801
#	SAVEPOINT	804
	SELECT INTO	806
	SET CONNECTION	809
	SET CURRENT DEGREE	812
	SET CURRENT LOCALE LC_CTYPE	814
	SET CURRENT OPTIMIZATION HINT	816

	SET CURRENT PACKAGESET	817
	SET CURRENT PATH	819
	SET CURRENT PRECISION	822
	SET CURRENT RULES	823
	SET CURRENT SQLID	824
#	SET host-variable assignment	826
#	SET transition-variable assignment	828
	SIGNAL SQLSTATE	831
	UPDATE	833
	VALUES	842
	VALUES INTO	843
	WHENEVER	845
#	Chapter 7. SQL procedure statements	847
#	SQL-procedure-statement	848
#	Assignment statement	849
#	CALL statement	851
#	CASE statement	853
#	Compound statement	855
#	GET DIAGNOSTICS statement	861
#	GOTO statement	862
#	IF statement	864
#	LEAVE statement	865
#	LOOP statement	866
#	REPEAT statement	867
#	WHILE statement	868
	Appendix A. Limits in DB2 for OS/390	869
	Appendix B. Characteristics of SQL statements in DB2 for OS/390	873
	Actions allowed on SQL statements	873
	SQL statements allowed in external functions and stored procedures	877
	SQL statements allowed in SQL procedures	879
	Appendix C. SQLCA and SQLDA	883
	SQL communication area (SQLCA)	883
	SQL descriptor area (SQLDA)	890
	Appendix D. DB2 catalog tables	911
	Table spaces and indexes	911
	New and changed catalog tables	917
	SYSIBM.IPNAMES table	920
	SYSIBM.LOCATIONS table	921
	SYSIBM.LULIST table	922
	SYSIBM.LUMODES table	923
	SYSIBM.LUNAMES table	924
	SYSIBM.MODESELECT table	926
	SYSIBM.SYSAUXRELS table	927
	SYSIBM.SYSCHECKDEP table	928
	SYSIBM.SYSCHECKS table	929
	SYSIBM.SYSCOLAUTH table	930
	SYSIBM.SYSCOLDIST table	931
	SYSIBM.SYSCOLDISTSTATS table	932
	SYSIBM.SYSCOLSTATS table	933

	SYSIBM.SYSCOLUMNS table	934
	SYSIBM.SYSCONSTDEP table	940
	SYSIBM.SYSCOPY table	941
	SYSIBM.SYSDATABASE table	944
	SYSIBM.SYSDATATYPES table	946
	SYSIBM.SYSDBAUTH table	947
	SYSIBM.SYSDBRM table	950
	SYSIBM.SYSDUMMY1 table	952
	SYSIBM.SYSFIELDS table	953
	SYSIBM.SYSFOREIGNKEYS table	954
	SYSIBM.SYSINDEXES table	955
	SYSIBM.SYSINDEXPART table	958
	SYSIBM.SYSINDEXSTATS table	960
	SYSIBM.SYSKEYS table	961
	SYSIBM.SYSLOBSTATS table	962
	SYSIBM.SYSPACKAGE table	963
	SYSIBM.SYSPACKAUTH table	968
	SYSIBM.SYSPACKDEP table	969
	SYSIBM.SYSPACKLIST table	970
	SYSIBM.SYSPACKSTMT table	971
	SYSIBM.SYSPARMS table	973
	SYSIBM.SYSPKSYSTEM table	975
	SYSIBM.SYSPLAN table	976
	SYSIBM.SYSPLANAUTH table	979
	SYSIBM.SYSPLANDEP table	980
	SYSIBM.SYSPLSYSTEM table	981
	SYSIBM.SYSPROCEDURES table	982
	SYSIBM.SYSRELS table	985
	SYSIBM.SYSRESAUTH table	986
	SYSIBM.SYSROUTINEAUTH table	988
	SYSIBM.SYSROUTINES table	989
	SYSIBM.SYSSCHEMAAUTH table	995
#	SYSIBM.SYSSEQUENCES table	996
#	SYSIBM.SYSSEQUENCESDEP table	997
	SYSIBM.SYSSTMT table	998
	SYSIBM.SYSSTOGROUP table	1000
	SYSIBM.SYSSTRINGS table	1001
	SYSIBM.SYSSYNONYMS table	1003
	SYSIBM.SYSTABAUTH table	1004
	SYSIBM.SYSTABLEPART table	1007
	SYSIBM.SYSTABLES table	1010
	SYSIBM.SYSTABLESPACE table	1014
	SYSIBM.SYSTABSTATS table	1017
	SYSIBM.SYSTRIGGERS table	1018
	SYSIBM.SYSUSERAUTH table	1019
	SYSIBM.SYSVIEWDEP table	1022
	SYSIBM.SYSVIEWS table	1023
	SYSIBM.SYSVOLUMES table	1024
	SYSIBM.USERNAMES table	1025
	Appendix E. SQL reserved words	1027
	Appendix F. Sample user-defined functions	1029
	ALTDATE	1030

Contents

ALTTIME	1033
CURRENCY	1035
DAYNAME	1037
MONTHNAME	1038
TABLE_LOCATION	1039
TABLE_NAME	1041
TABLE_SCHEMA	1043
WEATHER	1045
Appendix G. DB2 objects required by the DB2 for OS/390 SQL	
procedure processor	1047
Table spaces and indexes	1047
The SQL procedure source table (SYSIBM.SYSPSM)	1047
The SQL procedure options table (SYSIBM.SYSPSMOPTS)	1048
Created temporary table SYSIBM.SYSPSMOUT	1049
Appendix H. Notices	1051
Programming interface information	1052
Trademarks	1053
Glossary	1055
Bibliography	1075
Index	I-1

Chapter 1. Introduction

	Who should read this book	3
	How to use this book	3
#	Product terminology and citations	3
	SQL standards	4
	How to read the syntax diagrams	4
	Conventions for describing mixed data values	5
	How to use the DB2 library	6
	How to obtain DB2 information	8
	DB2 on the Web	8
	DB2 publications	8
	DB2 education	9
	How to order the DB2 library	9
	Subscription through the Publication Notification System (PNS)	10
	Summary of changes to DB2 UDB for OS/390 Version 6	10
	Capacity improvements	10
	Performance and availability	10
	Data sharing enhancements	11
	User productivity	11
	Network computing	12
	Object-relational extensions and active data	13
	More function	13
	Features of DB2 for OS/390	13
	Migration considerations	14
	Summary of changes to this book	14

Who should read this book

This book serves as a reference for Structured Query Language (SQL) for DB2 Universal Database Server™ for OS/390 (DB2® for OS/390®). It is intended for end users, application programmers, system and database administrators, and for persons involved in error detection and diagnosis.

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

Unless otherwise stated, references to SQL in this book imply SQL for DB2 for OS/390, and all objects described in this book are objects of DB2 for OS/390. 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 SQL Reference* if you intend to develop applications that adhere to IBM SQL.

How to use this book

This book has the following sections:

- “Chapter 1. Introduction” on page 1 identifies the purpose, the audience, and the use of the book.
- “Chapter 2. DB2 concepts” on page 17 describes the basic concepts of relational databases and SQL.
- “Chapter 3. Language elements” on page 43 describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- “Chapter 4. Built-in functions” on page 173 contains syntax diagrams, semantic descriptions, rules, and use examples of SQL column and scalar functions.
- “Chapter 5. Queries” on page 307 describes the various forms of a query, which is a component of various SQL statements.
- “Chapter 6. Statements” on page 337 contains syntax diagrams, semantic descriptions, rules, and examples of all SQL statements.
- “Chapter 7. SQL procedure statements” on page 847 contains syntax diagrams, semantic descriptions, rules, and examples of SQL procedure statements.
- The appendixes contain information about DB2 limits, SQLCA, SQLDA, catalog tables, and SQL reserved words.

#

When you first use this book, consider reading Chapters 2 and 3 sequentially. The rest of the book is designed for the quick location of answers to specific SQL questions.

Product terminology and citations

In this book, DB2 Universal Database Server for OS/390 is referred to as "DB2 for OS/390." In cases where the context makes the meaning clear, DB2 for OS/390 is referred to as "DB2." When this book refers to other books in this library, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to *IBM DATABASE 2 Universal Database Server for OS/390 SQL Reference*.)

References in this book to "DB2 UDB" relate to the DB2 Universal Database™ product that is available on the AIX®, OS/2®, and Windows NT™ operating systems. When this book refers to books about the DB2 UDB product, the citation includes the complete title and order number.

The following terms are used as indicated:

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

C and C language Represent the C programming language.

CICS® Represents CICS/ESA® and CICS Transaction Server for OS/390 Release 1.

IMS™ Represents IMS/ESA®.

MVS Represents the MVS element of OS/390.

SQL standards

In this book, the use of the term *SQL standard* refers collectively to:

- FIPS (Federal Information Processing Standards) publication 127-2, Database Language SQL, which announces ANSI X3.135-1992 as the standard for SQL
- ANSI (American National Standards Institute) X3.135-1992, Database Language SQL
- ISO (International Standards Organization) 9075-1992, Database Language SQL

How to read the syntax diagrams

The following rules apply to the syntax diagrams used in this book:

- 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.

Diagrams of syntactical units other than complete statements start with the **▶—** symbol and end with the **—▶** symbol.

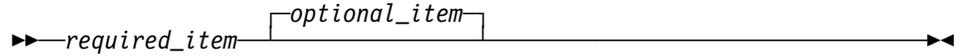
- Required items appear on the horizontal line (the main path).

▶▶—*required_item*—▶◀

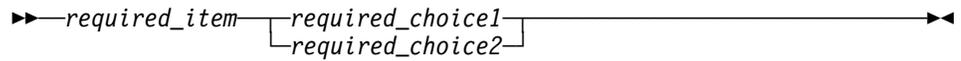
- Optional items appear below the main path.

▶▶—*required_item*——*optional_item*——▶◀

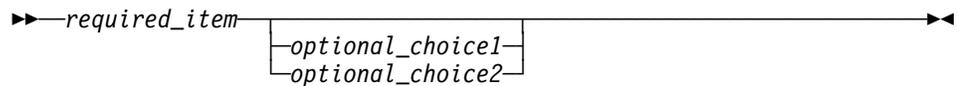
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



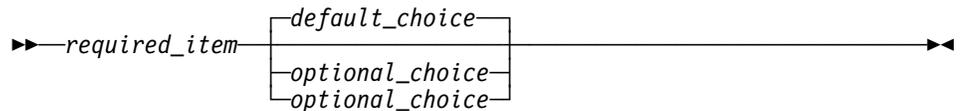
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



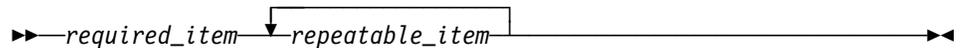
If choosing one of the items is optional, the entire stack appears below the main path.



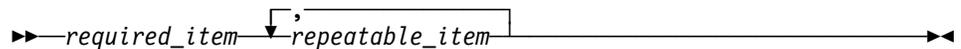
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- 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.

- Keywords appear in uppercase (for example, FROM). They 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

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 following conventions apply:

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

How to use the DB2 library

Titles of books in the library begin with DB2 Universal Database for OS/390 Version 6. 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. For a complete list of books in the library, and the sections in each book, see the bibliography at the back of this book.

Throughout the library, the DB2 for OS/390 licensed program and a particular DB2 for MVS/ESA subsystem are each referred to as “DB2.” In each case, the context makes the meaning clear.

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 associated with DB2 are grouped into the following major categories (but supplemental information relating to all of the below tasks for new releases of DB2 can be found in *DB2 Release Guide*):

Installation: If you are involved with DB2 only to install the system, *DB2 Installation Guide* might be all you need.

If you will be using data sharing then you also need *DB2 Data Sharing: Planning and Administration*, which describes installation considerations for data sharing.

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 SPUFI, and an alphabetized reference to the types of SQL statements. This information is found in *DB2 Application Programming and SQL Guide* and this book.

End users can also issue SQL statements through the Query Management Facility (QMF™) or some other program, and the library for that program might provide all the instruction or reference material they need. For a list of the titles in the QMF library, see the bibliography at the end of this book.

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 *DB2 Application Programming and SQL Guide*, *DB2 SQL Reference*, and *DB2 ODBC Guide and Reference* just as end users do.

Application programmers also need instructions on many other topics:

- How to transfer data between DB2 and a host program—written in COBOL, C, or FORTRAN, for example
- How to prepare to compile a program that embeds SQL statements
- How to process data from two systems simultaneously, say DB2 and IMS or DB2 and CICS®
- How to write distributed applications across platforms
- How to write applications that use DB2 ODBC to access DB2 servers
- How to write applications that use Open Database Connectivity (ODBC) to access DB2 servers
- How to write applications in the Java™ programming language to access DB2 servers

The material needed for writing a host program containing SQL is in *DB2 Application Programming and SQL Guide* and in *DB2 Application Programming Guide and Reference for Java™*. The material needed for writing applications that use DB2 ODBC or ODBC to access DB2 servers is in *DB2 ODBC Guide and Reference*. For handling errors, see *DB2 Messages and Codes*.

Information about writing applications across platforms can be found in *Distributed Relational Database Architecture™: Application Programming Guide*.

System and Database Administration: *Administration* covers almost everything else. *DB2 Administration Guide* divides those tasks among the following sections:

- Section 2 (Volume 1) of *DB2 Administration Guide* discusses the decisions that must be made when designing a database and tells how to bring the design into being by creating DB2 objects, loading data, and adjusting to changes.
- Section 3 (Volume 1) of *DB2 Administration Guide* 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.
- Section 4 (Volume 1) of *DB2 Administration Guide* 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.
- Section 5 (Volume 2) of *DB2 Administration 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.

In addition, the appendixes in *DB2 Administration Guide* contain valuable information on DB2 sample tables, National Language Support (NLS), writing exit routines, interpreting DB2 trace output, and character conversion for distributed data.

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, then 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, then 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 Codes*, which lists 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* and *DB2 Messages and Codes*.

How to obtain DB2 information

DB2 on the Web

Stay current with the latest information about DB2. View the DB2 home page on the World Wide Web. News items keep you informed about the latest enhancements to the product. Product announcements, press releases, fact sheets, and technical articles help you plan your database management strategy.

You can view and search DB2 publications on the Web, or you can download and print many of the most current DB2 books. Follow links to other Web sites with more information about DB2 family and OS/390 solutions. Access DB2 on the Web at the following address:

<http://www.ibm.com/software/db2os390>

DB2 publications

The DB2 publications for DB2 Universal Database Server for OS/390 are available in both hardcopy and softcopy format.

BookManager® format

Use online books on CD-ROM or DVD, or the Web. You can read, search across books, print portions of the text, and make notes in these BookManager books. With the IBM Library Reader, you can view these books in the OS/390, VM, OS/2, DOS, AIX, and Windows™ environments. You can also view many of the DB2 BookManager books on the Web.

PDF format

Many of the DB2 books are available in Portable Document Format (PDF) for viewing or printing from CD-ROM, DVD, or the Web. Download the PDF books to your intranet for distribution throughout your enterprise.

#

CD-ROMs and DVD

Books for Version 6 of DB2 Universal Database Server for OS/390 are available on CD-ROMs and DVD:

- *DB2 UDB for OS/390 Version 6 Licensed Online Book*, LK3T-3519, containing *DB2 UDB for OS/390 Version 6 Diagnosis Guide and Reference* in BookManager format, for ordering with the product.
- *DB2 UDB Server for OS/390 Version 6 Online and PDF Library*, SK3T-3518, a collection of books for the DB2 server in BookManager and PDF formats.

Periodically, the books will be refreshed on subsequent editions of these CD-ROMs. The books for Version 6 of DB2 UDB Server for OS/390 are also available on the following collection kits that contain online books for many IBM products:

- *Online Library Omnibus Edition OS/390 Collection*, SK2T-6700, in English
- *z/OS Software Collection*, SK3T-4270,, in English
- *z/OS and Software Products DVD Collection*, SK3T-4271-00, in English

#

#

DB2 education

IBM Education and Training offers a wide variety of classroom courses to help you quickly and efficiently gain DB2 expertise. Classes are scheduled in cities all over the world. You can find class information, by country, at the IBM Learning Services Web site:

<http://www.ibm.com/services/learning/>

For more information, including the current local schedule, please contact your IBM representative.

Classes can also be taught at your location, at a time that suits your needs. Courses can even be customized to meet your exact requirements. The *All-in-One Education and Training Catalog* describes the DB2 curriculum in the United States. You can inquire about or enroll in these courses by calling 1-800-IBM-TEACH (1-800-426-8322).

How to order the DB2 library

You can order DB2 publications and CD-ROMs through your IBM representative or the IBM branch office serving your locality. If you are located 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 1-800-565-1234.

To order additional copies of licensed publications, specify the SOFTWARE option. To order additional publications or CD-ROMs, specify the PUBLICATIONS and SLSS option. Be prepared to give your customer number, the product number, and the feature code(s) or order numbers you want.

Subscription through the Publication Notification System (PNS)

IBM has replaced the System Library Subscription Service (SLSS) with an up-to-date notification application, the Publication Notification System (PNS). IBM migrated all active SLSS subscriptions to the new PNS application, which you can access from the Web.

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. Unlike SLSS, 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. To access the PNS application on the World Wide Web, enter the following address on your Web browser command line:

www.ibm.com/shop/publications/pns/elink.ibm.com

Summary of changes to DB2 UDB for OS/390 Version 6

DB2 UDB for OS/390 Version 6 delivers an enhanced relational database server solution for OS/390. This release focuses on greater capacity, performance improvements for utilities and queries, easier database management, more powerful network computing, and DB2 family compatibility with rich new object-oriented capability, triggers, and more built-in functions.

Capacity improvements

16-terabyte tables provide a significant increase to table capacity for partitioned and LOB table spaces and indexes, and for nonpartitioning indexes.

Buffer pools in data spaces provide virtual storage constraint relief for the ssnmDBM1 address space, and data spaces increase the maximum amount of virtual buffer pool space allowed.

Performance and availability

Improved partition rebalancing lets you redistribute partitioned data with minimal impact to data availability. One REORG of a range of partitions both reorganizes and rebalances the partitions.

You can **change checkpoint frequency dynamically** using the new SET LOG command and initiate checkpoints any time while your subsystem remains available.

Utilities that are faster, more parallel, easier to use:

- **Faster backup and recovery** enables COPY and RECOVER to process a list of objects in parallel, and recover indexes and table spaces at the same time from image copies and the log.
- **Parallel index build** reduces the elapsed time of LOAD and REORG jobs of table spaces, or partitions of table spaces, that have more than one index; the elapsed time of REBUILD INDEX jobs is also reduced.
- Tests show **decreased elapsed and processor time for online REORG.**
- **Inline statistics** embeds statistics collection into utility jobs, making table spaces available sooner.

- You can **determine when to run REORG** by specifying threshold limits for relevant statistics from the DB2 catalog.

Query performance enhancements include:

- **Query parallelism extensions** for complex queries, such as outer joins and queries that use nonpartitioned tables
- **Improved workload balancing in a Parallel Sysplex®** that reduces elapsed time for a single query that is split across active DB2 members
- **Improved data transfer** that lets you request multiple DRDA query blocks when performing high-volume operations
- The ability to use an **index to access predicates with noncorrelated IN subqueries**
- **Faster query processing** of queries that include join operations

More performance and availability enhancements include:

- **Faster restart and recovery** with the ability to postpone backout work during restart, and a faster log apply process
- **Increased flexibility with 8-KB and 16-KB page sizes** for balancing different workload requirements more efficiently, and for controlling traffic to the coupling facility for some workloads
- **Direct-row access** using the new ROWID data type to re-access a row directly without using the index or scanning the table
- **Ability to retain prior access path** when you rebind a statement. You almost always get the same or a better access path. For the exceptional cases, Version 6 of DB2 for OS/390 lets you retain the access path from a prior BIND by using rows in an Explain table as input to optimization.
- An **increased log output buffer size** (from 1000 4-KB to 100000 4-KB buffers) that improves log read and write performance

Data sharing enhancements

More caching options use the coupling facility to improve performance in a data sharing environment for some applications by writing changed pages directly to DASD.

Control of space map copy maintenance with a new option avoids tracking of page changes, thereby optimizing performance of data sharing applications.

User productivity

Predictive governing capabilities enhance the resource limit facility to help evaluate resource consumption for queries that run against large volumes of data.

Statement cost estimation of processing resource that is needed for an SQL statement helps you to determine error and warning thresholds for governing, and to decide which statements need tuning.

A **default buffer pool** for user data and indexes isolates user data from the DB2 catalog and directory, and separating user data from system data helps you make better tuning decisions.

More information available for monitoring DB2 includes data set I/O activity in traces, both for batch reporting and online monitors.

Better integration of DB2 and Workload Manager delay reporting enables DB2 to notify Workload Manager about the current state of a work request.

More tables are allowed in SQL statements SELECT, UPDATE, INSERT, and DELETE, and in views. DB2 increases the limit from 15 to 225 tables. The number of tables and views in a subselect is not changed.

Improved DB2 UDB family compatibility includes SQL extensions, such as:

- A VALUES clause of INSERT that supports any expression
- A new VALUES INTO statement

Easier recovery management lets you achieve a single point of recovery and recover data at a remote site more easily.

Enhanced database commands extend support for pattern-matching characters (*) and let you filter display output.

You can easily **process dynamic SQL in batch mode** with the new object form of DSNTEP2 shipped with DB2 for OS/390.

Network computing

SQLJ, the newest Java implementation for the OS/390 environment, supports SQL embedded in the Java programming language. With SQLJ, your Java programs benefit from the superior performance, manageability, and authorization available to static SQL, and they are easy to write.

DRDA® support for three-part names offers more functionality to applications using three-part names for remote access and improves the performance of client/server applications.

Stored procedure enhancements include the ability to create and modify stored procedure definitions, make nested calls for stored procedures and user-defined functions, and imbed CALL statements in application programs or dynamically invoke CALL statements from IBM's ODBC and CLI drivers.

DB2 ODBC extensions include new and modified APIs and new data types to support the object-relational extensions.

ODBC access to DB2 for OS/390 catalog data improves the performance of your ODBC catalog queries by redirecting them to shadow copies of DB2 catalog tables.

Better performance for ODBC applications reduces the number of network messages that are exchanged when an application executes dynamic SQL.

Improvements for dynamically prepared SQL statements include a new special register that you use to implicitly qualify names of distinct types, user-defined functions, and stored procedures.

DDF connection pooling uses a new type of inactive thread that improves performance for large volumes of inbound DDF connections.

Object-relational extensions and active data

The object extensions of DB2 offer the benefits of object-oriented technology while increasing the strength of your relational database with an enriched set of data types and functions. Complementing these extensions is a powerful mechanism, triggers, that brings application logic into the database that governs the following new structures:

- **Large objects (LOBs)** are well suited to represent large, complex structures in DB2 tables. Now you can make effective use of multimedia by storing objects such as complex documents, videos, images, and voice. Some key elements of LOB support include:
 - LOB data types for storing byte strings up to 2 GB in size
 - LOB locators for easily manipulating LOB values in manageable pieces
 - Auxiliary tables (that reside in LOB table spaces) for storing LOB values
- **Distinct types** (which are sometimes called user-defined data types), like built-in data types, describe the data that is stored in columns of tables where the instances (or objects) of these data types are stored. They ensure that only those functions and operators that are explicitly defined on a distinct type can be applied to its instances.
- **User-defined functions**, like built-in functions or operators, support manipulation of distinct type instances (and built-in data types) in SQL queries.
- **New and extended built-in functions** improve the power of the SQL language with about 100 new built-in functions, extensions to existing functions, and sample user-defined functions.

Triggers automatically execute a set of SQL statements whenever a specified event occurs. These statements validate and edit database changes, read and modify the database, and invoke functions that perform operations inside and outside the database.

You can use the **DB2 Extenders** feature of DB2 for OS/390 to store and manipulate image, audio, video, and text objects. The extenders automatically capture and maintain object information and provide a rich body of APIs.

More function

Some function and capability is available to both Version 6 and Version 5 users. Learn how to obtain these functions now, prior to migrating to Version 6, by visiting the following Web site:

<http://www.software.ibm.com/data/db2/os390/v5apar.html>

Features of DB2 for OS/390

DB2 for OS/390 Version 6 offers a number of tools, which are optional features of the server, that are shipped to you automatically when you order DB2 Universal Database for OS/390:

- DB2 Management Tools Package, which includes the following elements:
 - DB2 UDB Control Center
 - DB2 Stored Procedures Builder
 - DB2 Installer

- DB2 Visual Explain
- DB2 Estimator
- Net.Data® for OS/390

You can install and use these features in a “Try and Buy” program for up to 90 days without paying license charges:

- Query Management Facility
- DB2 DataPropagator™
- DB2 Performance Monitor
- DB2 Buffer Pool Tool
- DB2 Administration Tool

Migration considerations

Migration to Version 6 eliminates all type 1 indexes, shared read-only data, data set passwords, use of host variables without the colon, and RECOVER INDEX usage. You can migrate to Version 6 only from a Version 5 subsystem.

Summary of changes to this book

The major changes to this book are:

Chapter 3. Language elements is enhanced and extended for the new data types (LOBs, row IDs, and distinct types) and for user-defined functions. The chapter also contains the descriptions of several new special registers (CURRENT OPTIMIZATION HINTS, CURRENT LOCALE LC_CTYPE, and CURRENT PATH) and shows some new forms for an expression.

Chapter 4. Built-in functions includes descriptions of over 60 new built-in functions. (See Table 24 on page 173 for a list and brief description of all the functions.)

Chapter 5. Queries contains new syntax for specifying a table function or table locator as an intermediate result table.

Chapter 6. Statements includes many new statements as well as changed statements in support of changes that are described under “Summary of changes to DB2 UDB for OS/390 Version 6” on page 10. The new statements are:

- “ALTER FUNCTION” on page 351
- “ALTER PROCEDURE (external)” on page 376
- “ALTER PROCEDURE (SQL)” on page 384
- “CREATE AUXILIARY TABLE” on page 459
- “CREATE DISTINCT TYPE” on page 465
- “CREATE FUNCTION” on page 472
- “CREATE PROCEDURE (external)” on page 541
- “CREATE PROCEDURE (SQL)” on page 555
- “CREATE TRIGGER” on page 615
- “DESCRIBE INPUT” on page 668
- “FREE LOCATOR” on page 710
- “GRANT (distinct type privileges)” on page 718
- “GRANT (function or procedure privileges)” on page 720
- “GRANT (schema privileges)” on page 728
- “HOLD LOCATOR” on page 738

- “REVOKE (distinct type privileges)” on page 781
- “REVOKE (function or procedure privileges)” on page 783
- “REVOKE (schema privileges)” on page 791
- “SET CURRENT LOCALE LC_CTYPE” on page 814
- “SET CURRENT OPTIMIZATION HINT” on page 816
- “SET CURRENT PATH” on page 819
- “SET host-variable assignment” on page 826
- “SET transition-variable assignment” on page 828
- “SIGNAL SQLSTATE” on page 831
- “VALUES” on page 842
- “VALUES INTO” on page 843

Statements with new clauses, new values for existing clauses, or other changes include:

- “ALTER DATABASE” on page 348
- “ALTER INDEX” on page 364
- “ALTER STOGROUP” on page 390
- “ALTER TABLE” on page 393
- “ALTER TABLESPACE” on page 412
- “ASSOCIATE LOCATORS” on page 424
- “CALL” on page 428
- “CREATE DATABASE” on page 462
- “CREATE INDEX” on page 525
- “LOCK TABLE” on page 751
- “CREATE GLOBAL TEMPORARY TABLE” on page 520
- “CREATE STOGROUP” on page 565
- “CREATE TABLE” on page 570
- “CREATE TABLESPACE” on page 597
- “CREATE VIEW” on page 627
- “DELETE” on page 653
- “DESCRIBE (prepared statement or table)” on page 659
- “DESCRIBE PROCEDURE” on page 671
- “DROP” on page 674
- “EXECUTE” on page 689
- “EXPLAIN” on page 694
- “FETCH” on page 707
- “GRANT (table or view privileges)” on page 733
- “INSERT” on page 742
- “OPEN” on page 753
- “PREPARE” on page 757
- “REVOKE (table or view privileges)” on page 796
- “SELECT INTO” on page 806
- “UPDATE” on page 833

#

Chapter 7. SQL procedure statements is added to describe the statements that can be used in SQL procedures.

Appendix C, SQLCA and SQLDA describes changes to the SQLDA to support LOBs and distinct types.

Appendix D, DB2 catalog tables includes descriptions of nine new catalog tables. (See “New and changed catalog tables” on page 917 for a summary of all catalog table changes.)

Appendix F, Sample user-defined functions is a new appendix that contains descriptions of the sample user-defined functions that are provided with DB2.

#

Appendix G, DB2 objects required by the DB2 for OS/390 SQL procedure processor is a new appendix that describes tables that are required by the DB2 for OS/390 SQL procedure processor and DB2 Stored Procedure Builder.

Chapter 2. DB2 concepts

Structured query language	19
Static SQL	19
Dynamic SQL	19
Deferred embedded SQL	19
Interactive SQL	20
DB2 Open Database Connectivity (ODBC)	20
DB2 access for Java (JDBC and SQLJ)	20
Schemas	20
Tables	21
Indexes	22
Keys	22
Unique keys	22
Primary keys	22
Parent keys	23
Foreign keys	23
Referential integrity	23
Check constraints	25
Triggers	25
Storage structures	26
Storage groups	26
Databases	26
Catalog	26
Views	27
Application processes, concurrency, and recovery	28
Locking, commit, and rollback	28
Unit of work	29
Unit of recovery	29
Rolling back work	30
Packages and application plans	31
Distributed data	31
DRDA access	32
DB2 private protocol access	33
Connection management for DRDA access and DB2 private protocol	34
Character conversion	38
Character sets and code pages	39
System CCSIDS	40
Restrictions on BIT data	41
Expanding conversions	41
Contracting conversions	42

Structured query language

Structured query language (SQL) is a standardized language for defining and manipulating data in a relational database. In accordance with the relational model of data, the database is perceived as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more tables. DB2 for OS/390 transforms the specification of a result table into a sequence of internal 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 be executed. The result of preparation is the executable or *operational form* of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish *static* SQL from *dynamic* SQL.

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.

A source program that contains static SQL statements must be processed by an SQL precompiler before it is compiled. The precompiler 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, as described in Section 6 of *DB2 Application Programming and SQL Guide*.

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 statement PREPARE or EXECUTE IMMEDIATE. Whether the operational form of the statement is persistent depends on whether dynamic statement caching is enabled. For details on dynamic statement caching, see Section 7 of *DB2 Application Programming and SQL Guide*.

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

In this book, *interactive SQL* refers to SQL statements submitted to SPUFI (SQL processor using file input). SPUFI prepares and executes these statements dynamically. For more details about using SPUFI, see Section 2 of *DB2 Application Programming and SQL Guide*.

DB2 Open Database Connectivity (ODBC)

DB2 Open Database Connectivity (DB2 ODBC) is an alternative to using embedded static or dynamic SQL. DB2 ODBC is an application programming interface in which functions are provided to application programs to process SQL statements. The function calls are available only for C and C++ application programs. Through the interface, the application invokes a C function at execution time to connect to the data source, to issue SQL statements, and to get returned data and status information. Unlike using embedded SQL, no precompilation is required. Applications developed using this interface might be executed on a variety of data sources without being compiled against each of the databases. Note that only C and C++ applications can use this interface.

DB2 ODBC provides a consistent interface to query and retrieve system catalog information across the DB2 family of database management systems. This interface reduces the need to write catalog queries that are specific to each database server. DB2 ODBC can return result sets to those programs.

The *DB2 ODBC Guide and Reference* describes the APIs supported with this interface.

DB2 access for Java (JDBC and SQLJ)

JavaSoft™ Java Database Connectivity (JDBC) and SQLJ are two methods for accessing DB2 data from the Java programming language. In general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL.

JDBC is an application programming interface (API) that Java applications can use to access any relational database. JDBC is similar to ODBC and is based on the X/Open SQL Call Level Interface specification.

SQLJ is an API that provides support for embedded static SQL in Java applications. Because DB2 for OS/390 SQLJ support includes JDBC, SQLJ applications can also execute dynamic SQL statements through JDBC.

The *DB2 Application Programming Guide and Reference for Java™* describes the APIs supported with these interfaces.

Schemas

A *schema* is a collection of named objects. The objects that a schema can contain include 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. When a distinct type, function, 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 second part is the name of the object. When a

stored procedure is created, it is given a three-part name. The first part is a location name, which is implicitly or explicitly specified, the second part is the schema name, which is implicitly or explicitly specified, and the third part is the name of the object.

Schemas extend the concept of qualifiers for tables, views, indexes, and aliases to enable the qualifiers for distinct types, functions, stored procedures, and triggers to be called schema names.

Tables

Tables are logical structures maintained by DB2. Tables are made up of *columns* and *rows*. There is no inherent order of the rows within a 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.

Some types of tables include:

base table A table created with the SQL statement CREATE TABLE and used to hold persistent user data.

auxiliary table A table created with the SQL statement CREATE AUXILIARY TABLE and used to hold the data for a column that is defined in a base table.

temporary table A table defined by either the SQL statement CREATE GLOBAL TEMPORARY TABLE (a created temporary table) or DECLARE GLOBAL TEMPORARY TABLE (a declared temporary table) and used to hold data temporarily, such as the intermediate results of SQL transactions. Both created temporary tables and declared temporary tables persist only as long as the application process. The description of a created temporary table is stored in the DB2 catalog and the description is shareable across application processes while the description of a declared temporary table is neither stored nor shareable. Thus, each application process might refer to the same declared temporary table but have its own unique description of it. For a complete comparison of the two types of temporary tables, including how they differ from base tables, see Section 2 of *DB2 Administration Guide*.

result table A set of rows that DB2 selects or generates from one or more base tables.

empty table A table with zero rows.

sample table One of several tables sent with the DB2 licensed program that contains sample data. Many examples in this book are based on sample tables. See Appendix A of *DB2 Application Programming and SQL Guide* for a description of the sample tables.

Indexes

An *index* is an ordered set of pointers to rows of a base table or an auxiliary table. Each index is based on the values of data in one or more columns. An index is an object that is separate from the data in the table. When you define an index using the CREATE INDEX statement, DB2 builds this structure and maintains it automatically.

Indexes can be used by DB2 to improve performance and ensure uniqueness. In most cases, access to data is faster with an index. A table with a unique index cannot have rows with identical keys. For more details on designing indexes and on their uses, see Section 2 (Volume 1) of *DB2 Administration Guide*.

Keys

A *key* is one or more columns that are identified as such in the description of a table, an index, or a referential constraint. Referential constraints are described in “Referential integrity” on page 23. The same column can be part of more than one key. A key composed of more than one column is called a composite key.

A *composite key* is an ordered set of columns of the same table. The ordering of the columns is not constrained by their ordering within the table. The term *value*, when used with respect to a composite key, denotes a *composite value*. Thus, a rule, such as “the value of the foreign key must be equal to the value of the parent key,” means that each component of the value of the foreign key must be equal to the corresponding component of the value of the parent key.

Unique keys

A *unique key* is a key that is constrained so that no two of its values are equal. DB2 enforces the constraint during the execution of the LOAD utility and the SQL INSERT and UPDATE statements. The mechanism used to enforce the constraint is a *unique index*. Thus, every unique key is a key of a unique index. Such an index is also said to have the UNIQUE attribute. A unique key can be defined using the UNIQUE clause of the CREATE TABLE statement. A table can have an arbitrary number of unique keys.

Primary keys

A *primary key* is a unique key that is a part of the definition of a table. A table can have only one primary key, and the columns of a primary key cannot contain null values. 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, the table is marked unavailable until the primary index is created by the user unless the CREATE TABLE statement is processed by the schema processor. In that case, DB2 automatically creates the primary index.

When a primary key is defined in an ALTER TABLE statement, a unique index must already exist on the columns of that primary key. This unique index is designated as the primary index.

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 using the CREATE or ALTER statement. A foreign key refers to or is related to a specific parent key. 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.

Referential integrity

Referential integrity is the state in which all values of all foreign keys at a given DB2 are valid. 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.

Referential constraints are optional and can be defined using SQL CREATE TABLE and ALTER TABLE statements. Refer to Section 2 (Volume 1) of *DB2 Administration Guide* for examples.

DB2 enforces referential constraints when:

- An INSERT statement is applied to a dependent table.
- An UPDATE statement is applied to a foreign key of a dependent table.
- An UPDATE statement is applied 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 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, 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 <i>self-referencing constraint</i> .

The rules of referential integrity are:

insert rule	A nonnull insert value of the foreign key must match some value of the parent key of the parent table.
update rule	A nonnull update value of the foreign key must match some value of the parent key of the parent table.
delete rule	The choices 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.

The delete rule of a referential constraint applies when a row of the parent table is deleted. More precisely, the 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. Let P denote the parent table, let D denote the dependent table, and let p denote a parent row that is the object of a delete or propagated delete operation. If the delete rule is:

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

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

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

- If D is a dependent of P and the delete rule is RESTRICT or NO ACTION, D is involved in the operation but is not affected by the operation.
- 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 allowed in one or more columns of every row of a table. Check constraints are optional and can be defined using the SQL statements CREATE TABLE and ALTER TABLE. The definition of a check constraint is a restricted form of a search condition. One of the restrictions is that a column name in a check constraint on table T must identify a column of T. See Section 2 (Volume 1) of *DB2 Administration Guide* for examples.

A table can have an arbitrary number of check constraints. DB2 enforces the constraints when:

- A row is inserted into the table.
- A row of the table is updated.
- The LOAD utility with the ENFORCE CONSTRAINTS option is used to populate the table.

A check constraint is enforced by applying its search condition to each row that is inserted, updated, or loaded. An error occurs if the result of the search condition is **false** for any row.

Triggers

A *trigger* defines a set of actions that are executed when a delete, insert, or update operation occurs on a specified table. When such an SQL operation is executed, the trigger is said to be *activated*.

Triggers can be used along with referential constraints and check constraints to enforce data integrity rules. Triggers are more powerful than constraints because they can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions that perform operations both inside and outside of DB2. For example, instead of preventing an update to a column if the new value exceeds a certain amount, a trigger can substitute a valid value and send a notice to an administrator about the invalid update.

Triggers are useful for defining and enforcing business rules that involve different states of the data, for example, limiting a salary increase to 10%. Such a limit requires comparing the value of a salary before and after an increase. For rules that do not involve more than one state of the data, consider using referential and check constraints.

Triggers also move the application logic that is required to enforce business rules into the database, which can result in faster application development and easier maintenance. With the logic in the database, for example, the previously mentioned limit on increases to the salary column of a table, DB2 checks the validity of the changes that any application makes to the salary column. In addition, the application programs do not need to be changed when the logic changes.

Triggers are optional and are defined using the CREATE TRIGGER statement.

For information on using triggers, see Section 3 of *DB2 Application Programming and SQL Guide*.

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*. A storage structure can be one of the following:

- table space** A table space can hold one or more base tables, or one auxiliary table. All tables are kept in table spaces. A table space can be defined using the CREATE TABLESPACE statement.
- index space** An index space contains a single index. An index space is defined when the index is defined using the CREATE INDEX statement.

Storage groups

Defining and deleting the data sets of a storage structure can be left to DB2. If it is left to DB2, the storage structure has an associated *storage group*. The storage group is a list of DASD volumes on which DB2 can allocate data sets for associated storage structures. The association between a storage structure and its storage group is explicitly or implicitly defined by the statement that created the storage structure.

Alternatively, Storage Management Subsystem (SMS) can be used to manage DB2 data sets. Refer to Section 2 (Volume 1) of *DB2 Administration Guide* for more information.

Databases

In DB2, a *database* is a set of table spaces and index spaces. These index spaces contain indexes on the tables in the table spaces of the same database. Databases are defined using the CREATE DATABASE statement and are primarily used for administration. Whenever a table space is created, it is explicitly or implicitly assigned to an existing database.

Catalog

Each DB2 maintains a set of tables that contain information about the data under its control. These tables are collectively known as the *catalog*. The *catalog tables* contain information about DB2 objects such as tables, views, and indexes. In this book, “catalog” refers to a DB2 catalog unless otherwise indicated. In contrast, the

catalogs maintained by access method services are known as “integrated catalog facility catalogs.”

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 system. Each DB2 ensures that the catalog contains accurate descriptions of the objects that the DB2 controls.

Views

A view provides an alternative way of looking at the data in one or more tables. A *view* is a named specification of a result table. The specification is an SQL SELECT statement that is effectively executed whenever the view is referenced in an SQL statement. At any time, the view consists of the rows that would result if the subselect were executed. Thus, a view can be thought of as having columns and rows just like a base table. However, columns added to the base tables after the view is defined do not appear in the view. For retrieval, all views can be used like base tables. Whether a view can be used in an insert, update, or delete operation depends on its definition, as described in “CREATE VIEW” on page 627.

Views can be used to control access to a table and make data easier to use. Access to a view can be granted without granting access to the table. The view can be defined to show only portions of data in the table. A view can show summary data for a given table, combine two or more tables in meaningful ways, or show only the selected rows that are pertinent to the process using the view.

Example: The following SQL statement defines a view named XYZ. The view represents a table whose columns are named EMPLOYEE and WHEN_HIRED. The data in the table comes from the columns EMPNO and HIREDATE of the sample table DSN8610.EMP. The rows from which the data is taken are for employees in departments A00 and D11.

```
CREATE VIEW XYZ (EMPLOYEE, WHEN_HIRED)
AS SELECT EMPNO, HIREDATE
FROM DSN8610.EMP
WHERE WORKDEPT IN ('A00', 'D11');
```

An index cannot be created for a view. However, an index created for a table on which a view is based might improve the performance of operations on the view. The column of a view inherits its attributes (such as data type, precision, and scale) from the table or view column, constant, function, or expression from which it is derived. In addition, a view column that maps back to a base table column inherits any default values or constraints specified for that column of the base table. For example, if a view includes a foreign key of its base table, insert and update operations using that 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 using that view are subject to the same rules as delete operations on the base table. See the description of “INSERT” on page 742 and “UPDATE” on page 833 for restrictions that apply to views with derived columns. For information on referential constraints, see “Referential integrity” on page 23.

Read-only views cannot be used for insert, update, and delete operations. For a discussion of read-only views, see “CREATE VIEW” on page 627.

The definition of a view is stored in the DB2 catalog. An SQL DROP VIEW statement can drop a view, and the definition of the view is removed from the catalog. The definition of a view is also removed from the catalog when any view or base table on which the view depends is dropped.

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 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 space¹. *Locking* is used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously. See Section 5 (Volume 2) of *DB2 Administration Guide* for more information about DB2 locks.

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 or CICS environment. Refer to Section 5 of *DB2 Application Programming and SQL Guide* for more details.

¹ See the description of a table space under “Storage structures” on page 26. Concurrent execution of SQL statements and utilities is discussed in Section 5 (Volume 2) of *DB2 Administration Guide*.

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.

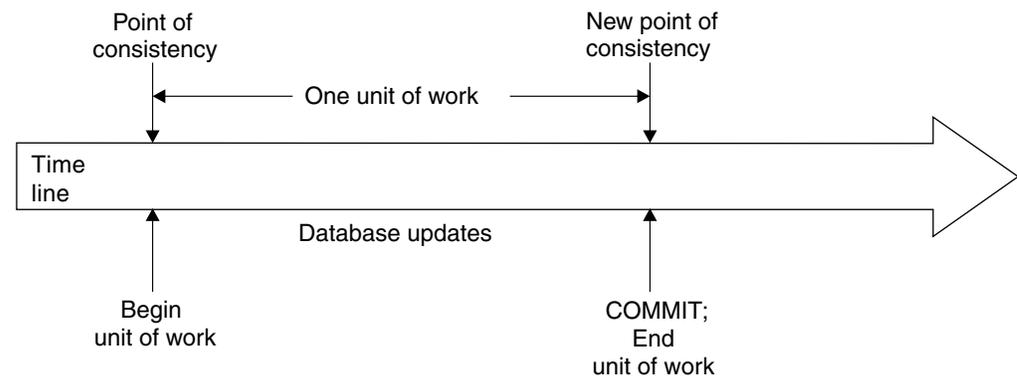


Figure 1. Unit of work with a commit operation

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
 # it assumes existed when the unit of work was initiated. That is, DB2 *undoes* the
 # work, as shown in the diagram below:

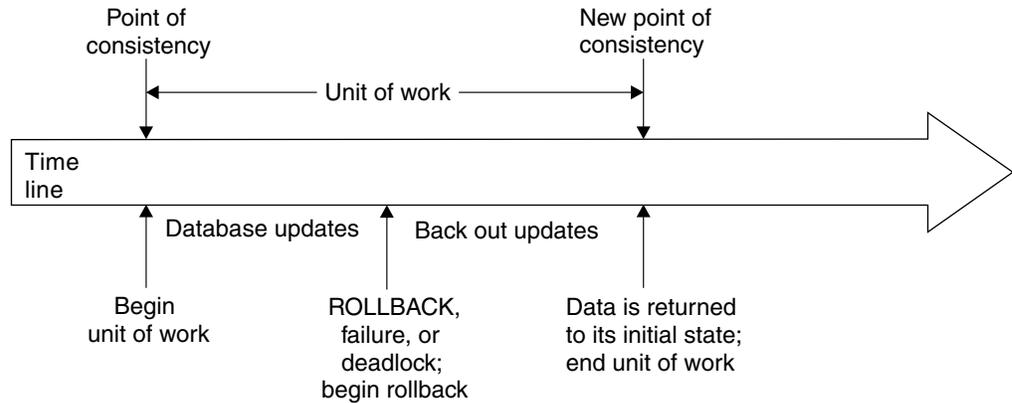


Figure 2. Rolling back all changes from a unit of work

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.

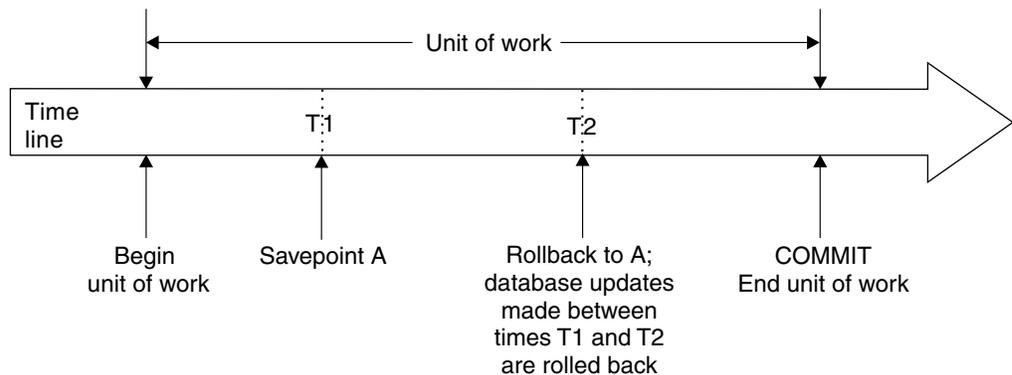


Figure 3. Rolling back changes to a savepoint within a unit of work

Packages and application plans

A *package* contains control structures used to execute SQL statements. Packages are produced during program preparation. The control structures can be thought of as the bound or operational form of SQL statements taken from a *database request module (DBRM)*. The DBRM contains SQL statements extracted from the source program during program preparation. All control structures in a package are derived from the SQL statements embedded in a single source program.

An *application plan* relates an application process to a local instance of DB2, specifies processing options, and contains one or both of the following *elements*:

- A list of package names
- The bound form of SQL statements taken from one or more DBRMs

Every DB2 application requires an application plan. Plans and packages are created using the DB2 subcommands BIND PLAN and BIND PACKAGE, respectively, as described in *DB2 Command Reference*. See Section 6 of *DB2 Application Programming and SQL Guide* for a description of program preparation and identifying packages at run time. Refer to “SET CURRENT PACKAGESET” on page 817 for rules regarding the selection of a plan element.

Distributed data

A DB2 application program can use SQL to access data at other database
 # management systems (DBMSs) other than the DB2 at which the application's plan
 # is bound. This DB2 is known as the *local DB2*. The local DB2 and the other
 # DBMSs are called *application servers*. Any application server other than the local
 # DB2 is considered a *remote server*, and access to its data is a distributed
 # operation. The recommended method of accessing data at remote application
 # servers is “DRDA access” on page 32. “DB2 private protocol access” on page 33
 # is also available but is not recommended.

For application servers that support the two-phase commit process, both methods allow for updating data at several remote locations within the same unit of work. To obtain the more restrictive level of function available at DB2 Version 2 Release 3, refer to 33. Table 1 summarizes the main differences between DRDA access and DB2 private protocol access.

Table 1 (Page 1 of 2). Differences between DRDA access and DB2 private protocol access

Item	DRDA access	DB2 private protocol access
Program preparation	Requires a remote BIND of packages	A remote BIND is not applicable
Plan members	Can use in packages only	Can use in packages or DBRMs bound directly to the plan
Processing of embedded statements	Processed as static SQL	Processed as deferred embedded SQL. For a definition, see “Deferred embedded SQL” on page 19.
Servers	Can use any server that uses the DRDA protocols	Can use DB2 for OS/390 servers only

Table 1 (Page 2 of 2). Differences between DRDA access and DB2 private protocol access

Item	DRDA access	DB2 private protocol access
SQL statements	Can use most SQL statements supported by the system that executes the statement (for details, see <i>DB2 Application Programming and SQL Guide</i>)	Limited to SQL INSERT, UPDATE, and DELETE statements, and to statements supporting SELECT
Connection management	Three-part names and aliases can be used to refer to objects at another server if the package was bound with bind option DBPROTOCOL(DRDA) implicitly or explicitly specified. Otherwise, the CONNECT statement is used to connect an application process to a server.	Three-part names and aliases are used to refer to objects at another server.

Common restrictions: IMS and CICS applications are restricted to read-only operations at a remote site if:

- Its application server does not support two-phase commit.
- It uses DB2 private protocol access to a DB2 Version 2 Release 3. (DB2 private protocol access from a DB2 Version 3 or subsequent release application requester to a DB2 Version 2 Release 2 application server is not supported).

See Section 5 of *DB2 Application Programming and SQL Guide* for more details about common restrictions.

DRDA access

DRDA access supports the execution of dynamic SQL statements and SQL statements that satisfy all the following conditions:

- The static statements appear in a package bound to an accessible server.
- The statements are executed using that package.
- The objects involved in the execution of the statements are at the server where the package is bound. If the server is a DB2 subsystem, three-part names and aliases can be used to refer to another DB2 server.

DRDA access can be used in application programs by coding explicit CONNECT statements or by coding three-part names and specifying the DBPROTOCOL(DRDA) bind option.

DRDA access is based on a set of protocols known as *Distributed Relational Database Architecture* (DRDA). (These protocols are documented by the Open Group Technical Standard in *DRDA Volume 1: Distributed Relational Database Architecture (DRDA)*.) DRDA communication conventions are invisible to DB2 applications, and allow a DB2 to bind and rebind packages at other servers and to execute the statements in those packages. See Section 6 of *DB2 Application Programming and SQL Guide* for the steps involved in binding packages and plans. If the application server supports the two-phase commit process, use the

CONNECT (Type 2) statement and other connection management statements such as RELEASE.

A system that uses DRDA can request the execution of SQL statements at any DB2. Preparing DB2 for incoming SQL requests is discussed in Section 3 of *DB2 Installation Guide*.

When preparing a program for use at a server other than DB2, observe the following rules:

- For SQL statements processed by the server, use the SQL syntax and semantic rules of that server. For other statements, use the DB2 rules. For a list of where statements are processed, see Appendix B, “Characteristics of SQL statements in DB2 for OS/390” on page 873.
- Use the precompiler option SQL(ALL) when precompiling the program. Statements that violate DB2 rules are flagged, but their detection does not prevent the creation of a DBRM.

For more information, refer to the *Distributed Relational Database Library*.

Remote unit of work is a restricted level of function that is available by DRDA access when the CONNECT(1) precompiler option is specified. An application process can have only one connection at a time and cannot connect to a new application server until it executes a commit or rollback operation. This restricts the situations in which the CONNECT statement can be executed. See “CONNECT” on page 446 for more information about these restrictions. For more details about CONNECT (Type 1) and a description of the connection states, refer to “CONNECT (Type 1)” on page 449.

DB2 private protocol access

DB2 private protocol access allows one DB2 to execute a range of statements at another DB2.

A statement is executed using DB2 private protocol access if it refers to objects that are not at the current server and is implicitly or explicitly bound with DBPROTOCOL(PRIVATE). The *current server* is the DBMS to which an application is actively connected. DB2 private protocol access uses *DB2 private connections*. The statements that can be executed are SQL INSERT, UPDATE, and DELETE, and SELECT statements with their associated SQL OPEN, FETCH, and CLOSE statements. “When an application process has a current server” on page 447 describes what happens when an application process has a current server.

In a program running under DB2, a *three-part name* or an *alias* can refer to a table or view at another DB2. The location name identifies the other DB2 to the DB2 application server. A three-part name has the form:

location-name.aaaaa.sssss

where *aaaaa.sssss* uniquely identifies the object at the server named *location-name*. For example, the name USIBMSTODB21.DSN8610.EMP refers to a table named DSN8610.EMP at the server whose location name is USIBMSTODB21. Location naming conventions are described in “Location identifiers” on page 50. Preparing DB2 for incoming SQL requests is discussed in Section 3 of *DB2 Installation Guide*.

Alias names have the same allowable forms as table or view names. The name can refer to a table or view at the current server or to a table or view elsewhere. For more on aliases, see “Aliases and synonyms” on page 58. For more on three-part names, and on SQL naming conventions in general, see “Naming conventions” on page 50.

DRDA access has some significant advantages over DB2 private protocol access:

- # • DRDA access uses a more compact format for sending data over the network and thus improves performance on slow network links.
- # • Queries sent by DB2 private protocol access are bound at the server whenever they are first executed in a unit of work. Repeated binds can reduce the performance of a query that is executed often.

A DBRM for statements executed by DRDA access is bound to a package at the server once. Those statements can include PREPARE and EXECUTE so that your application can accept dynamic statements to be executed at the server. But binding the package is an extra step in program preparation.

- # • You can use stored procedures with DRDA access.

While a stored procedure is running, it requires no message traffic over the network and thus reduces the biggest hindrance to high performance for distributed data.

Connection management for DRDA access and DB2 private protocol

An *SQL connection* is an association between an application process and a local or remote application server. SQL connections can be managed by the application or by using bind options. At any time:

- An application process is in the *connected* or *unconnected* state and has a set of zero or more SQL connections. Each SQL connection of an application process is uniquely identified by the name of the application server of the SQL connection.
- An SQL connection is in one of the following states:
 - Current and held
 - Current and release pending
 - Dormant and held
 - Dormant and release pending

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

The following diagram shows the state transitions:

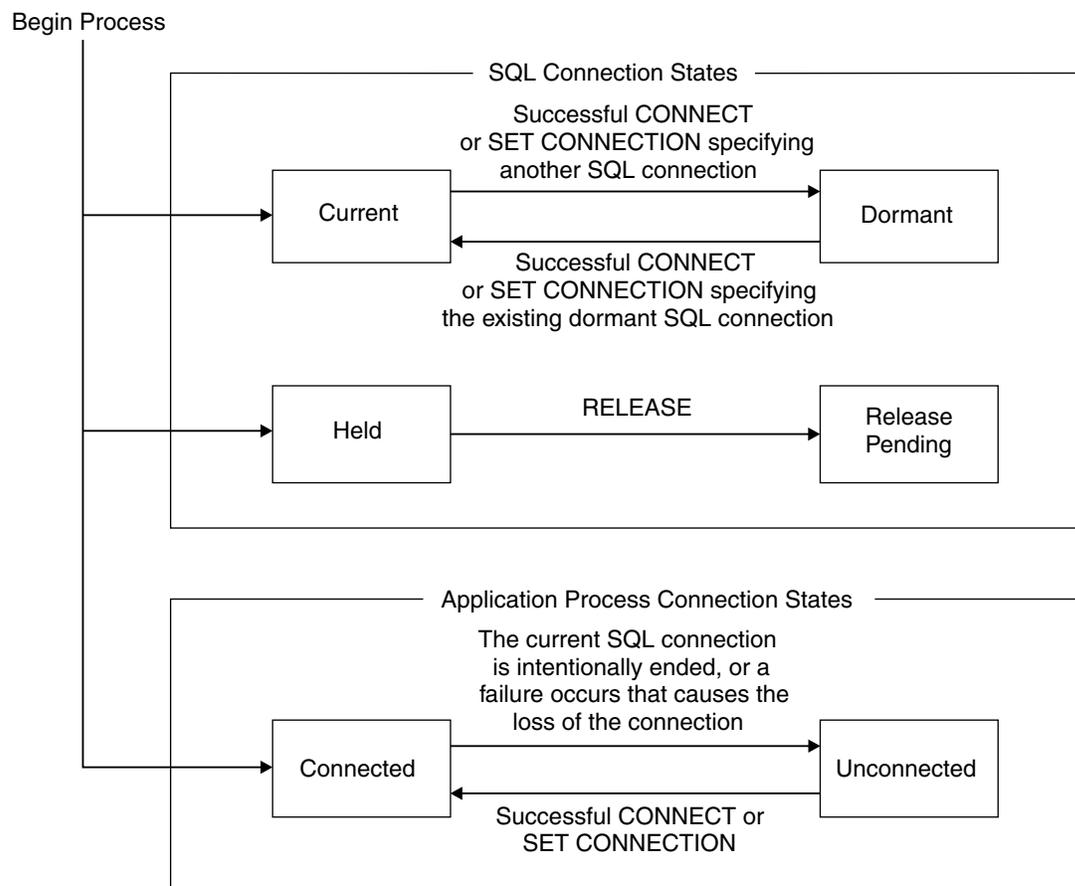


Figure 4. SQL connection and application process connection state transitions

SQL connection states

If an application process executes a `CONNECT TO` statement and the specified location is known to the local DB2 and is not in the set of existing connections of the application process, the location is added to the set of connections and the connection is placed in the current and held state. If the specified location is the current SQL connection of the application process, and if the `SQLRULES(DB2)` bind option is in effect, the states of all existing connections remain the same.

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 previous 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 status.

An SQL connection is placed in the release pending status 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 status can still be used until the next commit operation. Likewise, DB2 private connections in the release pending status can be used until the next commit operation. There is no way to change the state of a connection from release pending to held.

Application process connection states

A different 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 application server at the same time.
- When an application process executes a SET CONNECTION statement, the specified location name must be an existing SQL connection in the set of connections of the application process.
- When an application process executes a CONNECT TO statement and the SQLRULES(STD) bind option is in effect, the specified location must not be an existing SQL connection in the set of connections of the application process.

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

An application process in the unconnected state enters the 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 the *unconnected* state. The CURRENT SERVER special register contains blanks. The only SQL statements that can be executed successfully at the application requester are CONNECT, SET CONNECTION, RELEASE, COMMIT, ROLLBACK, and local SET statements. COMMIT and ROLLBACK are also processed by an application server. If the application process is in the unconnected state, the application server that processes a COMMIT or ROLLBACK is the local DB2.

An application process in the connected state enters the 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 application server and loss of the SQL connection. SQL connections are intentionally ended when an application process successfully executes a commit operation and any of the following 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.

A connect (type 1) statement is implicitly 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 in the *unconnected* state.

DB2 private connections

When the application server is a DB2 subsystem, DB2 private connections are allocated as necessary to support references to objects at other DB2 subsystems. Like SQL connections, DB2 private connections are initially in the held state and can be placed in the release pending status.

An application process cannot have an explicit SQL connection and a DB2 private connection to the same DB2 subsystem at the same time. However, an implicit SQL connection and a DB2 private connection can exist concurrently. Accordingly:

- CONNECT TO x fails if the application process has a DB2 private connection to x, and
- An attempt to allocate a DB2 private connection to x fails if the application process has an explicit SQL connection to x.
- A implicit SQL connection through a three-part name is successful if the application process has a DB2 private connection to x, and
- An attempt to allocate a DB2 private connection to x is successful if the application process has an implicit SQL connection to x.

When a connection is ended

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 deallocated. In the case of an SQL connection to a DB2 subsystem, the resources acquired can include DB2 private connections. When the SQL connection is ended, such DB2 private connections are also ended. This is true even if the DB2 subsystem is the local DB2. For example, assume that an application process implicitly connected to the local DB2 used DB2 private protocol access to open a cursor at another DB2. If the application process executes a RELEASE CURRENT statement, that cursor will be closed when the connection is ended during the next commit operation, unless the cursor has an attribute of WITH HOLD .

A connection can also be ended as a result of a communications failure in which case the application process is placed in the unconnected state. All connections of an application process are ended when the process terminates.

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*.

In client/server environments, character conversion can occur when an SQL statement is executed remotely. Consider, for example, these two cases:

- The values of host variables sent from the application requester to the current server.
- The values of result columns sent from the current server to the application requester.

In either case, the string could have a different representation at the sending and receiving systems. Conversion can also occur during string operations on the same system.

In a local environment, character conversion can occur when:

- An overriding CCSID is specified in the SQLDA (see “SQL descriptor area (SQLDA)” on page 890).

For languages other than REXX, the CCSID is in the SQLNAME field. For
REXX, the CCSID is in the SQLCCSID field.

- A mixed character string is assigned to an SBCS column or host variable.

Most users do not need a knowledge of character conversion. When character conversion does occur, it does so automatically, and the conversion, if successful, is invisible to the application.

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

character set	A defined set of characters. For example, the following character set appears in several code pages: <ul style="list-style-type: none"> • 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. In EBCDIC, for example, 'A' is assigned code point X'C1' and 'B' is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.
code point	A unique bit pattern that represents a character.
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.

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.

encoding scheme

A set of rules used to represent character data. For example:

- Single-byte EBCDIC
- Single-byte ASCII²
- Double-byte EBCDIC

substitution byte

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.

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

“Conversion rules for string assignment” on page 90

“Conversion rules for string comparison” on page 95

“Character conversion in unions and concatenations” on page 328

Character sets and code pages

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

² The term ASCII is used throughout this book to refer to IBM-PC Data or ISO 8 data.

DB2 Concepts

code page: pp1 (ASCII)

	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		Ä	σ
5			%	5	E	U		Ä	ε
E			.	>	N			¼	ö
F			/	*	O			®	

code point: 2F

character set ss1
(in code page pp1)

code page: pp2 (EBCDIC)

	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	¢	;	Á	{	

character set ss1
(in code page pp2)

Even with the same encoding scheme, there are many different coded character sets, and the same code point can represent a different character in different coded character sets. Furthermore, a byte in a character string does not necessarily represent a character from a single-byte character set (SBCS). Character strings are also used for mixed data (that is a mixture of single-byte characters and double-byte characters) and for data that is not associated with any character set (called bit data). Note that this is not the case with graphic strings; every pair of bytes in every graphic string is assumed to represent a character from a double-byte character set (DBCS).

Character encoding for IBM systems is described in *Character Data Representation Architecture Reference and Registry*.

System CCSIDS

Every string used in an SQL operation has a CCSID, and the CCSID identifies the manner in which the characters in the string are encoded. Strings can be encoded in EBCDIC or ASCII. A string representing characters can be one of three types:

- An SBCS string (*single-byte character set*). In an SBCS string, each character is represented by a single byte. SBCS is a subtype of the character data type.
- A graphic string composed of DBCS (*double-byte character set*) characters. In a graphic string, each character is represented by a pair of bytes.
- A mixed string, in which both single-byte and double-byte characters can occur. In an EBCDIC mixed string, certain *shift characters* serve as left- and

right-delimiters for sequences of double-byte characters. MIXED is a subtype of the character data type.

At a given DB2, all columns that contain SBCS strings are assumed to have a common CCSID known as the *corresponding ASCII or EBCDIC system CCSID* for SBCS data. Likewise, all columns that contain graphic strings have a common CCSID, known as the corresponding ASCII or EBCDIC system CCSID for graphic data, and all columns that contain mixed strings have a common CCSID known as the corresponding ASCII or EBCDIC system CCSID for mixed data. For example, DB2 can use a system CCSID when character data is fetched from a table at another DBMS. The system CCSID is used to convert the incoming data to the appropriate CCSID. If the character string has a subtype of BIT, its bytes do not represent characters and are not converted.

The values specified in fields ASCII CODED CHAR SET and EBCDIC CODED CHAR SET on installation panel DSNTIPF when DB2 was installed determine the ASCII and EBCDIC system CCSIDs. Those fields should contain valid SBCS CCSIDs if field MIXED DATA on that same installation panel is NO, or valid MIXED CCSIDs if field MIXED DATA is YES.

Field DEF ENCODING SCHEME on the same installation panel determines whether the default encoding scheme for the DB2 system is ASCII or EBCDIC. For example, one CCSID whose value is 37 identifies a widely used form of EBCDIC encoding. That particular CCSID could be the system CCSID for EBCDIC SBCS strings.

For more information about character string subtypes and SBCS and DBCS DB2 sites, see “Data types” on page 66. For information on the subsystem parameters that determine the default encoding scheme and the system CCSIDs, see *DB2 Installation Guide*.

Restrictions on BIT data

If the CCSID of an input host variable or a host variable substituted for a parameter marker is different from the CCSID determined at bind time, and if either CCSID is X'FFFF' (BIT data), an error occurs. Otherwise, the host variable is converted to the coded character set determined by the CCSID at bind time.

Expanding conversions

An *expanding conversion* occurs when the length of the converted string is greater than that of the source string. An expanding conversion occurs when an ASCII mixed data string that contains DBCS characters is converted to EBCDIC mixed data. Because of the addition of shift codes, an error occurs when an expanding conversion is performed on a fixed-length input host variable that requires conversion from ASCII mixed to EBCDIC mixed. The remedy is to use a varying-length string variable with a maximum length that is sufficient to contain the expansion.

Contracting conversions

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

Chapter 3. Language elements

Characters	47
Tokens	47
Identifiers	48
SQL identifiers	48
Location identifiers	50
Host identifiers	50
Naming conventions	50
Qualification of unqualified object names	56
Schemas and the SQL path	57
Aliases and synonyms	58
Authorization IDs and authorization-names	59
Authorization IDs and schema names	60
Authorization IDs and statement preparation	60
Authorization IDs and dynamic SQL	61
Authorization IDs and remote execution	63
Data types	66
Character strings	67
Graphic strings	70
Binary strings	71
Large objects (LOBs)	71
Restrictions using long strings	72
Numbers	73
Datetime values	75
Row ID values	78
Distinct types	79
Promotion of data types	81
Casting between data types	83
Assignment and comparison	84
Numeric assignments	86
String assignments	89
Datetime assignments	91
Row ID assignments	92
Distinct type assignments	92
Numeric comparisons	94
String comparisons	94
Datetime comparisons	96
Row ID comparisons	96
Distinct type comparisons	96
Rules for result data types	99
Character string operands	99
Graphic string operands	99
Binary string operands	100
Numeric operands	100
Datetime operands	100
Row ID operands	101
Distinct type operands	101
Nullable attribute of a result	101
Constants	101
Integer constants	101
Floating-point constants	102

Language Elements

Decimal constants	102
Character string constants	102
Datetime constants	103
Graphic string constants	103
Special registers	104
General rules for special registers	104
CURRENT DATE	106
CURRENT DEGREE	106
CURRENT LOCALE LC_CTYPE	107
CURRENT OPTIMIZATION HINT	107
CURRENT PACKAGESET	108
CURRENT PATH	108
CURRENT PRECISION	109
CURRENT RULES	109
CURRENT SERVER	111
CURRENT SQLID	111
CURRENT TIME	112
CURRENT TIMESTAMP	112
CURRENT TIMEZONE	112
USER	113
Column names	114
Qualified column names	114
Correlation names	114
Column name qualifiers to avoid ambiguity	115
Column name qualifiers in correlated references	116
Resolution of column name qualifiers and column names	117
Referencing host variables	120
Host variables in dynamic SQL	121
References to LOB host variables	121
References to LOB locator variables	122
References to stored procedure result sets	122
References to result set locator variables	123
Host structures in PL/I, C, and COBOL	123
Functions	125
Types of functions	125
Function resolution	127
Expressions	131
Without operators	131
With the concatenation operator	131
With arithmetic operators	133
Arithmetic with two integer operands	134
Arithmetic with an integer and a decimal operand	134
Arithmetic with two decimal operands	134
Arithmetic with floating-point operands	137
Datetime operands and durations	137
Datetime arithmetic in SQL	138
Precedence of operations	143
CASE expressions	143
CAST specification	146
Predicates	150
Basic predicate	150
Quantified predicate	151
BETWEEN predicate	153
EXISTS predicate	153

IN predicate	155
LIKE predicate	156
NULL predicate	161
Search conditions	163
Options affecting SQL	164
Precompiler options for dynamic statements	166
Decimal point representation	166
Apostrophes and quotation marks in string delimiters	168
Katakana characters for EBCDIC	169
Mixed data in character strings	169
Formatting of datetime strings	170
SQL standard language	170
Positioned updates of columns	172

This chapter defines the basic syntax of SQL and language elements that are common to many SQL statements.

Characters

The basic symbols of SQL are characters from the EBCDIC syntactic character set. These *characters* are classified as letters, digits, or special characters:

- A *letter* is any one of the uppercase alphabetic characters A through Z plus the three EBCDIC code points reserved as alphabetic extenders for national languages (the code points X'5B', X'7B', and X'7C', which display as \$, #, and @ using code pages 37 and 500).
- A *digit* is any one of the characters 0 through 9.
- A *special character* is any character other than a letter or a digit.

SQL statements can also contain *double-byte character set (DBCS)* characters. Regardless of the value of the field MIXED DATA on installation panel DSNTIPF, double-byte characters can be used in SQL ordinary identifiers and graphic string constants when enclosed by the necessary shift characters. If the value of MIXED DATA is YES, double-byte characters can also be used in string constants and delimited identifiers. In SQL application programs, any use of double-byte characters must be contained within a single line. Thus, a graphic string constant cannot be continued from one line to the next and, if MIXED DATA is YES, a character string constant and delimited identifier can be continued from one line to the next only if the break occurs between single-byte characters. This restriction also applies to the use of double-byte characters within tokens of the host language.

Tokens

The basic syntactical units of the 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 757.

Examples:

, 'string' "fld1" = .

String constants and certain delimited identifiers are the only tokens that can include a space or control character. Any token can be followed by a space or control character. Every ordinary token must be followed by a delimiter token, a

Identifiers

space, or a control character; if the syntax does not allow a delimiter token, a space or a control character must follow the ordinary token.

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. DB2 handles the following control characters:

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

Uppercase and lowercase: Any token can include lowercase letters, but a lowercase letter in an ordinary token is folded to uppercase unless the SQL statement is embedded in a C program. Delimiter tokens are never folded to uppercase.

Example: The statement:

```
select * from DSN8610.EMP where lastname = 'Smith';
```

is equivalent, after folding, to:

```
SELECT * FROM DSN8610.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, a location identifier, or a host identifier. See Appendix A, "Limits in DB2 for OS/390" on page 869 for the identifier length limits that DB2 imposes.

SQL identifiers

SQL identifiers can be *ordinary identifiers* or *delimited identifiers*. They can also be *short identifiers* or *long identifiers*. Thus, an SQL identifier can be in one of four categories: short ordinary, long ordinary, short delimited, or long delimited.

Ordinary identifiers

An *ordinary identifier* is a letter followed by zero or more characters, each of which is a letter, a digit, or the underscore character. An ordinary identifier with an EBCDIC encoding scheme can include Katakana characters if the value of field EBCDIC CODED CHAR SET on installation panel DSNTIPF is set to 930 or 5026 when the statement is parsed.

DBCS characters are allowed in SQL ordinary identifiers. An SQL ordinary identifier, when used as the name of a table, column, alias, synonym, view, statement, cursor, correlation, distinct type, stored procedure, user-defined function, or trigger name can be specified using either DBCS characters or *single-byte character set* (SBCS) characters. However, an SQL ordinary identifier cannot contain a mixture of SBCS and DBCS characters.

The following list shows the rules for forming DBCS SQL ordinary identifiers. These are EBCDIC rules because DB2 processes SQL statements in EBCDIC.

- The identifier must start with a shift-out (X'0E'), end with a shift-in (X'0F'), and an odd-numbered byte between those shifts must not be a shift-out.
- The maximum length is 18 bytes including the shift-out and the shift-in. In other words, there is a maximum of 16 bytes (8 double-byte characters) between the shift-out and the shift-in.
- There must be an even number of bytes between the shift-out and the shift-in.
- DBCS blanks (X'4040') are not acceptable between the shift-out and the shift-in.
- The identifiers are not folded to uppercase or changed in any other way.
- Continuation to the next line is not allowed.

An ordinary identifier must not be identical to a keyword that is a reserved word in any context in which the identifier is used. For a list of reserved words, see Appendix E, “SQL reserved words” on page 1027.

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 (") 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 bind option values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.
 - DYNAMICRULES bind, invoke, or define 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.

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

“SYNONYM”

Naming Conventions

Short and long identifiers

SQL identifiers are also classified according to their maximum length. A *long identifier* has a maximum length of 18 bytes. A *short identifier* has a maximum length of 8 bytes. These limits do not include the escape characters of a delimited identifier.

Whether an identifier is long or short depends on what it represents. For example, the name of a storage group is a short identifier, whereas an unqualified table name is a long identifier. “Naming conventions” describes what identifiers can represent and whether those representing a given type of entity are long or short.

Database names and table space names are examples of short identifiers that will be used as part of data set names. Such identifiers, whether ordinary or delimited, must conform to the MVS rules for forming data set names. For example, a short ordinary identifier used to name a database must not contain an underscore character.

Location identifiers

A location identifier is like an SQL identifier, except as follows:

- The maximum length is 16 bytes.
- The ordinary form must 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.

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.

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. 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`.

An alias can refer to a table or view that is at the current server or a remote server, and 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.

Statements that use three-part names and refer to distributed data result in either DB2 private protocol access or DRDA access to the remote site. 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 PROTOCOL 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.

authorization-name	A short identifier that designates a set of privileges. It can also designate a user or group of users, but DB2 does not control this property. See “Authorization IDs and authorization-names” on page 59 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 <i>table-name</i> . See <i>table-name</i> on page 54.								
bpname	A name that identifies a buffer pool. The following list shows the names of the different buffer pool sizes. <table> <tr> <td>4KB</td> <td>BP0, BP1, BP2, ..., BP49</td> </tr> <tr> <td>8KB</td> <td>BP8K0, BP8K1, BP8K2, ..., BP8K9</td> </tr> <tr> <td>16KB</td> <td>BP16K0, BP16K1, BP16K2, ..., BP16K9</td> </tr> <tr> <td>32KB</td> <td>BP32K, BP32K1, BP32K2, ..., BP32K9</td> </tr> </table>	4KB	BP0, BP1, BP2, ..., BP49	8KB	BP8K0, BP8K1, BP8K2, ..., BP8K9	16KB	BP16K0, BP16K1, BP16K2, ..., BP16K9	32KB	BP32K, BP32K1, BP32K2, ..., BP32K9
4KB	BP0, BP1, BP2, ..., BP49								
8KB	BP8K0, BP8K1, BP8K2, ..., BP8K9								
16KB	BP16K0, BP16K1, BP16K2, ..., BP16K9								
32KB	BP32K, BP32K1, BP32K2, ..., BP32K9								
built-in-data-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 56.								
catalog-name	A short identifier that designates an integrated catalog facility catalog.								
collection-id	A long identifier that identifies a collection of packages; therefore, a collection ID is a qualifier for a package ID. Refer to Chapter 1 of <i>DB2 Command Reference</i> 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								

Naming Conventions

and a long identifier. The qualifier is a table name, a view name, a synonym, an alias, or a correlation name.

An unqualified column name is a long identifier.

constraint-name	A short identifier that designates a referential constraint on a table, or a long identifier that designates a check constraint on a table.
correlation-name	A long identifier that designates a table, a view, or individual rows of a table or view.
cursor-name	A long identifier that designates an SQL cursor.
database-name	A short 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 “Referencing host variables” on page 120 for a description of a host identifier. A descriptor name never includes an indicator variable.
distinct-type-name	<p>A qualified or unqualified name that designates a distinct type.</p> <p>A qualified distinct type name is a two-part name. The first part is a short identifier. The short identifier is the schema name of the distinct type. The second part is a long identifier. A period must separate each of the parts.</p> <p>An unqualified distinct type name is a long 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 56.</p>
function-name	<p>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.</p> <p>A qualified function name is a two-part name. The first part is a short identifier. The short identifier is the schema name of the function. The second part is a long identifier. A period must separate each of the parts.</p> <p>An unqualified function name is a long identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the context in which the function appears as described by the rules in “Qualification of unqualified object names” on page 56.</p>
host-variable	A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, as explained in “Referencing host variables” on page 120.
index-name	<p>A qualified or unqualified name that designates an index.</p> <p>A qualified index name is a short identifier followed by a period and a long identifier. The short identifier is the authorization ID that owns the index.</p>

# #	An unqualified index name is a long identifier with an implicit qualifier. The implicit qualifier is an authorization ID that is determined by the rules set forth in “Qualification of unqualified object names” on page 56.
	For an index on a declared temporary table, the qualifier must be SESSION.
location-name	A location identifier that identifies an instance of a database management system.
package-id	A short identifier that identifies a package. For packages created using DB2, a package ID is the name of the program whose precompilation produced the package's DBRM. Refer to Chapter 1 of <i>DB2 Command Reference</i> for naming conventions.
plan-name	A short identifier that identifies an application plan. Refer to Chapter 1 of <i>DB2 Command Reference</i> for naming conventions.
procedure-name	<p>A qualified or unqualified name that designates a stored procedure.</p> <p>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 a long identifier. A period must separate each of the parts.</p> <p>A two-part procedure name is implicitly qualified with the location name of the current server. The first part is the schema name of stored procedure. The second part is a long identifier. A period must separate the two parts.</p> <p>A one-part or unqualified procedure name is a long identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second implicit qualifier depends on the application server. If the server is DB2 for OS/390, 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 56.</p>
program-name	A short identifier that designates an exit routine.
schema-name	A short identifier that designates a schema. A <i>schema-name</i> that is used as a qualifier of the name of an object is often also an authorization ID. The objects that are qualified with a schema name are distinct types, stored procedures, triggers, and user-defined functions. Built-in data types and built-in functions are also qualified with a schema name.
specific-name	<p>A qualified or unqualified name that designates a unique name for a user-defined function.</p> <p>A qualified specific name is a two-part name. The first part is a short identifier. The short identifier is the schema</p>

The second part is a long identifier. A period must separate the two parts.

A one-part or unqualified table name is a long identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second is an authorization ID, which is determined by the rules set forth in “Qualification of unqualified object names” on page 56.

#

For a declared temporary table, the qualifier that designates the owner (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 639.

table-space-name

A short 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 a short identifier. The short identifier is the schema name of the trigger. The second part is also a short identifier. A period must separate each of the parts.

An unqualified trigger name is a short identifier with an implicit qualifier. The implicit qualifier is the schema name, which is determined by the rules set forth in “Qualification of unqualified object names” on page 56.

version-id

An identifier³ of 1 to 64 characters that is assigned to a package when the package is created. The version ID that is assigned is taken from the version ID associated with the program being bound. Version IDs are specified for programs as a parameter of the DB2 precompiler. Refer to Chapter 1 of *DB2 Command Reference* for naming conventions.

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 the authorization ID that designates the owner of the view. The third part is a long 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 the authorization ID that designates the owner of the view. The second part is a long identifier. A period must separate the two parts.

³ The *version-id* can begin with a digit, for example, when it is a timestamp.

A one-part or unqualified view name is a long identifier with two implicit qualifiers. The first implicit qualifier is the location name of the current server. The second is an authorization ID, which is determined by the rules set forth in “Qualification of unqualified object names.”

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, table, and view names

Unqualified alias, index, table, and view names are implicitly qualified as follows:

- For static SQL statements, the implicit qualifier is the identifier specified in the QUALIFIER option of the BIND subcommand used to bind the SQL statements. If this bind option was not used when the plan or package was created or last rebound, the implicit qualifier is the authorization ID of the owner of the plan or package.
- For dynamic SQL statements, the behavior as specified by the combination of bind option DYNAMICRULES and the run-time environment determines the implicit qualifier. (For a list of these behaviors and the DYNAMICRULES values that determine them, see Table 2 on page 61).
 - If DYNAMICRULES *run behavior* applies, the implicit qualifier is the SQL authorization ID in the CURRENT SQLID special register. Run behavior is the default.
 - If *bind behavior* applies, the identifier implicitly or explicitly specified in the QUALIFIER option of the BIND subcommand, as explained above for static SQL statements.
 - If *define behavior* applies, the implicit qualifier is the owner of the function or stored procedure (the owner is the definer).
 - If *invoke behavior* applies, the implicit qualifier is the authorization ID of the invoker of the function or stored procedure.

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

Unqualified data type, function, or procedure

The qualification of data type, function, and stored procedure depends on the SQL statement in which the unqualified name appears:

- If an unqualified name is the main object of an ALTER, CREATE, COMMENT ON, 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 used to bind the SQL statements. If this bind option was not used when the plan or package was created or last rebound, the implicit qualifier is the authorization ID of the owner of the plan or package.
 - In a dynamic statement, the implicit schema name is the SQL authorization ID in the CURRENT SQLID special register.

- Otherwise, the implicit schema name for the unqualified name is determined as follows:
 - For data type names, DB2 searches the SQL path and selects the first schema in the path such that the data type exists in the schema and the user has authorization to use the data type.
 - For function names, DB2 uses the SQL path in conjunction with function resolution, as described under “Function resolution” on page 127.
 - For stored procedure names⁴, 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 information on the SQL path, see “Schemas and the SQL path.”

Schemas and the 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, function, and stored procedure names that appear in any context other than as the main object of an ALTER, CREATE, DROP, COMMENT ON, GRANT or REVOKE statement.⁵ Searching through the path from left to right, DB2 implicitly qualifies the object name with the first schema name in the path that contains the same object with the same unqualified name for which the user has appropriate authorization. For procedures, DB2 selects a matching procedure name only if the number of parameters is also the same. 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 can reside in a schema. (For details, see “Function resolution” on page 127.)

For example, if the SQL path is SMITH, XGRAPHIC, SYSIBM and an unqualified distinct type name MYTYPE was specified, DB2 looks for MYTYPE first in schema SMITH, then XGRAPHIC, and then SYSIBM.

The PATH bind 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 a literal (CALL 'literal')

If the PATH bind option was not specified when the plan or package was created or last rebound, its default value 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*)

⁴ In CALL statements only.

⁵ 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.

Aliases and Synonyms

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

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

For additional details on the initial value of CURRENT PATH special register and changing its value, see “CURRENT PATH” on page 108 and “SET CURRENT PATH” on page 819.

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

SYSPROC, SYSIBM, GEORGIA, SMITH

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 alternate names for tables and views.

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. Nevertheless, an alias or a synonym can be used wherever a table or view can be referred to in an SQL statement, with two exceptions: a local alias cannot be used in CREATE ALIAS, and a synonym cannot be used in CREATE SYNONYM. If an alias is used in CREATE SYNONYM, 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 CREATE ALIAS, 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:

- SYSADM or SYSCTRL authority 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 designate an alias defined at another server if that alias represents a table or view at the other server.) When a synonym is used, this error cannot occur.

Authorization IDs and authorization-names

An *authorization ID* is a character string that designates a defined set of privileges. Processes can successfully execute SQL statements only if they have the authority to perform the specified functions. A process derives this authority from its authorization IDs. An authorization ID can also designate a user or a group of users, but DB2 does not control this property.

DB2 uses authorization IDs to provide:

- Authorization checking of SQL statements
- Implicit qualifiers for database objects like tables, views, aliases, and indexes

Whenever a connection is established between DB2 and a process, DB2 obtains an authorization ID and passes it to the authorization exit. The list of one or more authorization IDs returned by the exit are used as the authorization IDs of the process.

Every process has exactly one primary authorization ID. Any other authorization IDs of a process are secondary authorization IDs. As explained below, the use of these authorization IDs depends on whether the process is a bind process or an application process.

An *authorization-name* specified in an SQL statement should not be confused with an authorization ID of a process. For 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 DSN8610.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 SET CURRENT SQLID. Thus, when the GRANT statement is prepared and executed by SPUFI, the SQL authorization ID is SMITH. KEENE is an authorization name specified in the GRANT statement.

Authorization to execute the GRANT statement is checked against SMITH, and SMITH is the implicit qualifier of TDEPT. The authorization rule is that the privilege set designated by SMITH must include the SELECT privilege with the GRANT option on SMITH.TDEPT. There is no check involving KEENE.

Authorization IDs and Authorization-names

If SMITH is the implicit qualifier for a statement that contains NAME1, NAME1 identifies the same object as SMITH.NAME1. If the implicit qualifier is other than SMITH, NAME1 and SMITH.NAME1 identify different objects.

Authorization IDs and schema names

An authorization ID that is the same as the name of a schema implicitly has the CREATEIN, ALTERIN, and DROPIN privileges for that schema.

Authorization IDs and statement preparation

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. The authorization ID specified as owner must be one of the authorization IDs of the process, unless one of these has SYSADM or SYSCTRL authority. In this case, the owner can be set to any value. BINDAGENT can specify an owner other than himself (or one of his secondaries), 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). BIND and REBIND are discussed in Chapter 2 of *DB2 Command Reference*.

The authorization ID used for the authorization checking of embedded SQL statements is that 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 63.

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 634.

Authorization IDs and dynamic SQL

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 63.

Bind option DYNAMICRULES determines the authorization ID that is used for checking authorization when dynamic SQL statements are processed. In addition, the option also controls other dynamic SQL attributes such as the implicit qualifier that is used for unqualified alias, index, table, and view names; the source for application programming options; and whether certain SQL statements can be invoked dynamically.

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. As Table 2 shows, the combination of the value of the DYNAMICRULES bind option and the run-time environment determines which of the behaviors is used. DYNAMICRULES(RUN), which implies run behavior, is the default.

Table 2. How DYNAMICRULES and the run-time environment determine dynamic SQL statement behavior

DYANMICRULES 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
INVOKEBIND	Bind behavior	Invoke behavior

Note: BIND and RUN values can be specified for both packages and plans. The other values can be specified only for packages.

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 ID of the application process and the SQL authorization ID (the value of special register CURRENT SQLID) for authorization checking of dynamic SQL statements.

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 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.

Privilege set: If the dynamically prepared statement is other than an ALTER, CREATE, 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.

If the dynamic SQL statement is an ALTER, CREATE, 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.

Implicit qualification: As explained under “Qualification of unqualified object names” on page 56, when an SQL statement is dynamically prepared, the SQL authorization ID is also used as the implicit qualifier for all unqualified tables, aliases, views, and indexes.

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 primary 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 60.

Privilege set: The privilege set is the privileges that are held by the primary authorization ID of 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 is the implicit qualifier for all unqualified tables, views, aliases, and indexes. If this bind option was not used when the plan or package was created or last rebound, the implicit qualifier is the authorization ID of the owner of the plan or package.

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 stored procedure or user-defined function owner.

Implicit qualification: The authorization ID of the stored procedure or user-defined function owner is also the implicit qualifier for unqualified table, view, alias, and index 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 authorization ID of the stored procedure or user-defined function invoker 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 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 held by each authorization ID.

Implicit qualification: The authorization ID of the stored procedure or user-defined function invoker is also the implicit qualifier for unqualified table, view, alias, and index names.

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, DROP, GRANT, RENAME, and REVOKE.

For more information on authorization and examples of determining authorization for dynamic SQL statements, see Section 3 of *DB2 Administration Guide*. For complete details about the DYNAMICRULES bind option, see *DB2 Command Reference*.

Authorization IDs and remote execution

The authorization rules for remote execution depend on whether the distributed operation is:

- DRDA access with a DB2 for OS/390 server and requester
- DRDA access with a server and requester other than DB2
- DB2 private protocol access

DRDA access with DB2 for OS/390 only

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 that 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 Section 3 (Volume 1) of *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 derived during connection processing. 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 Section 3 (Volume 1) of *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 OS/390, which is described next under “DRDA access with a server or requester other than DB2 for OS/390.”

DRDA access with a server or requester other than DB2 for OS/390

DB2 for OS/390 as the server: If the application requester is not a DB2 for OS/390 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 61.

DB2 for OS/390 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

Three authorization IDs played roles in the foregoing discussion. These are the user's primary authorization ID and those for the owner of the application plan and the owner of a package. Each of these is sent to the remote DBMS. And each may undergo translations before it is used.

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. For how to use and maintain these tables, see Section 3 (Volume 1) of *DB2 Administration Guide*.

Other security measures

The fact that DB2 authority requirements are satisfied does not guarantee that a user has access to a given server. Other security measures may 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 Section 3 (Volume 1) of *DB2 Administration Guide*.

Data 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:

- Constants
- Columns
- Expressions
- Functions
- Host variables
- Special registers

DB2 supports both IBM-supplied data types (built-in data types) and user-defined data types (distinct types). This section describes the built-in data types. For a description of distinct types, see “Distinct types” on page 79.

Figure 5 shows the built-in data types that DB2 supports.

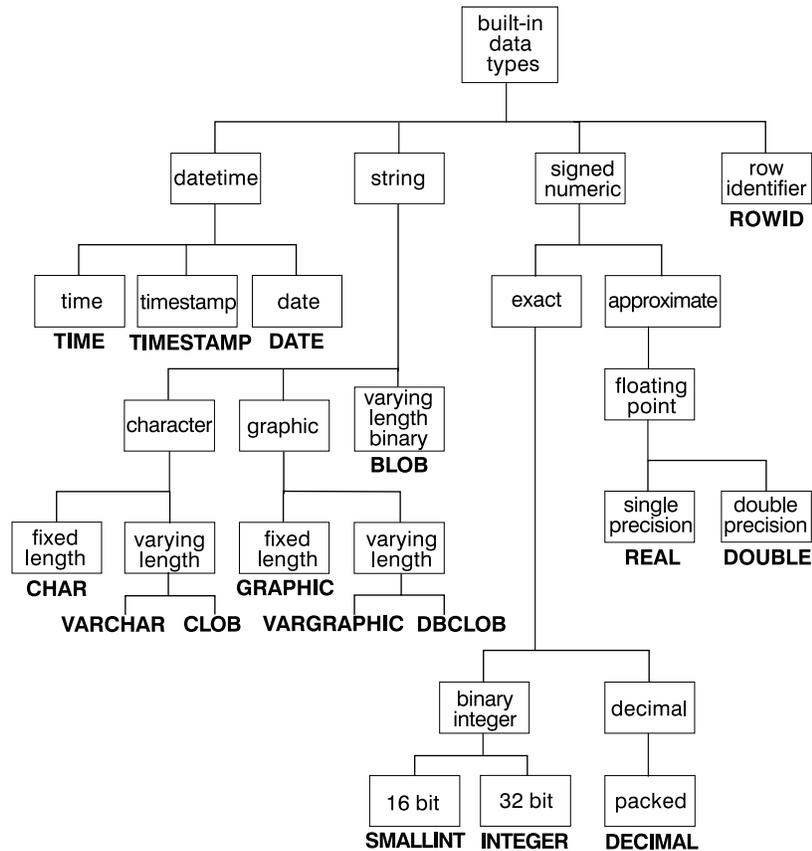


Figure 5. Built-in data types supported by DB2

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.

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.

Except for C NUL-terminated strings, the length of a varying-length string is specified by the value of its length control field. For varying-length character strings, the length control field specifies the number of bytes.

Each character string has a subtype of SBCS, MIXED, or BIT with the exception of character strings with a CLOB data type, which can only have an SBCS or MIXED subtype.

- The bytes of a character string with subtype SBCS represent characters from a single-byte character set (SBCS). Such strings are called SBCS data.
- The bytes of a character string with subtype MIXED can represent a mixture of characters from a single-byte character set (SBCS) and a double-byte character set (DBCS). Strings that may contain both SBCS and DBCS characters are called *mixed data*. EBCDIC mixed data may contain shift characters, which represent neither SBCS nor DBCS data.
- The bytes of a character string with BIT subtype do not represent characters; therefore, character conversion never occurs for these strings. Such strings are called BIT data.

Character subtypes provide a simple and portable way of specifying the CCSID of a character string column. The subtype is implicitly or explicitly specified when the column is defined in a CREATE or ALTER TABLE statement. The default is SBCS or MIXED depending on the value of the field MIXED DATA on installation panel DSNTIPF. The following list shows the CCSID for each subtype:

BIT The CCSID is X'FFFF' (65535).

SBCS The CCSID is the system CCSID for SBCS data.

MIXED The CCSID is the system CCSID for mixed data.

The FOREIGNKEY column of the SYSCOLUMNS catalog table stores information about the subtype of a character string column. An administrator can update this column to change the subtype of existing columns. DB2 does not ensure that the bytes of a character string are consistent with its CCSID and does not use CCSIDs for any purpose other than character conversion.

DBCS characters and ASCII and EBCDIC

The method of representing DBCS characters within a mixed string differs between ASCII and EBCDIC.

- ASCII reserves a set of code points for SBCS characters and another set as the first half of DBCS characters. Upon encountering the first half of a DBCS character, the system knows that it is to read 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 recognizer 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 recognizer enters 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 recognizer returns to SBCS mode. An error occurs if the recognizer is in 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.

Examples

$\overline{\pi}$ gen $\overline{\kappa}$ ki CHAR(9) in ASCII.

$\textcircled{\text{0}}\overline{\pi}$ $\textcircled{\text{1}}$ gen $\textcircled{\text{0}}\overline{\kappa}$ $\textcircled{\text{1}}$ ki CHAR(13) in EBCDIC.

Because of the differences of the representation of mixed data strings in ASCII and EBCDIC, 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 both ASCII and EBCDIC data.

SBCS sites

An SBCS site is a DB2 in which the subtype of character strings is SBCS or BIT. The value of field MIXED DATA on installation panel DSNTIPF is NO. The values of fields ASCII CODED CHAR SET and EBCDIC CODED CHAR SET determine the system CCSID that identifies the SBCS coded character set used at that site. The default subtype is SBCS data.

DBCS sites

A DBCS site is a DB2 in which the subtype of character strings can be SBCS, BIT, or MIXED. The value of field MIXED DATA on installation panel DSNTIPF is YES. The values of fields ASCII CODED CHAR SET and EBCDIC CODED CHAR SET determine the system CCSIDs used for SBCS data, mixed data, and graphic data. The default subtype is mixed data.

A mixed data string can have zero or more sequences of SBCS characters and zero or more sequences of DBCS characters. Each EBCDIC DBCS sequence must be preceded by the shift-out control character (X'0E') and followed by the shift-in control character (X'0F'). There must be an even number of bytes between the shift characters and each pair of bytes is assumed to represent a DBCS character.

DB2 recognizes DBCS sequences within mixed data strings when performing character-sensitive operations at DBCS sites (the field MIXED DATA is YES). These operations include parsing, character conversion, and the pattern matching specified by the LIKE predicate. DB2 also recognizes DBCS sequences:

- In source language statements, static SQL statements, and deferred embedded SQL statements if the GRAPHIC precompiler option is implicitly or explicitly specified
- In dynamic SQL statements if DYNAMICRULES bind, define, or invoke behavior is in effect, the value of installation panel field USE FOR

DYNAMICRULES is NO, and the GRAPHIC precompiler option is implicitly or explicitly specified

Fixed-length character strings

All the values of a column with a fixed-length character string data type have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 255 inclusive. Every fixed-length string column is a *short string column*. A fixed-length character string column can also be called a CHAR or CHARACTER column.

Varying-length character strings

The types of varying-length character strings are:

- VARCHAR (or synonyms CHAR VARYING and CHARACTER VARYING)⁶
- CLOB (or synonyms CHAR LARGE OBJECT and CHARACTER LARGE OBJECT)

The values of a column with any one of these string types can have different lengths. The length attribute of the column determines the maximum length a value can have.

For a VARCHAR column, the length attribute must be between 1 and m inclusive, where m is determined by the maximum record size as described in Maximum record size on page 592 in the description of the CREATE TABLE statement. For a CLOB column, 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 71.

A VARCHAR column with a maximum length that is greater than 255 bytes or a CLOB column of any length is a *long string column*. For the restrictions that apply to the use of long string columns, see “Restrictions using long strings” on page 72.

Character string host variables

Host variables with CHAR and CLOB string types can be defined in all host languages. (In C, CHAR string variables are limited to a length of 1.) Host variables with a VARCHAR string type can be defined in all host languages except Fortran. In Assembler, C, and COBOL, VARCHAR string variables are simulated as described in Section 3 of *DB2 Application Programming and SQL Guide*. In C, VARCHAR string variables can also be represented by NUL-terminated strings.

A VARCHAR string variable with a maximum length that is greater than 255 bytes or a CLOB string variable of any length is a *long string variable*. Long string variables are subject to the same restrictions as long string columns. For information, see “Restrictions using long strings” on page 72.

⁶ The syntax of the ALTER and CREATE TABLE statements allows a column to be defined as LONG VARCHAR as an alternative for VARCHAR(a) where a is the maximum number of characters that is associated with the column. However, after processing the CREATE or ALTER TABLE statement, DB2 considers the column to be VARCHAR(a).

Graphic strings

A *graphic string* is a sequence of DBCS 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. At a DBCS site, the CCSID of every graphic string column is the system CCSID for GRAPHIC data.

Fixed-length graphic strings

All the values of a column with a fixed-length graphic string data type have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 127 inclusive. Every fixed-length graphic string column is a short string column. 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:

- VARGRAPHIC⁷
- DBCLOB

The values of a column with any one of these string types can have different lengths. The length attribute of the column determines the maximum length a value can have. For varying-length graphic strings, the length control field specifies the number of DBCS characters.

For VARGRAPHIC columns, the length attribute of the column must be between 1 and m inclusive, where m is determined by the maximum record size as described in Maximum record size on page 592 in the description of the CREATE TABLE statement. For DBCLOB columns, the length attribute must be between 1 and 1 073 741 823 inclusive. In all cases, the length control field of a varying-length graphic string indicates the number of characters, not bytes. For more information about DBCLOBs, see “Large objects (LOBs)” on page 71.

A VARGRAPHIC column with a maximum length that is greater than 127 characters or a DBCLOB column of any length is a *long string column*. For the restrictions that apply to the use of long string columns, see “Restrictions using long strings” on page 72.

Graphic string host variables

Host variables with a graphic string type can be defined in all host languages except Fortran.

A VARGRAPHIC string variable with a maximum length that is greater than 127 characters or a DBCLOB string variable of any length is a *long string variable*. Long string variables are subject to the same restrictions as long string columns. For information, see “Restrictions using long strings” on page 72.

⁷ The syntax of the ALTER and CREATE TABLE statements allows a column to be defined as LONG VARGRAPHIC as an alternative for VARGRAPHIC(a) where a is the maximum number of characters that is associated with the column. However, after processing the CREATE or ALTER TABLE statement, DB2 considers the column to be VARGRAPHIC(a).

Binary strings

A *binary string* is a sequence of bytes. The length of a binary string (BLOB string) is the number of bytes in the sequence. The CCSID is X'FFFF' (65535).

For a BLOB column, the length attribute must be between 1 and 2 147 483 647 inclusive. (2 147 483 647 is 2 gigabytes minus 1 byte.)

A host variable with a BLOB string type can be defined in all host languages.

For more information about BLOBs, see “Large objects (LOBs).”

Large objects (LOBs)

The term *large object (LOB)* refers to any of the following data types:

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. 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. 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 considered to be 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.

Each LOB column is a long string column, and each LOB string variable is a long string variable. For information on the restrictions that apply to the use of long strings and the additional restrictions that apply only to LOBs, see “Restrictions using long strings” on page 72.

When an application does not need a LOB value to be stored in application memory, the application can use a *large object locator* (LOB locator) to reference the LOB value.

A LOB locator is a host variable with a value that represents a single LOB value in the database server. A LOB locator can also represent a LOB expression, such as:

```
SUBSTR( <lob 1> CONCAT <lob 2> CONCAT <lob 3>, <start>, <length> )
```

For information on manipulating LOBs with LOB locators, see Section 4 of *DB2 Application Programming and SQL Guide*.

Restrictions using long strings

A long string has one of the following varying-length string data types:

- *For character strings.* A VARCHAR string with a maximum length that is greater than 255 bytes or any CLOB string.
- *For graphic strings.* A VARGRAPHIC string with a maximum length that is greater than 127 characters or any DBCLOB string.
- *For binary strings.* Any BLOB string.

Table 3 indicates the contexts in which long strings cannot be referenced. The restrictions differ slightly for long strings with LOB data types (CLOB, DBCLOB, and BLOB).

Table 3. Contexts in which long strings cannot be referenced

Context of usage	VARCHAR (>255) or VARGRAPHIC (>127)	LOB (CLOB, DBCLOB, or BLOB)
A GROUP BY clause	Not allowed	Not allowed
An ORDER BY clause	Not allowed	Not allowed
A CREATE INDEX statement	Not allowed	Not allowed
A SELECT DISTINCT statement	Not allowed	Not allowed
A subselect of a UNION without the ALL keyword	Not allowed	Not allowed
# A host variable in a EXECUTE # IMMEDIATE or a PREPARE # statement	—	Not allowed
Predicates	Cannot be used in any predicate except EXISTS and LIKE. This restriction includes a <i>simple-when-clause</i> in a CASE expression. <i>expression When expression</i> in a <i>simple-when-clause</i> is equivalent to a predicate with <i>expression=expression</i> .	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 When expression</i> in a <i>simple-when-clause</i> is equivalent to a predicate with <i>expression=expression</i> .
# A search-condition or a # result-expression in a CASE # expression	Not allowed	Not allowed
The definition of primary, unique, and foreign keys	Not allowed	Not allowed
Check constraints	—	Cannot be specified for a LOB column
Field procedure	—	Cannot be specified for a LOB column.
Parameters of built-in functions	Some functions that allow varying-length character strings, varying-length graphic strings, or both types of strings as input arguments do not support VARCHAR or VARGRAPHIC long strings, CLOB or DBCLOB strings, or both as input. See the description of the individual functions in “Chapter 4. Built-in functions” on page 173 for the data types that are allowed as input to each function.	
Distributed data applications	—	LOB columns cannot be used if remote access is performed with DB2 private protocol access.

Numbers

The numeric data types are binary integer, floating-point, and decimal. Binary integer includes small integer and large integer. Floating-point includes single precision and double precision. Binary numbers are exact representations of integers, decimal numbers are exact representations of real numbers, and floating-point numbers are approximations of real numbers.

All numbers have a sign and a precision. When the value of a column or the result of an expression is a decimal or floating-point zero, its sign is positive. The precision of binary integers and decimal numbers is the total number of binary or decimal digits excluding the sign. The precision of floating-point numbers is either single or double, referring to the number of hexadecimal digits in the fraction.

Small integer (SMALLINT)

A *small integer* is a System/370™ 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 System/370 binary integer with a precision of 31 bits. The range of large integers is -2147483648 to +2147483647.

Single precision floating-point (REAL)

A *single precision floating-point* number is a System/370 short (32 bits) floating-point number. The range of single precision floating-point numbers is about $-7.2E+75$ to $7.2E+75$. In this range, the largest negative value is about $-5.4E-79$, and the smallest positive value is about $5.4E-079$.

Double precision floating-point (DOUBLE or FLOAT)

A *double precision floating-point* number is a System/370 long (64 bits) floating-point number. The range of double precision floating-point numbers is about $-7.2E+75$ to $7.2E+75$. In this range, the largest negative value is about $-5.4E-79$, and the smallest positive value is about $5.4E-079$.

Decimal (DECIMAL or NUMERIC)

A *decimal* number is a System/370 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 $10^{31} - 1$.

Numeric host variables

Binary integer variables can be defined in all host languages.

Floating-point variables can be defined in all host languages. All languages support System/390 floating-point format. Assembler, C, and C++ also support IEEE floating-point format. In assembler, C, and C++ programs, the precompiler option FLOAT tells DB2 whether floating-point variables contain data in System/390 floating-point format or IEEE floating-point format.

Decimal variables can be defined in all host languages except Fortran. In COBOL, decimal numbers can be represented in the packed decimal format used for columns or in the format denoted by DISPLAY SIGN LEADING SEPARATE.

Datetime values

The datetime data types are described in the following sections. Such 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 76.

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.⁸ The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to 28, 29, 30, or 31, depending on the month and year.

The internal representation of a date is a string of 4 bytes. Each byte consists of two packed decimal digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.

The length of a DATE column as described in the catalog is the internal length which is 4 bytes. The length of a DATE column as described in the SQLDA is the external length which is 10 bytes unless a date exit routine was specified when your DB2 subsystem was installed. (Writing a date exit routine is described in Appendix B (Volume 2) of *DB2 Administration Guide*.) In that case, the string format of a date can be up to 255 bytes in length. Accordingly, DCLGEN⁹ 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.

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 your DB2 subsystem was installed. (Writing a date exit routine is described in Appendix B (Volume 2) of *DB2 Administration Guide*.) In that case, the string format of a time can be up to 255 bytes in length. Accordingly, DCLGEN 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.

⁸ 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.

⁹ DCLGEN is a DB2 DSN subcommand for generating table declarations for designated tables or views. The declarations are stored in MVS data sets, for later inclusion in DB2 source programs.

Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) that represents a date and time as defined previously, except that the time 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`⁹ therefore defines 26-byte, fixed-length string variables for `TIMESTAMP` columns.

String representations of datetime values

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 character strings. These representations directly concern the SQL user because there are no special SQL constants for datetime values and no host variables with a data type of date, time, or timestamp.

For retrieval, datetime values must be assigned to character string variables. When a date or time is assigned to a 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 value with `LOCAL` specified can be any short character string. The following sections describe the string formats that are recognized by DB2.

Datetime values that are represented by character strings can appear in contexts requiring values whose data types are date, time, timestamp by using the `DATE`, `TIME`, or `TIMESTAMP` functions.

Date strings: A string representation of a date is a character 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.

Table 4 on page 77 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 4. 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	—

Note: For LOCAL, the date exit for ASCII data is different (DSNXVDTA versus DSNXVDTX) than the exit for EBCDIC data.

Time strings: A string representation of a time is a character 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.

Table 5 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 5. Formats for string representations of times

Format name	Abbreviation	Time format	Example
International Standards Organization ¹⁰	ISO	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: For LOCAL, the time exit for ASCII data is different (DSNXVTMA versus DSNXVTMX) than the exit for EBCDIC data.

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.

¹⁰ This is an earlier version of the ISO format. JIS can be used to get the current ISO format.

- A single blank must precede the AM or PM.
- The hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM.

Using the ISO format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

- 12:01 AM through 12:59 AM correspond to 00.01.00 through 00.59.00
- 01:00 AM through 11:59 AM correspond to 01.00.00 through 11.59.00
- 12:00 PM (noon) through 11:59 PM correspond to 12.00.00 through 23.59.00
- 12:00 AM (midnight) corresponds to 24.00.00
- 00:00 AM (midnight) corresponds to 00.00.00

Timestamp strings: A string representation of a timestamp is a character 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.nnnnnn*. 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*.

Restrictions on the use of local datetime formats

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 literals. 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 above under For input: on page 78.

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. Each value in a ROWID column must be unique, and DB2 maintains the values permanently, even across table space reorganizations. 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 translation 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.

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*.

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 DISTINCT TYPE.

For example, the following statement creates a distinct type named AUDIO:

```
CREATE DISTINCT 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 upon the context in which the distinct type appears. If an unqualified distinct type name is used:

- In a CREATE DISTINCT TYPE or the object of DROP, COMMENT ON, 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 “Schemas and the SQL path” on page 57.

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:

Data Types

```
| CREATE DISTINCT TYPE MONEY AS DECIMAL(9,2) WITH COMPARISONS;
```

```
| 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, a
| table can only have one ROWID column. Therefore, a table with a ROWID column
| cannot also have a column with distinct type that is sourced on a row ID.

| The comparison operators are automatically generated for distinct types, except
| those that are sourced 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 127)
- Casting distinct types (see “Casting between data types” on page 83)
- Assigning distinct types to built-in data types (see “Distinct type assignments” on page 92)

For each data type, Table 6 on page 82 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.

Promotion of Data Types

Table 6. Precedence of data types

Data type ^{1,2}	Data type precedence list (in best-to-worst order)
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB or DBCLOB
CLOB or DBCLOB	CLOB or DBCLOB
BLOB	BLOB
SMALLINT	SMALLINT, INTEGER, decimal, real, double
INTEGER	INTEGER, decimal, real, double
decimal	decimal, real, double
real	real, double
double	double
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
ROWID	ROWID
A distinct type	The same distinct type

Notes:

1. The data types in lowercase letters represent the following data types:

decimal DECIMAL(p,s) or NUMERIC(p,s)

real REAL or FLOAT(*n*) where *n* is not greater than 24

double DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is greater than 24

2. Other synonyms for the listed data types are considered to be the same as the synonym listed.

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, as described in “Promotion of data types” on page 81, is one example of when a value with one data type needs to be cast to a 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 92). 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 472).

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 unlike the assignment of character or graphic strings to a target when an error occurs if any non-blank characters are truncated.

For casts that involve a distinct type as either the data type to be cast to or from, Table 7 shows the supported casts. For casts between built-in data types, Table 8 on page 84 shows the supported casts.

Table 7. 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 81)
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
VARGRAPHIC	Distinct type <i>DT</i> if <i>DT</i> 's source data type is GRAPHIC

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 127 and “CAST specification” on page 146, 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.

Assignment and Comparison

Table 8. Supported casts between built-in data types

To data type →	V A R C H A R A B L E S T I M E S T A M P S R O W I D															
	S M A L L I N T	I N T E G E R	D E C I M A L	R E A L	D O U B L E	C H A R	V A R C H A R	C L O B	G R A P H I C	V A R G R A P H I C	D B C L O B	B L O B	D A T E	T I M E	T I M E S T A M P	R O W I D
Cast from data type ↓	T	R	L	L	E	R	R	B	C	C	B	B	E	E	P	D
SMALLINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
CHAR	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
CLOB	-	-	-	-	-	Y	Y	Y	Y	Y	-	Y	-	-	-	-
GRAPHIC	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-
VARGRAPHIC	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-
DBCLOB	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-
DATE	-	-	-	-	-	Y	Y	-	-	-	-	-	Y	-	-	-
TIME	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-
TIMESTAMP	-	-	-	-	-	Y	Y	-	-	-	-	-	Y	Y	Y	-
ROWID	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	Y

Note:

1. Other synonyms for the listed data types are considered to be the same as the synonym listed.

Assignment and comparison

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of INSERT, UPDATE, FETCH, SELECT INTO, SET *assignment*, 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 99. Table 9 on page 85 shows the compatibility matrix for data types.

Table 9. Compatibility of data types for assignments and comparisons. 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.

Operands	Binary integer	Decimal number	Floating point	Character string	Graphic string	Binary string	Date	Time	Time-stamp	Row ID	Distinct type
Binary Integer	Y	Y	Y	N	N	N	N	N	N	N	2
Decimal Number	Y	Y	Y	N	N	N	N	N	N	N	2
Floating Point	Y	Y	Y	N	N	N	N	N	N	N	2
Character String	N	N	N	Y	N	N ³	1	1	1	N	2
Graphic String	N	N	N	N	Y	N	N	N	N	N	2
Binary String	N	N	N	N ³	N	Y	N	N	N	N	2
Date	N	N	N	1	N	N	Y	N	N	N	2
Time	N	N	N	1	N	N	N	Y	N	N	2
Time-stamp	N	N	N	1	N	N	N	N	Y	N	2
Row ID	N	N	N	N	N	N	N	N	N	Y	2
Distinct Type	2	2	2	2	2	2	2	2	2	2	Y ²

Notes:

- The compatibility of datetime values is limited to assignment and comparison:
 - Datetime values can be assigned to character string columns and to character string variables, as explained in “Datetime assignments” on page 91.
 - 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 92.
- All character strings, even those with subtype FOR BIT DATA, are not compatible with binary strings.

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, nor to a 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 “Referencing host variables” on page 120 for a discussion of indicator variables.

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.

The following examples show a single precision floating-point number converted to an integer:

Example 1:

The floating-point number assigned to an integer column or host variable is:	2.0000045E6	2000000
--	-------------	---------

Example 2:

The floating-point number assigned to an integer column or host variable is:	2.00000555E8	200001000
--	--------------	-----------

The following examples show a double precision floating-point number converted to an integer:

Example 1:

The floating-point number assigned to an integer column or host variable is:	2.0000045E6	2000004
--	-------------	---------

Example 2:

The floating-point number assigned to an integer column or host variable is:	2.00000555E8	200000555
--	--------------	-----------

The following examples show a decimal number converted to an integer:

Example 1:

The decimal number	2000004.5
assigned to an integer	
column or host variable is:	2000004

Example 2:

The decimal number	200000555.0
assigned to an integer	
column or host variable is:	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.

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 or 11,0 for a large integer.

Floating-point to floating-point

When a single precision System/390 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 System/390 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 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 +5
Rounding adds 5 in the 16th position	.00000123456789148765
and truncates the result to	.000001234567891
Zeros are then added to the end of a 31-digit result:	.0000012345678910000000000000000

Example 2:

The floating-point number	1.2339999999999E+01
in decimal notation is:	12.33999999999900 +5
Rounding adds 5 in the 16th position	12.33999999999905
and truncates the result to	12.3399999999990
Zeros are then added to the end of a 31-digit result:	12.33999999999900000000000000000

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

The following rules apply when both the source and the target are strings. When a datetime data type is involved, see “Datetime assignments” on page 91. 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 92.

There are two types of string assignments:

- *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.

Storage assignment

The basic rule is that the length of the string 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 defined with the FOR BIT DATA attribute.

Retrieval Assignment

The length of a string 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 this occurs, the value W is assigned to the SQLWARN1 field of the SQLCA. Furthermore, if an indicator variable is provided, it is set to the original length of the string.

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.

Assignments involving mixed data strings

A mixed data string that contains DBCS 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 modified 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 referenced in a CONNECT statement need not contain a NUL (X'00'). 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 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, 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 placed in the last byte 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 placed in the next byte.

Conversion rules for string assignment

A character or graphic string 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 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.
- The SYSSTRINGS catalog table indicates that conversion is required.

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 Language Environment.
- 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.
- A mixed data string that contains DBCS characters is assigned to an SBCS column.

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, 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 DATE, TIME, or TIMESTAMP value can only be assigned to a column with a matching data type (whether DATE, TIME, or TIMESTAMP), a character string column, or a character string variable. The character string column or string variable can be fixed or varying-length, but must not be a CLOB column or variable. A value assigned to a DATE, TIME, or TIMESTAMP column must have a matching data type (whether DATE, TIME, or TIMESTAMP) or must be a valid character string representation of the matching data type. The character string representation must not be a CLOB. 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 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 on the type of target.

- If the target is a character column, 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

Assignment and Comparison

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 only be assigned 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.

| 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 DISTINCT TYPE AGE AS SMALLINT WITH COMPARISONS;
```

| 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 sourced 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 7 on
| page 83 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 DISTINCT TYPE AGE AS SMALLINT WITH COMPARISONS
```

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, Table 10 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 10. 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

Numeric comparisons

Numbers are compared algebraically, that is, with regard to sign. For example, -2 is less than $+1$.

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.

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

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. 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* may name a field procedure.

Conversion rules for string comparison

When two strings are compared, one of the strings is first converted, if necessary, to the coded character set of the other string. When it occurs, conversion takes place after any application of a field procedure. 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 BIT data or is a BLOB string).
- The string selected for conversion is neither null nor empty.
- The SYSSTRINGS catalog table indicates that conversion is required.

The conversion that occurs when SBCS data is compared with mixed data depends on the value of the field MIXED DATA on installation panel DSNTIPF at the DB2 that does the comparison:

- If this value is YES, the SBCS operand is converted to MIXED.
- If this value is NO, the MIXED operand is converted to SBCS.

Otherwise, the string selected for conversion depends on the type of the operands. The following table shows which operand supplies the target CCSID, given the operand types.

Table 11. Operand that supplies the CCSID for character conversion

First operand	Second operand				
	Column value	String constant	Special register	Derived value	Host variable
Column Value	first	first	first	first	first
String Constant	second	first	first	first	first
Special Register	second	first	first	first	first
Derived Value	second	second	second	first	first
Host Variable	second	second	second	second	first/second ¹

Note: 1. Both operands are converted, if necessary, to the system CCSID of the server.

For example, assume a comparison of the form:

```
string-constant = derived-value
```

Here, the relevant table entry is in the second row and fourth column. The value for this entry shows that the first operand (*string-constant*) supplies the target CCSID. Thus, the derived value is converted, if necessary, to the coded character set of the string constant.

Assignment and Comparison

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 Language Environment. . A warning occurs if a character of the string is converted to a substitution character.

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 involving TIME values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds are implied.

Comparisons involving 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.

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. Therefore, 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.

Table 12 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 DISTINCT TYPE AGE_TYPE AS INTEGER WITH COMPARISONS;
CREATE DISTINCT TYPE CAMP_DATE AS DATE WITH COMPARISONS;

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 12 (Page 1 of 2). 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;	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 > 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);	Yes	Integer ATTENDEE_NUMBER is cast to the distinct type AGE_TYPE, making both values the same distinct type.
Distinct types with constants		
SELECT * FROM CAMP_ROSTER WHERE AGE IN (15,16,17);	No	A distinct type cannot be compared to a constant.
SELECT * FROM CAMP_ROSTER WHERE INTEGER(AGE) IN (15,16,17);	Yes	The distinct type is cast to the data type of constants, making all the values in the comparison integers.
SELECT * FROM CAMP_ROSTER WHERE AGE IN (AGE_TYPE(15),AGE_TYPE(16),AGE_TYPE(17));	Yes	Constants are cast to distinct type AGE_TYPE, making all the values in the comparison the same distinct type.
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > '06/12/99';	No	A distinct type cannot be compared to a constant.
SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > CAST('06/12/99' AS CAMP_DATE);	No	The string constant '06/12/99', a VARCHAR data type, cannot be cast directly to distinct type CAMP_DATE, which is sourced 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.

Assignment and Comparison

Table 12 (Page 2 of 2). Examples of valid and invalid comparisons involving distinct types

SQL statement	Valid	Reason
<pre>SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > CAST(DATE('06/12/1999') AS CAMP_DATE);</pre>	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 sourced 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		
<pre>SELECT * FROM CAMP_ROSTER WHERE AGE BETWEEN :HV_INTEGER AND :HV_INTEGER2;</pre>	No	The host variables have integer data types. A distinct type cannot be compared to an integer.
<pre>SELECT * FROM CAMP_ROSTER WHERE AGE BETWEEN CAST(:HV_INTEGER AS AGE_TYPE) AND AGE_TYPE(:HV_INTEGER2);</pre>	Yes	The host variables are cast to distinct type AGE_TYPE, making all the values the same distinct type.
<pre>SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > :HV_VARCHAR;</pre>	No	The host variable has a VARCHAR data type. A distinct type cannot be compared to a VARCHAR.
<pre>SELECT * FROM CAMP_ROSTER WHERE FIRST_CAMP_DATE > CAST(DATE(:HV_VARCHAR) AS CAMP_DATE);</pre>	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 sourced 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. This section explains when those rules apply and lists them by the possible data types of operands.

The rules apply to:

- Corresponding columns in UNION or UNION ALL operations
- Result expressions of a CASE expression
- Arguments of the scalar functions COALESCE, IFNULL, and VALUE
- Expression values of the IN list of an IN predicate

The rules are applied subject to other restrictions on long strings for the various operations.

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.

Character string operands

Character strings are compatible only with other character strings.

Table 13. Result data types with character string operands

One operand	Other operand	Data type of the result
CHAR(x)	CHAR(y)	CHAR(z) where $z = \max(x,y)$
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where $z = \max(x,y)$
CLOB(x)	CHAR(y), VARCHAR(y), or CLOB(y)	CLOB(z) where $z = \max(x,y)$

Note: CREATE TABLE and ALTER TABLE allow a column to be defined as LONG VARCHAR as a syntax alternative for VARCHAR(a) where a is the maximum number of characters that is associated with the column. However, after the CREATE or ALTER TABLE statement is processed, the column is considered to be VARCHAR(a).

Graphic string operands

Graphic strings are compatible only with other graphic strings.

Table 14 (Page 1 of 2). Result data types with graphic string operands

One operand	Other operand	Data type of the result
GRAPHIC(x)	GRAPHIC(y)	GRAPHIC(z) where $z = \max(x,y)$
VARGRAPHIC(x)	GRAPHIC(y) or VARGRAPHIC(y)	VARGRAPHIC(z) where $z = \max(x,y)$
DBCLOB(x)	GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where $z = \max(x,y)$

Rules for Result Data Types

Table 14 (Page 2 of 2). Result data types with graphic string operands

One operand	Other operand	Data type of the result
Note: CREATE TABLE and ALTER TABLE allow a column to be defined as LONG VARGRAPHIC as a syntax alternative for VARGRAPHIC(<i>b</i>) where <i>b</i> is the maximum number of characters that is associated with the column. However, after the CREATE or ALTER TABLE statement is processed, the column is considered to be VARGRAPHIC(<i>b</i>).		

Binary string operands

Binary strings (BLOBs) are compatible only with other binary strings (BLOBs). The data type of the result is a BLOB. Other data types can be treated as a BLOB data type by using the BLOB scalar function to cast the data type to a BLOB. The length of the result BLOB is the largest length of all the data types.

Table 15. Result data types with binary string operands

One operand	Other operand	Data type of the result
BLOB(<i>x</i>)	BLOB(<i>y</i>)	BLOB(<i>z</i>) where $z = \max(x, y)$

Numeric operands

Numeric types are compatible only with other numeric types.

Table 16. Result data types with numeric operands

One operand	Other operand	Data type of the result
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
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>)	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 and $s = \max(x, z)$
REAL	REAL	REAL
REAL	DECIMAL, INTEGER, or SMALLINT	DOUBLE
DOUBLE	any numeric	DOUBLE

Note:

1. Precision cannot exceed 31.

Datetime operands

A DATE type is compatible with another DATE type, or any CHAR or VARCHAR expression that contains a valid string representation of a date. The data type of the result is DATE.

A TIME type is compatible with another TIME type, or any CHAR or VARCHAR expression that contains a valid string representation of a time. The data type of the result is TIME.

A TIMESTAMP type is compatible with another TIMESTAMP type, or any CHAR or VARCHAR expression that contains a valid string representation of a timestamp. The data type of the result is TIMESTAMP.

Row ID operands

A row ID data type is compatible only with itself. The result has a row ID data type.

Distinct type operands

A distinct type is compatible only with itself. The data type of the result is the distinct type.

Nullable attribute of a result

With the exception of the COALESCE and VALUE functions, the result of an operation can be null unless the operands do not allow nulls.

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, or decimal. String constants are classified as character or graphic.

All constants have the attribute NOT NULL. A negative sign in a numeric constant with a value of zero is ignored.

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 12 on page 97, which contains an example of casting data types to compare a constant to a distinct type.

Integer constants

An *integer constant* specifies a binary integer as a signed or unsigned number that has a maximum of 10 significant digits and no decimal point. If the value is not within the range of a large integer, the constant is interpreted as a decimal constant. The data type of an integer constant is large integer.

Examples:

64 -15 +100 32767 720176

In syntax diagrams, the term *integer* is used for an integer constant that must not include a sign.

Floating-point constants

A *floating-point constant* specifies a 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. The data type of a floating-point constant is double precision floating-point.

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* specifies a decimal number as 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; 23 and 2:

```
025.50  1000.  -15.  +37589333333333333333.33
```

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 in string delimiters" on page 168. 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 255. 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 254. 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 rightmost string in the example (') represents an empty character string constant, which is a string of zero length.

At DBCS sites, 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. The CCSID assigned to the constant is the appropriate system CCSID of the application server. A mixed string constant can be continued from one line to the next only if the break occurs between single byte characters.

Datetime constants

A *datetime constant* is a character string constant of a particular format. Character string constants are described under the previous heading, “Character string constants” on page 102. For information about the valid string formats, see “String representations of datetime values” on page 76.

Graphic string constants

A *graphic string constant* specifies a varying-length graphic string. (Shift-in and shift-out characters for EBCDIC data are discussed in “Character strings” on page 67.)

In EBCDIC environments, the forms of graphic string constants are¹¹:

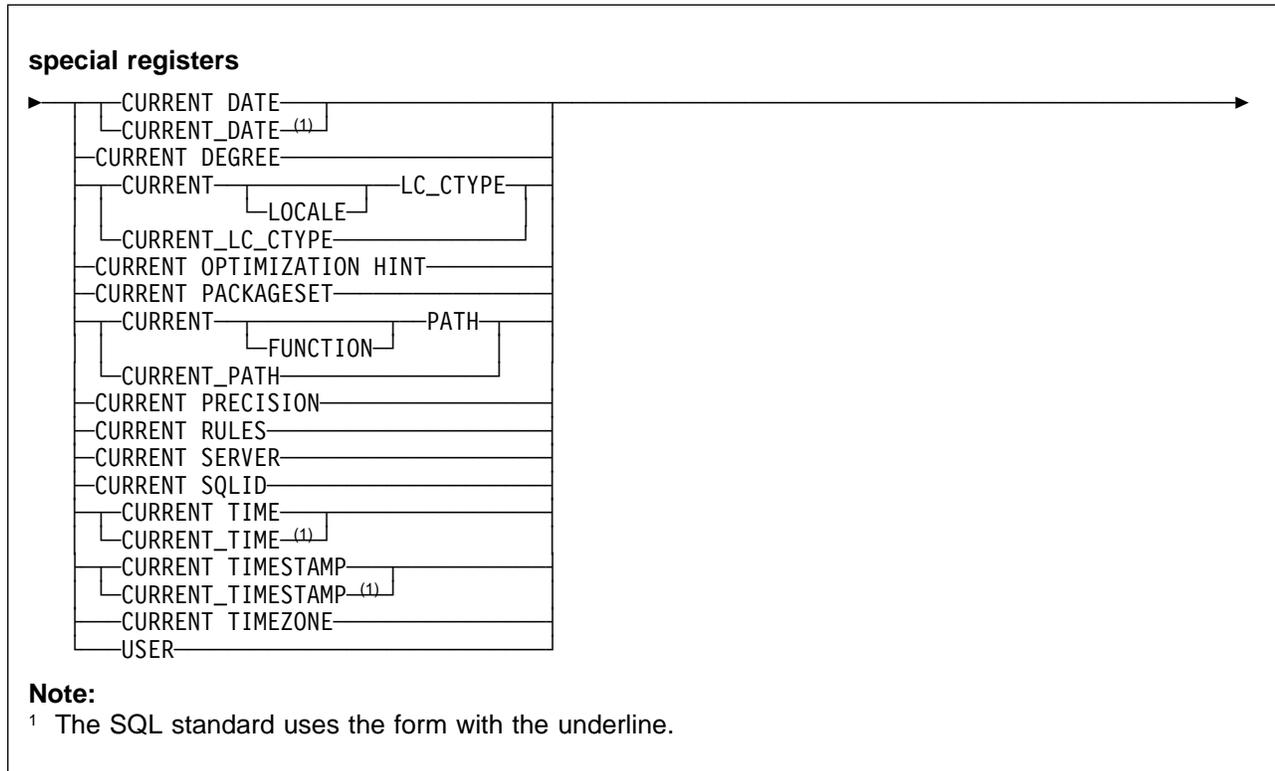
Context	Graphic String Constant	Empty String	Example
PL/I	$\text{S}_0 \text{ / dbcs-string / G } \text{S}_1$ $\text{S}_0 \text{ / dbcs-string / } \text{S}_1 \text{ G}$	$\text{S}_0 \text{ / / G } \text{S}_1$ $\text{S}_0 \text{ / / } \text{S}_1 \text{ G}$	$\text{S}_0 \text{ / 元 気 / G } \text{S}_1$
	<p>G represents a DBCS G (X'42C7')</p> <p>/ represents a DBCS apostrophe (X'427D')</p>		
All other contexts	$\text{G ' S}_0 \text{ dbcs-string } \text{S}_1 \text{ '}$	$\text{G ' S}_0 \text{ S}_1 \text{ '}$ G '' $\text{g ' S}_0 \text{ S}_1 \text{ '}$ g ''	$\text{G ' S}_0 \text{ 元 気 } \text{S}_1 \text{ '}$
	$\text{N ' S}_0 \text{ dbcs-string } \text{S}_1 \text{ '}$	$\text{N ' S}_0 \text{ S}_1 \text{ '}$ N '' $\text{n ' S}_0 \text{ S}_1 \text{ '}$ n ''	

In SQL statements and in host language statements in a source program, graphic string constants cannot be continued from one line to the next. The maximum number of DBCS characters in a graphic string constant is 124.

¹¹ The PL/I form of graphic string constants is supported only in static SQL statements.

Special registers

A special register is a storage area defined for a process by DB2. Wherever its name appears in an SQL statement, the name is replaced by the register's value when the statement is executed. Thus, the name acts like a function that has no arguments. The form of a special register is as follows:



General rules for special registers

Following these general rules for special registers, each special register is described individually.

Changing register values: A commit or 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 CURRENT DEGREE, CURRENT LOCALIZATION CTYPE, CURRENT OPTIMIZATION HINT, CURRENT PACKAGESET, CURRENT PATH, CURRENT PRECISION, CURRENT RULES, and CURRENT SQLID¹².
- SQL CONNECT statements can change the value of CURRENT SERVER.

¹² 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 Chapter 2 of *DB2 Command Reference*.

CCSIDS for register values: The values of certain special registers are character strings. The registers with string values are:

- CURRENT DEGREE
- CURRENT LOCALE LC_CTYPE
- CURRENT OPTIMIZATION HINT
- CURRENT PACKAGESET
- CURRENT PATH
- CURRENT PRECISION
- CURRENT RULES
- CURRENT SERVER
- CURRENT SQLID
- USER

The CCSID that is associated with these registers is either the one named in the ASCII CODED CHAR SET or EBCDIC CODED CHAR SET field on installation panel DSNTIPF at the server executing the statement. The CCSID that is used depends on whether the SQL statement in which the special register is referenced involves data in ASCII or EBCDIC tables; if no table is involved, the CCSID for the default encoding scheme for your system is used. Field DEF ENCODING SCHEME on installation panel specifies whether the default encoding scheme is EBCDIC or ASCII.

Datetime special registers: The datetime registers are named CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP. Datetime special registers are stored in an internal format. When two or more of these registers are implicitly or explicitly specified in a single SQL statement, they represent the same point in time. A datetime special register is implicitly specified when it is used to provide the default value of a datetime column.

If the SQL statement in which a datetime special register is used is in a user-defined function or stored procedure that is within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement to determine the special register value.

The values of these special registers are based on:

- The time-of-day clock of the processor for the server executing the SQL statement
- The MVS 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 MVS TIMEZONE parameter is zero or the clock is set to GMT and the MVS 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 MVS parameter PARMTZ(SYS1.PARMLIB(CLOCKXX)), their values are affected if the MVS local time at the server is changed by the MVS system command SET CLOCK. The values of the CURRENT DATE and CURRENT

Special Registers

TIMESTAMP special registers might be affected if the MVS local date at the server is changed by the MVS system command SET DATE¹³.

Where special registers are processed: In distributed applications, CURRENT
SERVER and CURRENT PACKAGESET are processed locally. All other special
registers are processed at the server.

CURRENT DATE

CURRENT DATE, or equivalently CURRENT_DATE, specifies the current date. The data type is DATE. 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 on page 105.

Example: Display the average age of employees.

```
SELECT AVG(YEAR(CURRENT DATE - BIRTHDATE))
FROM DSN8610.EMP;
```

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 Section 5 (Volume 2) of *DB2 Administration 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 DSNTIP4. 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 stored procedure or user-defined function is inherited from the invoker.

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 812.

CURRENT DEGREE is a register at the application server. Its value applies to queries that are dynamically prepared at that application 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 application server and that DB2 subsystem.

Example: The following statement inhibits parallelism:

```
SET CURRENT DEGREE = '1';
```

¹³ Whether the SET DATE command affects these special registers depends on the MVS system level and the program temporary fix (PTF) level of the system.

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 stored procedure or user-defined function is inherited from the invoker.

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 814.

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 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 CHAR(8).

The value of the register identifies the rows in auth.PLAN_ID that DB2 uses to generate the access path. If the value of the register is 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 DSNTIP4), a warning occurs.

The initial value of CURRENT OPTIMIZATION HINT is the value of the OPTHINT bind option. The initial value of CURRENT OPTIMIZATION HINT in a stored procedure or user-defined function is determined in the following way:

- If a SET CURRENT OPTIMIZATION HINT statement is executed within the scope of the user-defined function or stored procedure invoker, the initial value of CURRENT OPTIMIZATION HINT is inherited from the invoker.
- Otherwise, the initial value is the value of the OPTHINT bind option for the user-defined function or stored procedure package.

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 816.

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 Section 7 of *DB2 Application Programming and SQL Guide*.

CURRENT PACKAGESET

CURRENT PACKAGESET specifies a string of blanks or the collection ID of the package or packages that will be used to execute SQL statements. The data type is CHAR(18). If necessary, the collection ID is padded on the right with blanks so that its length is 18 bytes.

The initial value of CURRENT PACKAGESET is blanks. 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 817. The initial value of CURRENT PACKAGESET in a stored procedure or user-defined function is determined in the following way:

- If the COLLID parameter was specified in the CREATE FUNCTION or CREATE PROCEDURE statement, the initial value of CURRENT PACKAGESET in the user-defined function or stored procedure is the value of the COLLID parameter.
- Otherwise, the initial value is inherited from the invoker.

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, or equivalently 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(254).

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 254 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 “Schemas and the SQL path” on page 57.

The initial value of the CURRENT PATH special register is:

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

The initial value of CURRENT PATH in a stored procedure or user-defined function is determined in the following way:

- If a SET CURRENT PATH statement is executed within the scope of the user-defined function or stored procedure invoker, the initial value of CURRENT PATH is inherited from the invoker.
- Otherwise, the initial value is the value of the PATH bind option for the user-defined function or stored procedure package.

You can change the value of the register by executing the statement SET CURRENT PATH. For details about this statement, see “SET CURRENT PATH” on page 819.

Example: Set the special register so that schema SMITH is searched before schemas SYSIBM, SYSFUN, and SYSPROC.

```
SET CURRENT 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), and the only valid values are 'DEC15' and 'DEC31'. 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.

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 stored procedure or user-defined function is inherited from the invoker.

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 822.

CURRENT PRECISION only affects dynamic SQL. If the value of CURRENT PRECISION is DEC15 when an SQL statement is dynamically prepared, DEC15 rules will apply. If the value of CURRENT PRECISION is DEC31 when an SQL statement is dynamically prepared, 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. For more information, see “Arithmetic with two decimal operands” on page 134.

Example: Set CURRENT PRECISION so that subsequent statements that are prepared use DEC31 rules for decimal arithmetic:

```
SET CURRENT PRECISION = 'DEC31';
```

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 application 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 stored procedure or user-defined function is inherited from the invoker.

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 823.

CURRENT RULES affects the statements listed in Table 17 on page 110. 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.

Table 17. Summary of statements affected by CURRENT RULES

Statement	What is affected	Details on page
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.	393
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. Whether DB2 creates an index for a ROWID column that is defined with GENERATED BY DEFAULT.	570
DELETE	Authorization requirements for searched DELETE.	653
GRANT	Granting privileges to yourself.	711
REVOKE	Revoking privileges to yourself.	772
UPDATE	Authorization requirements for searched UPDATE.	833

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 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 stored procedure or user-defined function is inherited from the invoker. 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:

- The application process is in the unconnected state, or
- 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 CHAR(8). If necessary, the authorization ID is padded on the right with blanks so that its length is 8 bytes.

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 initial value of CURRENT SQLID in a stored procedure or user-defined function is determined in the following way:

- If a SET CURRENT SQLID statement is executed within the scope of the user-defined function or stored procedure invoker, the initial value of CURRENT SQLID is inherited from the invoker.
- Otherwise, the initial value is the primary authorization ID of the application process.

CURRENT SQLID can only be referred to in an SQL statement that is executed by the current server.

Example: Set the SQL authorization ID to 'GROUP34' (one of the authorization IDs of the process).

```
SET CURRENT SQLID = 'GROUP34';
```

CURRENT TIME

CURRENT TIME, or equivalently CURRENT_TIME, specifies the current time. The data type is TIME.

The time is derived by the DB2 that executes the SQL statement that refers to the special register. For a description of how the time is derived, see Datetime special registers on page 105.

Example: Display information about all project activities and include the current date and time in each row of the result.

```
SELECT DSN8610.PROJACT.*, CURRENT DATE, CURRENT TIME
FROM DSN8610.PROJACT;
```

CURRENT TIMESTAMP

CURRENT TIMESTAMP, or equivalently CURRENT_TIMESTAMP, specifies the current timestamp. The data type is TIMESTAMP.

The timestamp is derived by the DB2 that executes the SQL statement that refers to the special register. For a description of how the timestamp is derived, see Datetime special registers on page 105.

Example: 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 MVS 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 MVS TIMEZONE parameter is not between -24 and 24. The value of CURRENT TIMEZONE in a stored procedure or user-defined function is inherited from the invoker.

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 MVS TIMEZONE parameter to the difference from GMT.

```
SELECT DBNAME, TSNAME, DSNUM, ICTYPE, TIMESTAMP - CURRENT TIMEZONE
FROM SYSIBM.SYSCOPY;
```

USER

USER specifies the primary authorization ID of the process. The data type is CHAR(8). If necessary, the authorization ID is padded on the right with blanks so that its length is 8 bytes.

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 Section 3 (Volume 1) of *DB2 Administration Guide*.

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;
```

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 or in a CREATE FUNCTION statement that defines a table function.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
 - In a *column 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 5. Queries, which begins on page 307.) 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.
- Temporarily, rename a column, as in the *correlation-clause* of a table referenced in a FROM 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 ON and LABEL ON 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 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 115 and “Column name qualifiers in correlated references” on page 116 for details.

Correlation names

A *correlation name* can be defined in the FROM clause of a query and in the first clause of an UPDATE or DELETE statement. For example, the clause FROM X.MYTABLE Z establishes Z as a correlation name for X.MYTABLE.

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.

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:

```
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.

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, view, nested table expression, or table function. The tables, views, nested table expression, or table function reference 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 name the table from which the column comes.

Table designators: A qualifier that names 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 clause are named in the FROM clause that follows it, as in this statement:

```
SELECT DISTINCT Z.EMPNO, EMPTIME, PHONENO
FROM DSN8610.EMP Z, DSN8610.EMPPROJECT
WHERE WORKDEPT = 'D11'
AND EMPTIME > 0.5
AND Z.EMPNO = DSN8610.EMPPROJECT.EMPNO;
```

This example illustrates how to establish table designators in the FROM clause:

- A correlation name that follows a table name, view name, nested table expression, or table function is a table designator. Thus, Z is a table designator and qualifies the first column name after SELECT.
- A table name or view name that is *not* followed by a correlation name is a table designator. Thus, the qualified table name, DSN8610.EMPPROJECT is a table designator and qualifies the EMPNO column.

Avoiding undefined or ambiguous references in DB2 SQL: When a column name refers to values of a column, exactly one object table must include a column with that name. The following situations are considered errors:

Column Names

- 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.

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 DSN8610.EMP X, DSN8610.EMP Y
       WHERE Y.JOB = 'MANAGER'
             AND X.WORKDEPT = Y.WORKDEPT
             AND X.JOB <> 'MANAGER';
```

Column name qualifiers in correlated references

A *subselect* is a form of a query that can be used as a component of various SQL statements. Refer to “Chapter 5. Queries” on page 307 for more information on subselects. A subselect used within a search condition of any statement is called a *subquery*.

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 or DELETE statement. 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.C is used in a search condition 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 DSN8610.EMP at the level of the first FROM clause (which establishes X as a correlation name for DSN8610.EMP.). The statement lists employees who make less than the average salary for their department.

```
SELECT EMPNO, LASTNAME, WORKDEPT
       FROM DSN8610.EMP X
       WHERE SALARY < (SELECT AVG(SALARY)
                       FROM DSN8610.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

| 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 function
| reference is always exposed. A table name or a view name that is not followed by
| a correlation name is also exposed.

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.

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 SQLID. (If DYNAMICRULES bind behavior is in effect instead, A is the plan or package qualifier as determined during the bind process.) We refer to this step later 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 warnings indicate cases of ambiguous references in which the referent selected might not be the same one that would have been selected in releases of DB2 before Version 2 Release 3.

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.

Referencing host variables

A *host variable* is either of these items that is referred to in an SQL statement:

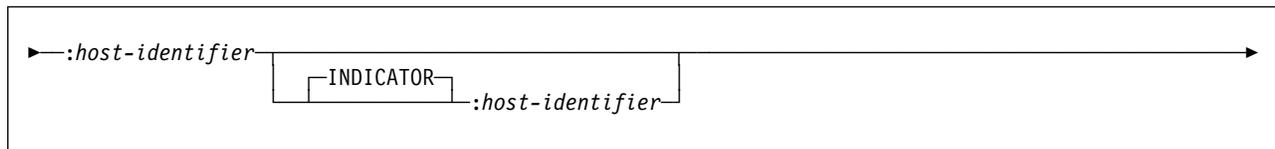
- A variable in a host language such as a PL/I variable, C variable, Fortran variable, COBOL data item, or Assembler language storage area
- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions

Host variables are defined directly by statements of the host language or indirectly by SQL extensions as described in Section 3 of *DB2 Application Programming and SQL Guide*. Host variables cannot be referenced in dynamic SQL statements.

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 123. 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 *assignment* 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 all other contexts, a host variable is an input variable which provides a value to DB2.

The general form of a host variable reference is:



Each host identifier must be declared in the source program. 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 -2 null indicates a numeric conversion or arithmetic expression error occurred in the SELECT list of an outer SELECT statement.
- 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 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.

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, that rule applies to the OPEN statement rather than to the DECLARE CURSOR statement.

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 on page 759 under the PREPARE statement.

References to LOB host variables

Regular LOB variables (CLOB, DBCLOB, and BLOB) and LOB locator variables (see “References to LOB locator variables” on page 122) can be defined in all host languages. Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable or a locator 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.

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 is not needed. However, host language restrictions, storage restrictions, or performance often dictate against storing an entire LOB value in temporary storage. When it is preferable not to store an entire LOB value, a LOB locator can be used to refer to the LOB value and portions of

Referencing Host Variables

the LOB value can be selected into or updated from host variables that contain only a portion of the LOB value.

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.

References to LOB locator variables

A LOB locator variable is a host variable that contains the locator representing a LOB value on the application 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 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.

At transaction commit, all LOB locators that were acquired by the transaction are released unless a HOLD LOCATOR statement was issued for the LOB locator. At transaction termination, all LOB locators are released.

References to stored procedure result sets

When an application needs to access a result set returned from a stored procedure, the invoking application must first define a result set locator to access the result set. An ASSOCIATE LOCATOR statement defines a result set locator, which identifies the stored procedure that returns the result set. The DESCRIBE PROCEDURE statement can be used to determine the number of result sets that a stored procedure returns, and the DESCRIBE CURSOR statement can be used to get information about a result set. The ALLOCATE CURSOR statement is used to define a cursor and associate it with a result set locator. The application then issues FETCH statements to retrieve rows from the result set.

References to result set locator variables

A result set locator variable is a host 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;
```

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 operation and the result set locators that are associated with those cursors. Otherwise, a cursor and its associated result set locator persist past the commit.

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. Host structures are defined by statements of the host language, as explained in Section 3 of *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 statement:

```
EXEC SQL FETCH CURSOR1 INTO :S1;
```

is equivalent to:

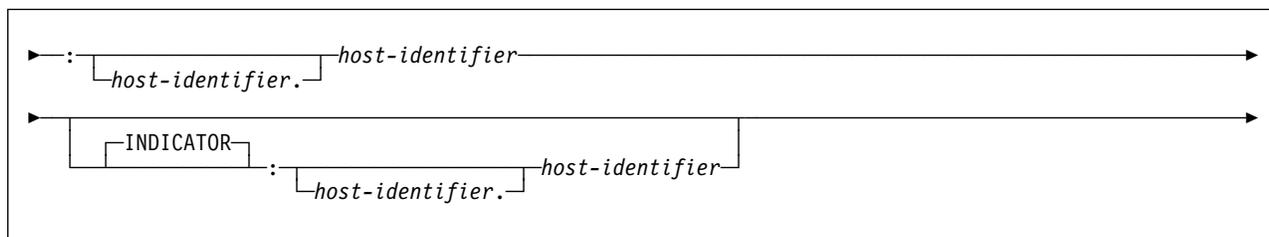
```
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3;
```

Host Structures in PL/I, C, and COBOL

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 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
```

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 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* are IBM-supplied functions that come with DB2 for OS/390 and are in the SYSIBM schema. Built-in functions include operator functions such as "+", column functions such as AVG, and scalar functions such as SUBSTR. For a list of the built-in column and scalar functions and information on these functions, see “Chapter 4. Built-in functions” on page 173.
- *User-defined functions* are functions that are registered to DB2 in catalog table SYSIBM.SYSROUTINES using the CREATE FUNCTION statement. These functions allow users to extend the function of the database system by adding their own or third party vendor function definitions.

A user-defined function is either *external* or *sourced*. An external function is defined to the database with a reference to a load module 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 are useful for supporting the use of built-in column and scalar functions for distinct types.

A user-defined function resides in the schema in which it was registered. The schema cannot be SYSIBM. In addition to being external or sourced, user-defined functions can be further categorized as scalar, column, or table functions.

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 Appendix F, “Sample user-defined functions” on page 1029. For more information on creating and using user-defined functions, see Section 3 of *DB2 Application Programming and SQL Guide*.

- Cast functions are automatically generated a distinct type is created using the CREATE DISTINCT TYPE statement. These 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.

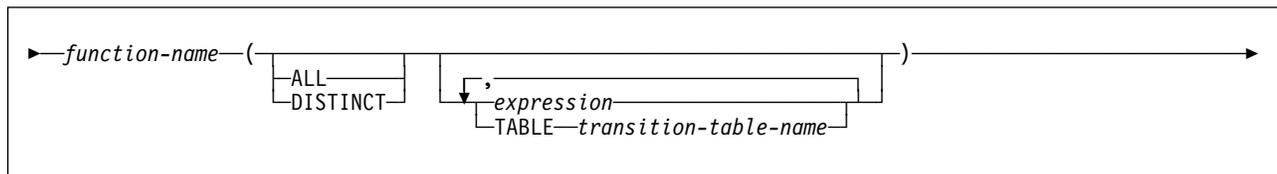
The generated cast functions reside in the same schema as the distinct type for which they were created. The schema cannot be SYSIBM. For more information on the functions that are generated for a distinct type, see “CREATE DISTINCT TYPE” on page 465.

Functions

Another way to classify functions is as column, scalar, or table functions, depending on the input data values and result values.

- A *column function* returns a single-value result for the argument it receives. The argument is a set of like values (such as the values of a column). Column functions are sometimes called *aggregating functions*. Built-in functions and user-defined sourced functions can be column functions. An external user-defined function cannot be a column function.
- A *scalar function* also returns a single-value result for the arguments it receives. Each argument is a single value. Built-in functions and user-defined functions, both external and sourced, 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. 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, get data from the World Wide Web and tabularize it, or access a Lotus® Notes™ database and return information about mail messages. Only external user-defined functions can be table functions.

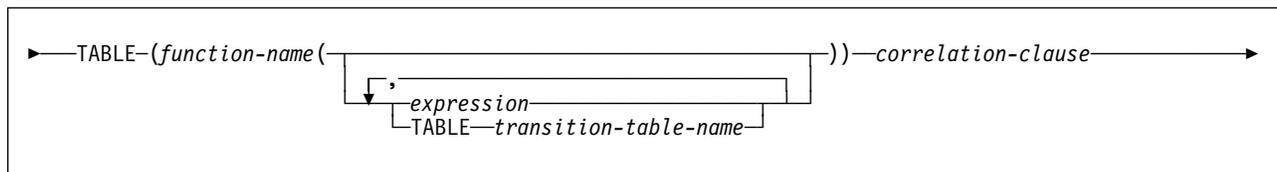
Each reference to a scalar or column function (either built-in or user-defined) conforms to the following syntax:



In the above syntax, *expression* cannot include a column name or column function.
See “Expressions” on page 131 for other rules for *expression*.

The ALL or DISTINCT keyword can only be specified for a column function or a user-defined function that is sourced on a column function. The TABLE keyword can only be used in a trigger body.

Each reference to a table function conforms to the following syntax:



In the above syntax, *expression* is the same as it is for a scalar or column function. For more details on referencing a table function, see the description of the FROM clause on page 315, the only place where a table function can be referenced.

For a description of the built-in functions, see “Chapter 4. Built-in functions” on page 173. *DB2 Application Programming and SQL Guide* contains detailed information on creating and using user-defined functions. (See Appendix F,

“Sample user-defined functions” on page 1029 for descriptions of the sample user-defined functions that are supplied with DB2).

Function resolution

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.

Because multiple functions with the same name can exist in the same schema or different schemas, DB2 must determine which function to execute. The process of choosing the function is called *function resolution*.

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 invoker of the function is authorized to execute 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.

This comparison of data types results in one best fit, which is the choice for execution (see “Method of finding the best fit” on page 128). For information on the promotion of data types, see “Promotion of data types” on page 81.

- 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 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 Appendix B (Volume 2) of *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 function 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 earliest in the SQL path.

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*.

Method of finding the best fit

More than one function instance with the same name might be a candidate for execution. In that case, DB2 compares the argument and parameter data types to determine which function is the best fit for the invocation.

If the data types of all the parameters for a given function are the same as those of the arguments in the function invocation, that function is the best fit. If there is no exact match, DB2 compares the data types in the parameter lists from left to right, using this method:

1. DB2 compares the data types of the first argument in the function invocation to the data type of the first parameter in each function. Any length, precision, scale, subtype, and encoding scheme attributes of the data types are not considered in the comparison.
2. 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 Table 6 on page 82 shows the data types that fit each data type, in best-to-worst order.

3. 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.

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)
```

The data types of the first parameter for the two function instances in the schema, which are both VARCHAR(10), fit the data type of the first argument of the function invocation, which is VARCHAR(10), 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 Function 1 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.
- 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).

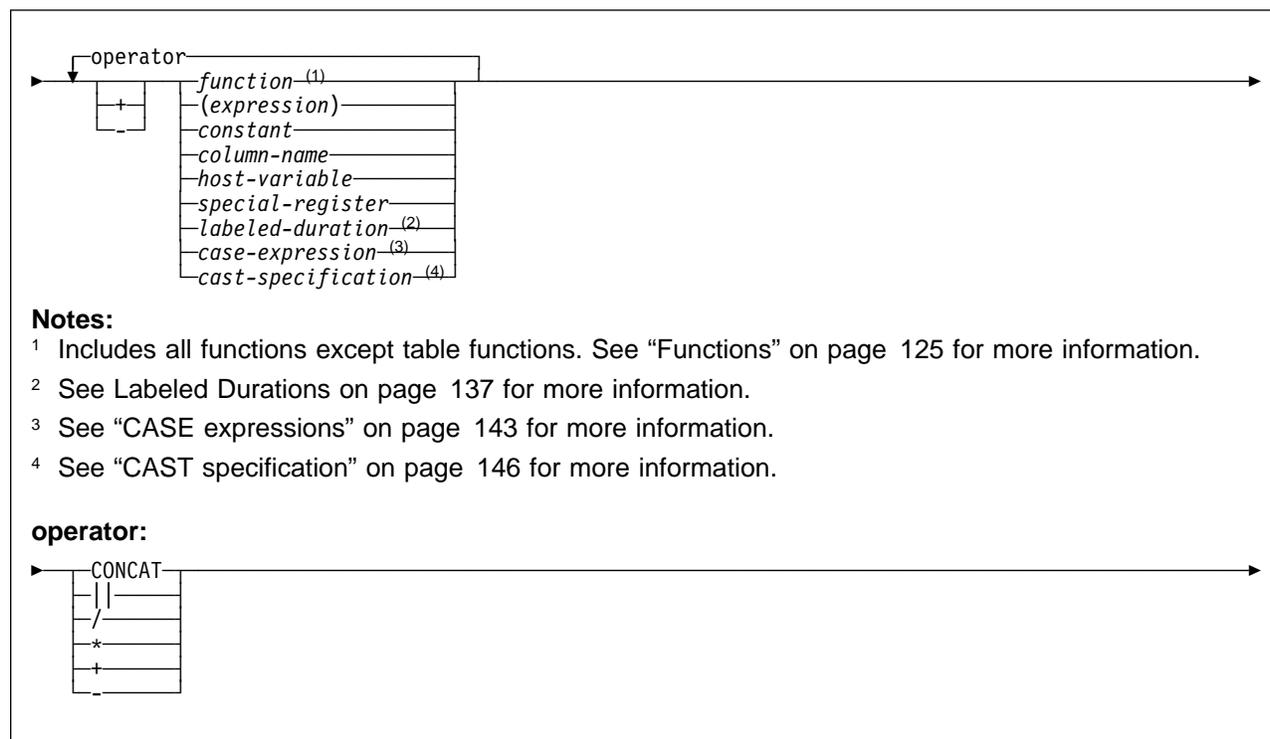
- Function 1 and 3 in schema SMITH are eliminated as candidates. The CHAR data type of the first parameter of Function 1 is not promotable to INT. Function 3 has the wrong number of parameters. Function 2 is a candidate because the data types of its parameters match or are promotable to the data types of the arguments.
- Of the remaining candidates, Function 2 and 5, DB2 selects Function 5 because schema JOHNSON comes before schema SMITH in the SQL path.

SQL path considerations for built-in functions

Function resolution applies to all functions, including built-in functions. The built-in functions are in schema SYSIBM. If a built-in function is invoked without its schema name, the SQL path is searched. If SYSIBM is not first in the path, it is possible that DB2 will select another function instead of the intended function. If schema SYSIBM 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 and SYSPROC. See “Schemas and the SQL path” on page 57 for information on how to specify the path so that the intended function is selected when it is invoked with an unqualified name.

Expressions

An *expression* specifies a value. The form of an expression is as follows:



Without operators

If no operators are used, the result of the expression is the specified value.

Examples:

SALARY :SALARY 'SALARY' MAX(SALARY)

With the concatenation operator

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 countries¹⁴) 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 CCSIDs¹⁴. Thus, CONCAT is the preferable 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 9 on page 85). A distinct type that is sourced

¹⁴ 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.

on a string type can be concatenated only if an appropriate user-defined function is created, as explained at the end of this section.

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.

Table 18 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 18. Data type and length of concatenated operands

One operand	Other operand	Combined length attribute	Result ¹
CHAR(A)	CHAR(B)	<256	CHAR(A+B) ²
	CHAR(B)	>255	VARCHAR(A+B)
	VARCHAR(B)	-	VARCHAR(A+B)
VARCHAR(A)	VARCHAR(B)	-	VARCHAR(A+B)
CLOB(A)	CHAR(B)	-	CLOB(MIN(A+B, 2G))
	VARCHAR(B)	-	CLOB(MIN(A+B, 2G))
	CLOB(B)	-	CLOB(MIN(A+B, 2G))
GRAPHIC(A)	GRAPHIC(B)	<128	GRAPHIC(A+B)
	GRAPHIC(B)	>127	VARGRAPHIC(A+B)
	VARGRAPHIC(B)	-	VARGRAPHIC(A+B)
VARGRAPHIC(A)	VARGRAPHIC(B)	-	VARGRAPHIC(A+B)
DBCLOB(A)	GRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
	VARGRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
	DBCLOB(B)	-	DBCLOB(MIN(A+B, 1G))
BLOB(A)	BLOB(B)	-	BLOB(MIN(A+B, 2G))

Notes:

1. 2G represents 2 147 483 647 bytes
1G represents 1 073 741 823 double-byte characters
2. Neither CHAR(A) nor CHAR(B) must contain mixed data. If either operand contains mixed data, the result is VARCHAR(A+B).

As Table 18 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 unions and concatenations” on page 328. Some consequences of those rules are the following:

- If either operand is BIT data, the result is BIT data.

- If one operand is mixed data and the other is SBCS data, the result is:
 - Mixed data if the MIXED DATA option at the server is YES¹⁵
 - SBCS data if the MIXED DATA option at the server is NO.

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. 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 sourced 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())
```

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.

¹⁵ The result is not necessarily well-formed mixed data.

Arithmetic with two integer operands

If both operands of an arithmetic operator are integers, the operation is performed in binary and the result is a large integer. Any remainder of division is lost. The result of an integer arithmetic operation (including unary minus) must be within the range of large integers.

Arithmetic with an integer and a decimal operand

If one operand is an integer and the other is decimal, the operation is performed in decimal using a temporary copy of the integer that has been converted to a decimal number with zero scale and precision as defined in the following table:

Operand	Precision of decimal copy
Column or variable: large integer	11
Column or variable: small integer	5
Constant: more than five digits (including leading zeros)	Same as the number of digits in the constant
Constant: five digits or fewer	5

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 bind option values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.

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 134).

Decimal division

The rules for a specific decimal division depend on three factors:

- Whether the DEC31 option is in effect for the operation
- Whether p is greater than 15
- 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 19. Precision (p) and scale (s) of the result of a decimal division

DEC31	p	p'	P	S
Not in effect	≤15	≤15	15	15-(p-s+s')
In effect	≤15	≤15	31	N-(p-s+s'), where N is 30-p' if p' is odd. N is 29-p' if p' is even.
N/A	>15	≤15	31	N-(p-s+s'), where N is 30-p' if p' is odd. N is 29-p' if p' is even.
N/A	N/A	>15	31	15-(p-s+x), where x is MAX(0,s'-(p'-15)) (See Note 2 below)

Notes on decimal division:

1. If the calculated value of S is negative, an error occurs.
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 MAX(0,s'-(p'-15)), which is the formula for x that appears in row 4 of Table 19. If, in the process of truncation, one or more nonzero digits are removed, SQLWARN7 in SQLCA is set to W, indicating loss of precision.
3. A value of YES for field MINIMUM DIVIDE SCALE on installation panel DSNTIPF specifies that the scale of the result of a decimal division is never less than 3. To this end, the precision and scale of the result are first calculated using the rules shown in Table 19. The actual scale is then the calculated scale or 3, whichever is greater. The actual precision is the calculated precision.

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.

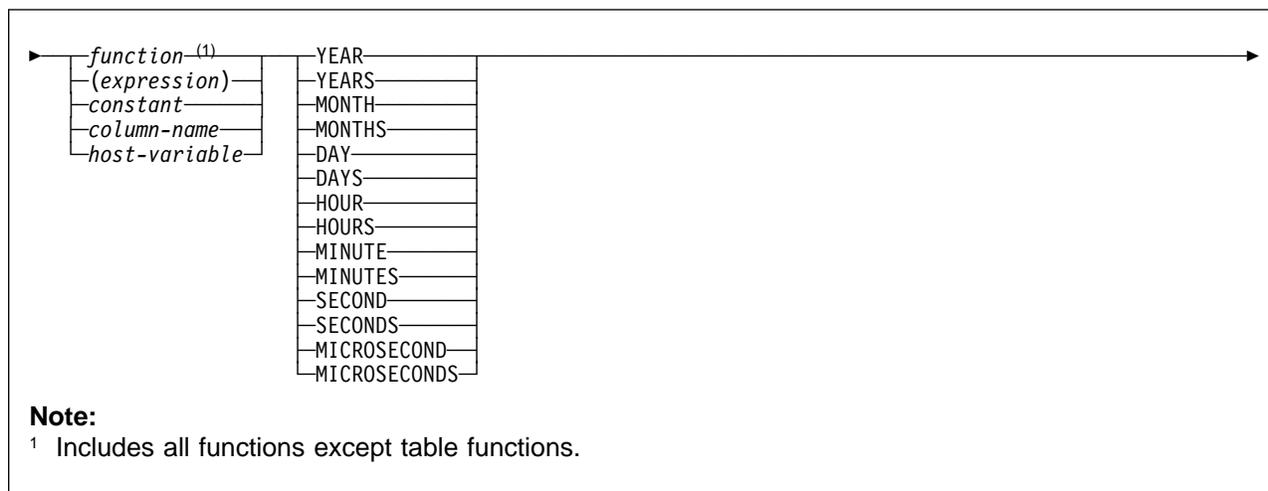
An operation involving a floating-point number and an integer is performed with a temporary copy of the integer that has been converted to double precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number that has been converted to double precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

Datetime operands and durations

Datetime values can be incremented, decremented, and subtracted. These operations may involve decimal numbers called durations. A *duration* is a number representing an interval of time. There are four types of durations:

Labeled Durations

The form a labeled duration is as follows:



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:¹⁶ YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS. 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, and the other operand must have a 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.

¹⁶ The singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

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 *yyyxxddhhmmsszzzzzz*, where *yyyy*, *xx*, *dd*, *hh*, *mm*, *ss* and *zzzzzz* 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 757.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of years, months, or days.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.

- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of hours, minutes, or seconds.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.
- Neither operand of the subtraction operator can be a parameter marker.

When an operand in a datetime expression is a string, it may 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

Dates 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 $N = \text{the last day of } MONTH(DATE2)$.
 $MONTH(DATE2)$ 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 $DATE('3/15/2000') - '12/31/1999'$ 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 138, the second

operand can be a character 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. Section 3 of *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.

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.

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 138, the second operand can be a character 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 may 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, $\text{TIME1} + X$, where X is a positive $\text{DECIMAL}(6,0)$ number, is equivalent to the expression

$$\text{TIME1} + \text{HOUR}(X) \text{ HOURS} + \text{MINUTE}(X) \text{ MINUTES} + \text{SECOND}(X) \text{ SECONDS}$$

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 $\text{DECIMAL}(20,6)$. 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 $\text{RESULT} = \text{TS1} - \text{TS2}$.

Timestamp subtraction: result = ts1 - ts2

- If $\text{MICROSECOND}(\text{TS2}) \leq \text{MICROSECOND}(\text{TS1})$
then $\text{MICROSECOND}(\text{RESULT}) = \text{MICROSECOND}(\text{TS1}) - \text{MICROSECOND}(\text{TS2})$.
- If $\text{MICROSECOND}(\text{TS2}) > \text{MICROSECOND}(\text{TS1})$
then $\text{MICROSECOND}(\text{RESULT}) = 1000000 + \text{MICROSECOND}(\text{TS1}) - \text{MICROSECOND}(\text{TS2})$
and $\text{SECOND}(\text{TS2})$ is incremented by 1.
- The seconds and minutes part of the timestamps are subtracted as specified in the rules for subtracting times.
- If $\text{HOUR}(\text{TS2}) \leq \text{HOUR}(\text{TS1})$
then $\text{HOUR}(\text{RESULT}) = \text{HOUR}(\text{TS1}) - \text{HOUR}(\text{TS2})$.
- If $\text{HOUR}(\text{TS2}) > \text{HOUR}(\text{TS1})$
then $\text{HOUR}(\text{RESULT}) = 24 + \text{HOUR}(\text{TS1}) - \text{HOUR}(\text{TS2})$
and $\text{DAY}(\text{TS2})$ is incremented by 1.
- 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'.

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.

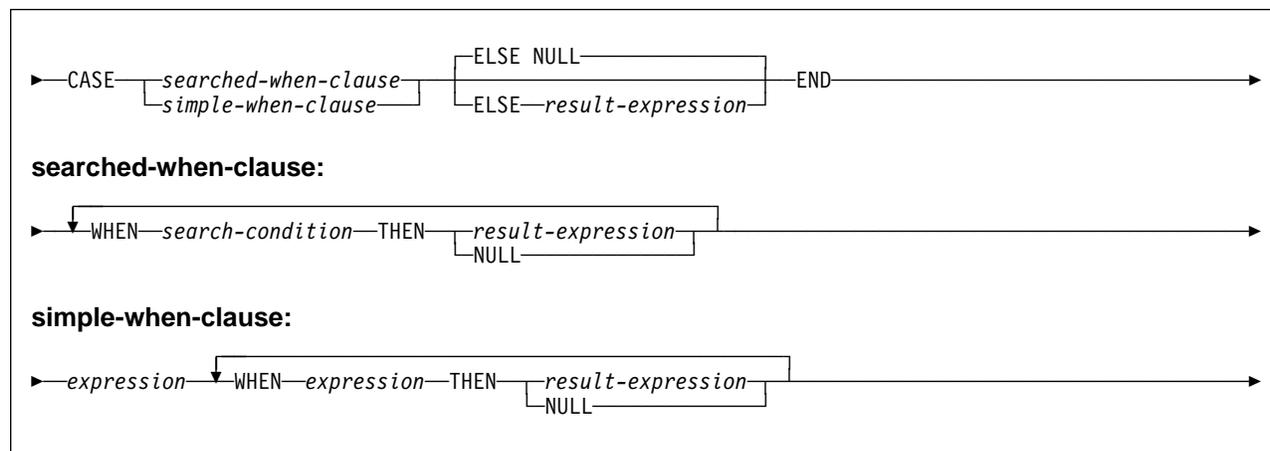
Example 1:

```
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. In general, the value of the case-expression is the value of the *result-expression* following the first (leftmost) case 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 NULLs), the case is NOT true and hence is treated the same way as a case that evaluates to false.

CASE

Begins a *case-expression*.

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 comparable to the data types of each *expression* that follows the WHEN keywords. 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 user-defined function that is nondeterministic or has an external action.

result-expression

Specifies an expression that follows the THEN and ELSE keyword. 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 be compatible. The attributes of the result are determined according to the rules that are described in “Rules for result data types” on page 99. When the result is a string, its attributes include a CCSID. For the rules on how the CCSID is determined, see “System CCSIDS” on page 40.

search-condition

Specifies a condition that is true, false, or unknown about a row or group of table data. The search-condition cannot contain a subselect. If the CASE expression is in a select list, an IN predicate, or a SET clause of an UPDATE statement, the search-condition cannot be a quantified predicate, an IN predicate, or an EXISTS predicate.

END

Ends a *case-expression*.

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 column 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
# SUM(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.

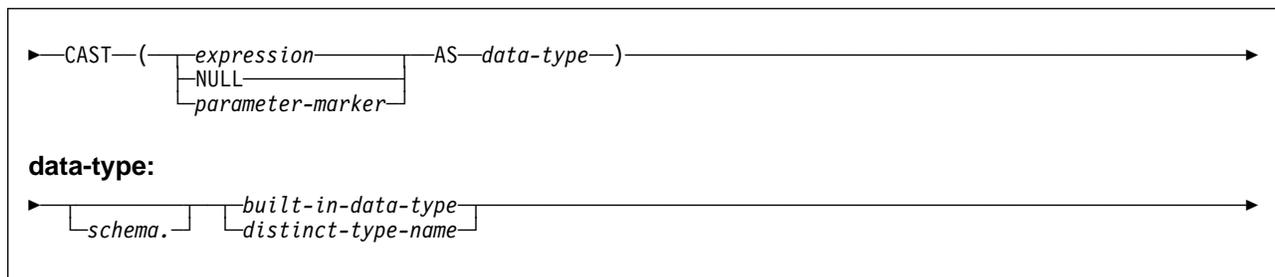
```
| SELECT CASE_DEPT,MAX(SALARY),MIN(SALARY),AVG(SALARY)
| FROM (SELECT SALARY,CASE WHEN WORKDEPT = 'A00' OR WORKDEPT = 'E21'
| THEN 'A00_E21'
| WHEN WORKDEPT = 'D11' OR WORKDEPT = 'E11'
| THEN 'D11_E11'
| ELSE WORKDEPT
| END AS CASE_DEPT
| FROM DSN8610.EMP) X
| GROUP BY CASE_DEPT;
```

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 20 shows the equivalent expressions using CASE or these functions.

Table 20. 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)

CAST specification



The CAST specification returns the first operand (the cast operand) converted to the data type that is specified by the second operand. 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. If the data type of the second operand is a distinct type, the privilege set must also include USAGE authority on the the distinct type.

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 *data type*.

Table 7 on page 83 and Table 8 on page 84 shows the supported casts between data types. If you specify an unsupported cast, an error occurs.

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 characters are truncated when a BLOB operand is cast.

NULL

Specifies that the cast operand is null. The result is a null value with the specified *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 that data type (using “store assignment” rules for strings). 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

The name of a built-in data type or a distinct type. If a data type has length, precision, or scale attributes, specify the attributes. If the attributes are not specified, the default values are used. For example, the default for CHAR is a length of 1, and the default for DECIMAL is a precision of 5 and a scale of 0. For the default values of the other data types, see the description of built-in-data-type on page 575 for the CREATE TABLE statement. (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 83 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.

Resolution of cast functions: DB2 uses the schema name and the data type name of the target data type to locate the function to use to convert the first operand to the data type of the second operand. If an unqualified data type name is specified for the second operand, DB2 first resolves the schema name of the data type (by searching the SQL path and selecting the first schema such that the data type exists in the schema and the user has authorization to use the data type). DB2 finds the appropriate cast function when all of the following conditions are true:

- The schema name of the function matches the schema name of the target data type.
- The function name matches the name of the target data type.
- The data type of the expression matches or is *promotable* to the data type of the function's parameter.

This comparison of data types results in one best fit, which is the choice for execution (see “Method of finding the best fit” on page 128). For information on the promotion of data types, see “Promotion of data types” on page 81.

- The user has EXECUTE authority on the function.
- The create timestamp for the function is older than the bind timestamp for the package or plan in which the CAST specification is used.

If DB2 authorization checking is in effect, and DB2 performs an automatic rebind on a plan or package that contains a CAST specification, any cast functions that were created after the original BIND or REBIND of the invoking plan or package are not candidates for execution.

Casting numeric data to character data: 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 196.) 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 INT function would give. (For more information see “INTEGER or INT” on
 | page 236.)

If the data type of the result is character, the subtype of the result is determined as
 # follows:

- # • If the *expression* and *data-type* are both character, the subtype of the result is
 # the same as the subtype of *expression*.
- # • If the field MIXED DATA on installation panel DSNTPF is NO, the subtype of
 # the result is SBSCS.
- # • If the *expression* is a row ID and *data-type* is character, the result has a
 # subtype of FOR BIT DATA, unless the *data-type* is CLOB.
- # • Otherwise, the subtype of the result is MIXED.

If the data type of the result is a string data type and not character FOR BIT DATA,
 # the encoding scheme of the result is determined as follows:

- # • If the *expression* and *data-type* are both character, the encoding scheme and
 # CCSID of the result are 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 and CCSID as the input.
- # • If the *expression* and *data-type* are both graphic, the encoding scheme and
 # CCSID of the result are the same as *expression*.
- # • If the result is character, the encoding scheme of the result depends on the
 # value of the field DEF ENCODING SCHEME on installation panel DSNTIPF.
 # The CCSID of the result is the default CCSID for the encoding scheme and
 # subtype of the result.
- # • If the result is graphic, the encoding scheme of the result depends on the value
 # of the field DEF ENCODING SCHEME on installation panel DSNTIPF. The
 # CCSID of the result is the default CCSID for the encoding scheme 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 DISTINCT TYPE D_MONEY AS DECIMAL(9,2) WITH COMPARISONS;
```

| 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 may 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 127. 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 DISTINCT TYPE SCHEMA1.AGE AS DECIMAL(2,0) WITH COMPARISONS;
    one of the generated cast functions is:
    FUNCTION SCHEMA1.AGE(SYSIBM.DECIMAL(2,0)) RETURNS SCHEMA1.AGE
```

```
CREATE DISTINCT TYPE SCHEMA2.AGE AS INTEGER WITH COMPARISONS;
    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 USAGE authority). 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 sourced on SMALLINT and is the data type for the AGE column in the PERSONNEL table. Distinct type D_YEAR was sourced on INTEGER and is the data type for the RETIRE_YEAR column in the same table. The following UPDATE statement could be prepared.

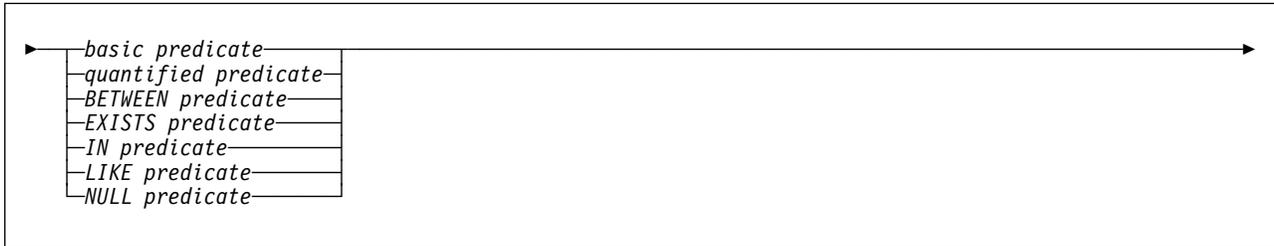
```
UPDATE PERSONNEL SET RETIRE_YEAR =?
    WHERE AGE = CAST( ? AS SCHEMAX.D_AGE);
```

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.

Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given row or group. The types of predicates are:

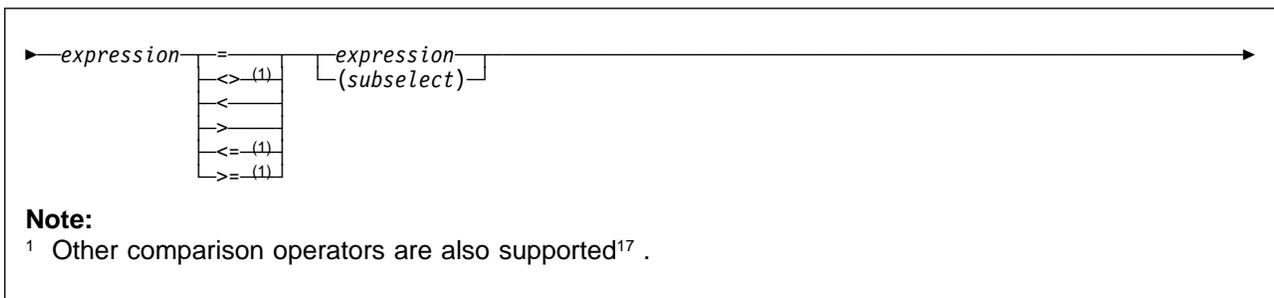


The following rules apply to predicates of any type:

- All values specified in a predicate must be compatible.
- Except for the first operand of LIKE, the operand of a predicate must not be a character string with a maximum length greater than 255 or a graphic string with a maximum length greater than 127.
- Except for EXISTS, a subselect in a predicate must specify a single column.

In addition to the examples of predicates in the following sections, see “Distinct type comparisons” on page 96, which contains several examples of predicates that use distinct types.

Basic predicate



A *basic predicate* compares two values. If the value of either operand is null or the result of the subselect is empty, the result of the predicate is unknown. Otherwise, the result is either true or false.

A subselect in a basic predicate must not return more than one value, whether null or not null.

¹⁷ The following forms of the comparison operators are also supported in basic and quantified predicates: !=, !<, 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 '!<'.

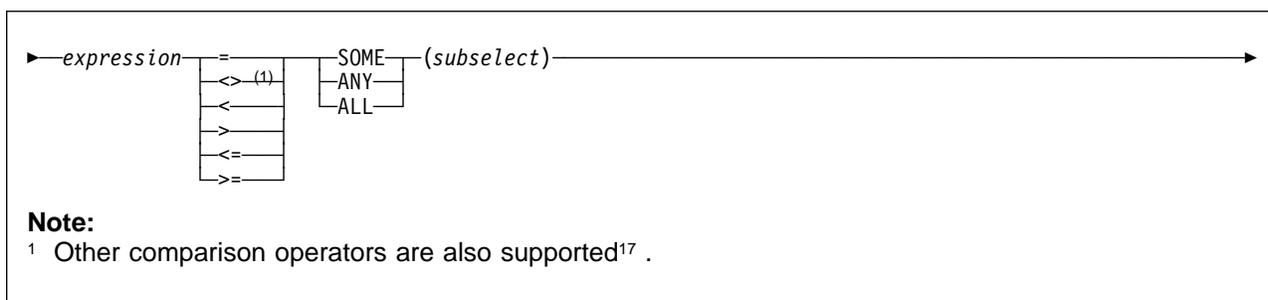
For values x and y:

Predicate	Is true if and only if...
x = y	x is equal to y
x <> y	x is not equal to y
x < y	x is less than y
x > y	x is greater than y
x <= y	x is less than or equal to y
x >= y	x is greater than or equal to y

Examples:

```
EMPNO = '528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY >=4 (SELECT AVG(SALARY) FROM DSN8610.EMP)
```

Quantified predicate



A *quantified predicate* compares a value with the set of values produced by the subselect. The subselect must specify a single result column and can return any number of values, whether null or not null.

When ALL is specified, the result of the predicate is:

- True if the result of the subselect is empty or if the specified relationship is true for every value returned by the subselect.
- False if the specified relationship is false for at least one value returned by the subselect.
- Unknown if the specified relationship is not false for any values returned by the subselect and at least one comparison is unknown because of a null value.

When SOME or ANY is specified, the result of the predicate is:

- True if the specified relationship is true for at least one value returned by the subselect.
- False if the result of the subselect is empty or if the specified relationship is false for every value returned by the subselect.
- Unknown if the specified relationship is not true for any of the values returned by the subselect and at least one comparison is unknown because of a null value.

Predicates

Examples: Use the tables below 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*.

TBLA:	COLA
	1
	2
	3
	4

TBLB:	COLB	COLC
	2	2
	3	--

Example 1: In the following predicate, the subselect 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)
```

Example 2: In the following predicate, the subselect 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)
```

Example 3: In the following predicate, the subselect 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)
```

Example 4: In the following predicate, the subselect 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)
```

Example 5: In the following predicate, the subselect returns an empty result column. Hence, the predicate is true for all rows of TBLA.

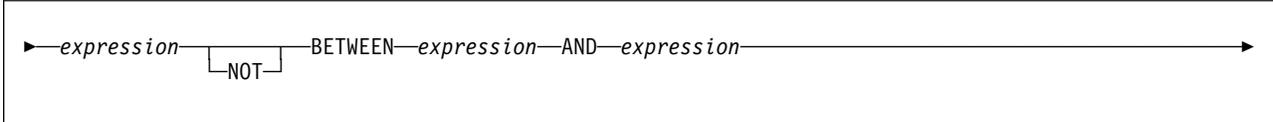
```
COLA < ALL(SELECT COLB FROM TBLB WHERE COLB>3)
```

Example 6: In the following predicate, the subselect returns an empty result column. Hence, the predicate is false for all rows of TBLA.

```
COLA < ANY(SELECT COLB FROM TBLB 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 specified in ascending order. Each of the predicate's two forms has an equivalent search condition, as shown below:

The predicate: `value1 BETWEEN value2 AND value3`
 is equivalent to: `value1 >= value2 AND value1 <= value3.`

The predicate: `value1 NOT BETWEEN value2 AND value3`
 is equivalent to: `NOT(value1 BETWEEN value2 AND value3)`
 and therefore also to: `value1 < value2 OR value1 > value3.`

Search conditions are discussed in “Search conditions” on page 163.

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.

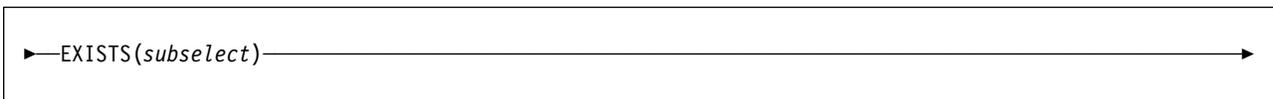
Example: Consider predicate:

`A BETWEEN B AND C`

The following table shows the value of the predicate for various values of A, B, and C.

Value of A	Value of B	Value of C	Value of predicate
1,2, or 3	1	3	true
0 or 4	1	3	false
0	1	null	false
4	null	3	false
null	any	any	unknown
2	1	null	unknown
3	null	4	unknown

EXISTS predicate



The EXISTS predicate tests for the existence of certain rows.

The result of the predicate is true if the result table returned by the subselect contains at least one row. Otherwise, the result is false.

Predicates

The SELECT clause in the subselect can specify any number of columns because the values returned by the subselect 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:

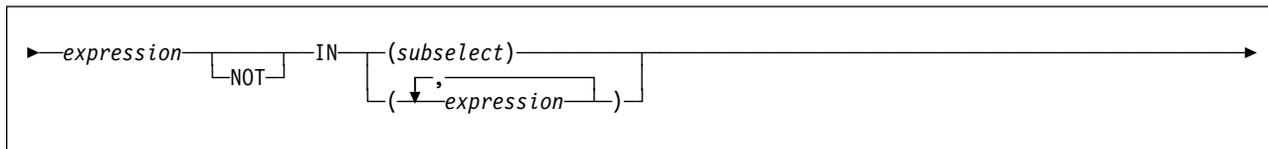
```
NOT EXISTS (subselect)
```

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 163. The result cannot be unknown.

Example: The following query lists the employee number of everyone represented in DSN8610.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 DSN8610.EMP X
WHERE EXISTS (SELECT * FROM DSN8610.EMP
              WHERE X.WORKDEPT=WORKDEPT AND SALARY<20000);
```

IN predicate



The IN predicate compares a value with a set of values.

In the subselect form, the subselect must specify a single result column and can return any number of values, whether null or not null. The IN predicate is equivalent to the quantified predicate as follows:

Predicate...	Is equivalent to quantified predicate...
--------------	--

expression IN (subselect)	expression = ANY (subselect)
expression NOT IN (subselect)	expression <> ALL (subselect)

In the non-subselect form of the IN predicate, the second operand is a set of one or more values specified by any combination of expressions. (For information on the types of expressions, see “Expressions” on page 131.) 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. An IN predicate of the form:

```
expression IN (value1, value2,..., valuen)
```

is logically equivalent to:

```
expression 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
  
```

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 “String comparisons” on page 94.

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 DSN8610.EMP
WHERE WORKDEPT = 'E11')
  
```

Predicates

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 DSN8610.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 DSN8610.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

```
match-expression [NOT] LIKE pattern-expression [ESCAPE escape-expression]
```

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 have the same string type. Therefore, the three arguments must all be character strings, all be graphic strings, or all be binary strings. 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)
- 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.

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 Section 3 of *DB2 Application Programming and SQL Guide*.

A simple description of the pattern

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.

A rigorous description of the pattern

This more rigorous description of the pattern ignores the use of the *escape-expression*, which is covered later.

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, 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. 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.
- Redundant shift bytes in *match-expression* or *pattern-expression* are ignored.

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 values that contain the actual percent and underscore characters to be matched.

The expression can be specified by any one of:

#

- A constant
- A host variable (including a LOB locator 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.¹⁸ 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 example shows the effect of successive occurrences of the escape character, which in this case is the plus sign (+).

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

¹⁸ If it is NUL-terminated, a C character string variable of length 2 can be specified.

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 '+'
```

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 DSN8610.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 DSN8610.EMP_PHOTO_RESUME  
WHERE RESUME LIKE '%JONES%';
```

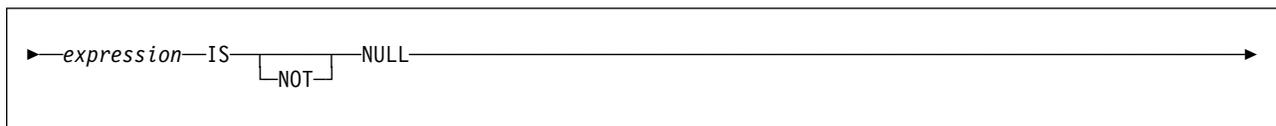
Example 6: In the following table, assume COL1 is a column that contains mixed EBCDIC data. The table shows the results when the predicate in the first column is evaluated using the COL1 value in the second column:

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 X C'	False
WHERE COL1 LIKE 'a _ C'	'a X C'	True
	'ax C'	False
WHERE COL1 LIKE ' ' C'	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 'aaaAB%C'	'aaaABDZC'	True
WHERE COL1 LIKE 'aaaAB %C'	'aaaAB dzxC'	True

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.

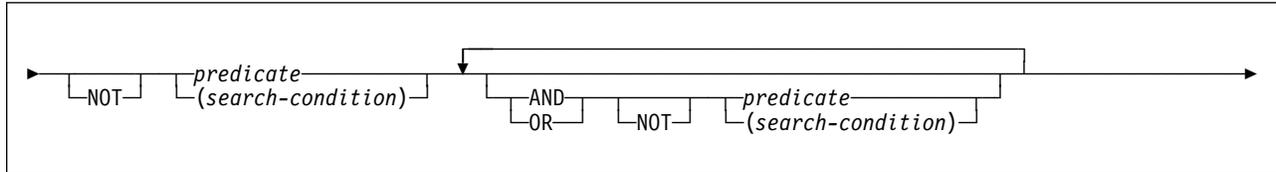
Predicates

Example: The following predicate is true whenever PHONENO has the null value, and is false otherwise.

```
PHONENO IS NULL
```

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 21. Truth table for AND and OR

P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown. The NOT logical operator has no effect 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)
```

Options affecting SQL

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 ORs could be applied in either order without changing the result. DB2 can therefore select the order of applying the ORs.

```
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, and special registers affect how SQL statements can be composed or determine how SQL statements are processed.

Table 22 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 22 (Page 1 of 3). Summary of items affecting composition and processing of SQL statements

Precompiler option	Other ¹	Affects
	DYNAMICRULES bind option	The rules that DB2 applies to dynamic SQL statements. For details about authorization, see “Authorization IDs and dynamic SQL” on page 61. The bind option can also affect decimal point representation, string delimiters, mixed data, and decimal arithmetic. For details about how DB2 applies the precompiler options to dynamic SQL statements when DYNAMICRULES bind, define, or invoke behavior is in effect, see “Precompiler options for dynamic statements” on page 166.
	USE FOR DYNAMICRULES	Use of precompiler options for dynamic statements when DYNAMICRULES bind, define, or, invoke behavior is in effect. For details, see “Precompiler options for dynamic statements” on page 166.
COMMA PERIOD	DECIMAL POINT IS	Representation of decimal points in SQL statements. For details, see page 166.
APOSTSQL QUOTESQL	SQL STRING DELIMITER	Representation of string delimiters in SQL statements. For details, see page 168.
	EBCDIC CODED CHAR SET	A numeric value that determines the CCSID of EBCDIC string data and whether Katakana characters can be used in ordinary identifiers. For details, see page 169.
	ASCII CODED CHAR SET	A numeric value that determines the CCSID of ASCII string data. For details, see page 169.

Table 22 (Page 2 of 3). Summary of items affecting composition and processing of SQL statements

Precompiler option	Other ¹	Affects
GRAPHIC NOGRAPHIC	MIXED DATA	Use of character strings with a mixture of SBCS and DBCS characters. For details, see page 169.
DATE TIME	DATE FORMAT TIME FORMAT LOCAL DATE LENGTH LOCAL TIME LENGTH	Formatting of datetime strings. For details, see page 170.
STDSQL		Conformance with the SQL standard. For details, see page 170.
NOFOR or STDSQL		Whether the FOR UPDATE OF clause must be specified (in the SELECT statement of the DECLARE CURSOR statement). For details, see page 172.
CONNECT		Whether the rules for a type 1 or a type 2 CONNECT statement apply. See “CONNECT” on page 446 for a description of the rules.
	CURRENTSERVER bind option	Establishing a server other than the local DB2 subsystem. For details, see “Establishing a different server” on page 448.
	SQLRULES bind option	Whether a type 2 CONNECT statement is processed with DB2 rules or SQL standard rules.
	CURRENT RULES special register	Whether the statements ALTER TABLE, CREATE TABLE, GRANT, and REVOKE are processed with DB2 rules or SQL standard rules. For details, see “CURRENT RULES” on page 109.
		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 on page 591.
		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 570.
#		Whether a stored procedure runs as a main or subprogram. For details, see “CREATE PROCEDURE (external)” on page 541.
#		
#		
#		
	SQLRULES bind option or CURRENT RULES special register	Whether SQLCODE +236 is issued when the SQLDA provided on DESCRIBE or PREPARE INTO is too small and the result columns do not involve LOBs or distinct types. For details, see “DESCRIBE (prepared statement or table)” on page 659 and Appendix C, “SQLCA and SQLDA” on page 883.
		Whether the SELECT privilege is required in a searched DELETE or UPDATE. For details, see “DELETE” on page 653 or “UPDATE” on page 833.

Options affecting SQL

Table 22 (Page 3 of 3). Summary of items affecting composition and processing of SQL statements

Precompiler option	Other ¹	Affects
DEC	DECIMAL ARITHMETIC or CURRENT PRECISION special register	Whether DEC15 or DEC31 rules are used when both operands in a decimal operation have 15 digits or less. For details, see “Arithmetic with two decimal operands” on page 134.

Note: ¹The entries in this column are fields on installation panels unless otherwise noted.

For further details on precompiler options, see Section 6 of *DB2 Application Programming and SQL Guide*. For more details on bind options, see Chapter 2 of *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, the following precompiler options are used instead of the application programming defaults:

- COMMA or PERIOD
- APOST or QUOTE
- APOSTSQL or QUOTESQL
- GRAPHIC or NOGRAPHIC
- 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 bind option DYNAMICRULES.

For additional information on the effect of precompiler options and application programming defaults on:

- Decimal point representation, see page 166.
- String delimiters, see page 168.
- Mixed data, see page 169.
- Decimal arithmetic, see “Arithmetic with two decimal operands” on page 134.

For a list of the DYNAMICRULES bind option values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.

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 application 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-COBL 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 bind option values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.

- 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 constant, a comma intended as a separator must be followed by 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 in string delimiters

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 application 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 bind option values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.

- 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

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

The field MIXED DATA on installation panel DSNTIPF can have the value YES or NO. The value YES indicates that character strings can contain a mixture of SBCS and DBCS characters. The value NO indicates that they cannot. A corresponding precompiler option (GRAPHIC or NOGRAPHIC) exists for every host language supported.

For static SQL statements, the value of the precompiler option determines whether character strings can contain mixed data. For dynamic SQL statements, either the value of field MIXED DATA or the precompiler option is used, depending on the value of bind option DYNAMICRULES in effect:

- If DYNAMICRULES run behavior applies, field MIXED DATA is used.
 - For a list of the DYNAMICRULES bind option values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.
- If bind, define, or invoke behavior applies and the value of install panel field USE FOR DYNAMICRULES is YES, field MIXED DATA is used. If USE FOR DYNAMICRULES is NO, the precompiler option is used.

The value of MIXED DATA and the precompiler option 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 131. 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: EBCDIC and ASCII ones for SBCS data, EBCDIC and ASCII ones for MIXED data, and EBCDIC and ASCII ones for GRAPHIC (DBCS) data. The CCSIDs for SBCS and DBCS data are derived from the value of the ASCII CODED CHAR SET or EBCDIC CODED CHAR SET field, whose value is a CCSID for MIXED data. 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 system CCSIDs are the EBCDIC and ASCII ones for SBCS data. Therefore, only BIT and SBCS can be data subtypes for character columns.

Formatting of datetime strings

Fields on installation panel DSNTIPF (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 76. 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 78 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)¹⁹ 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 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);
```

In Fortran programs, the variable SQLCOD should be used for SQLCODE, and either SQLSTATE or SQLSTA can be used for SQLSTATE.

Definitions for SQLCA: An SQLCA must not be defined in your program, either by coding its definition manually or by using the INCLUDE SQLCA statement. When

¹⁹ STDSQL(86) is a synonym, but STDSQL(YES) should be used.

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.

Comments in static SQL statements: Static SQL statements can include SQL comments. Two consecutive hyphens (--) indicate that the characters after the hyphens are a comment.

SQL comments are recognized only in a program that has been precompiled with the STDSQL(YES) option. If STDSQL(YES) is not specified, the use of an SQL comment might cause a syntax error. Host language comments can be used instead.

When allowed, SQL comments are subject to the following rules:

- The two hyphens must be on the same line, not separated by a space.
- Comments can be started wherever a space is valid (except within a delimiter token or between EXEC and SQL).
- Comments are terminated by the end of the line.
- Comments are not allowed within statements that are dynamically prepared (using PREPARE or EXECUTE IMMEDIATE).
- Within a statement embedded in a COBOL program, the two hyphens must be preceded by a blank unless they begin a line.

This example shows how to include comments in a statement:

```
EXEC SQL CREATE VIEW PRJ_MAXPER  -- projects with most support personnel
      AS SELECT PROJNO, PROJNAME  -- number and name of project
          FROM DSN8610.PROJ
          WHERE DEPTNO = 'E21'    -- systems support dept code
          AND PRSTAFF > 1
END-EXEC.
```

Positioned updates of columns

The NOFOR precompiler option affects the use of the FOR UPDATE OF 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 23. The NOFOR precompiler option

When NOFOR is in effect	When NOFOR is not in effect
The use of the FOR UPDATE OF 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 OF clause is not specified, positioned updates can be made to any columns that the program has DB2 authority to update.	The FOR UPDATE OF clause must be specified.
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 ease concurrency problems with DBRM libraries.	DBRMs can be built incrementally using the DB2 precompiler.

#

Chapter 4. Built-in functions

A *built-in function* is a function that is supplied with DB2 for OS/390. A built-in function is denoted by a function name followed by one or more operands that are enclosed in parentheses. The operands are called *arguments*, and each argument is specified by an *expression*. The result of a built-in function is a single value derived by applying the operation of the function to the arguments.

The built-in functions are in schema SYSIBM. A built-in function can be invoked with or without its schema name. Regardless of whether a schema name qualifies the function name, DB2 uses function resolution to determine which function to use. For more information on functions and the process of function resolution, see “Functions” on page 125.

Built-in functions are classified as *column functions* or *scalar functions*. Although both types of functions return a single value, their arguments differ. The argument of a column function is a set of like values. Each argument of a scalar function is a single value.

A built-in function can be used where an expression can be used; however, some restrictions apply to the use of column functions as specified in “Column functions” on page 178 and in “Chapter 5. Queries” on page 307. Most of the built-in functions can also be used as the source function for a user-defined function as described under “CREATE FUNCTION (sourced)” on page 508.

Table 24 lists the built-in functions that DB2 supports.

Table 24 (Page 1 of 5). Supported functions

Function name	Description	Page
ABS or ABSVAL	Returns the absolute value of its argument	188
ACOS	Returns the arccosine of an argument as an angle, expressed in radians	189
ASIN	Returns the arcsine of an argument as an angle, expressed in radians	190
ATAN	Returns the arctangent of an argument as an angle, expressed in radians	191
ATANH	Returns the hyperbolic arctangent of an argument as an angle, expressed in radians	192
ATAN2	Returns the arctangent of <i>x</i> and <i>y</i> coordinates as an angle, expressed in radians	193
AVG	Returns the average of a set of numbers	179
BLOB	Returns a BLOB representation of its argument	194
CEIL or CEILING	Returns the smallest integer greater than or equal to the argument	195
CHAR	Returns a fixed-length character string representation of its argument	196
CLOB	Returns a CLOB representation of its argument	202
COALESCE	Returns the first argument in a set of arguments that is not null	203

Built-in functions

Table 24 (Page 2 of 5). Supported functions

Function name	Description	Page
CONCAT	Returns the concatenation of two strings	205
COS	Returns the cosine of an argument that is expressed as an angle in radians	206
COSH	Returns the hyperbolic cosine of an argument that is expressed as an angle in radians	207
COUNT	Returns the number of rows or values in a set of rows or values	180
COUNT_BIG	Same as COUNT, except the result can be greater than the maximum value of an integer	181
DATE	Returns a date derived from its argument	208
DAY	Returns the day part of its argument	209
DAYOFMONTH	Similar to DAY	210
DAYOFWEEK	Returns an integer in the range of 1 to 7, where 1 represents Sunday	211
DAYOFYEAR	Returns an integer in the range of 1 to 366, where 1 represents January 1	212
DAYS	Returns an integer representation of a date	213
DBCLOB	Returns a DBCLOB representation of its argument	214
DECIMAL or DEC	Returns a decimal representation of its argument	215
DEGREES	Returns the number of degrees for an argument that is expressed in radians	217
DIGITS	Returns a character string representation of a number	218
DOUBLE or DOUBLE-PRECISION	Returns a double precision floating-point representation of its argument	219
EXP	Returns the exponential function of an argument	220
FLOAT	Same as DOUBLE	219
FLOOR	Returns the largest integer that is less than or equal to the argument	222
GRAPHIC	Returns a GRAPHIC representation of its argument	223
HEX	Returns a hexadecimal representation of its argument	225
hour	Returns the hour part of its argument	226
# IDENTITY_VAL_LOCAL #	Returns the most recently assigned value for an identity column	227
IFNULL	Returns the first argument in a set of two arguments that is not null	232
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	233
INTEGER or INT	Returns an integer representation of its argument	236
JULIAN_DAY	Returns an integer that represents the number of days from January 1, 4712 B.C.	237

Table 24 (Page 3 of 5). Supported functions

Function name	Description	Page
LCASE or LOWER	Returns a string with the characters converted to lowercase	238
LEFT	Returns a string that consists of the specified number of leftmost bytes of a string	239
LENGTH	Returns the length of its argument	241
LN	Returns the natural logarithm of an argument	242
LOCATE	Returns the position at which the first occurrence of an argument starts within another argument	243
LOG10	Returns the base 10 logarithm of an argument	245
LTRIM	Returns the characters of a string with the leading blanks removed	246
MAX	Returns the maximum value in a set of values	182
MICROSECOND	Returns the microsecond part of its argument	247
MIDNIGHT_SECONDS	Returns an integer in the range of 0 to 86400 that represents the number of seconds between midnight and the argument	248
MIN	Returns the minimum value in a set of values	183
MINUTE	Returns the minute part of its argument	249
MOD	Returns the remainder of one argument divided by a second argument	250
MONTH	Returns the month part of its argument	252
# MULTIPLY_ALT # #	Returns the product of the two arguments as a decimal value, used when the sum of the argument precisions exceeds 31	253
NULLIF	Returns NULL if the arguments are equal; else the first argument	254
POSSTR	Returns the position of the first occurrence of an argument within another argument	255
POWER	Returns the value of one argument raised to the power of a second argument	257
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	258
RADIANS	Returns the number of radians for an argument that is expressed in degrees	259
RAISE_ERROR	Raises an error in the SQLCA with the specified SQLSTATE and error description	260
RAND	Returns a double precision floating-point random number	261
REAL	Returns a single precision floating-point representation of its argument	262
REPEAT	Returns a character string composed of an argument repeated a specified number of times	263

Built-in functions

Table 24 (Page 4 of 5). Supported functions

Function name	Description	Page
REPLACE	Returns a string in which all occurrences of an argument within a second argument are replaced with a third argument	265
RIGHT	Returns a string that consists of the specified number of rightmost bytes of a string	267
ROUND	Returns a number rounded to the specified number of places to the right or left of the decimal place	269
ROWID	Returns a row ID representation of its argument	271
RTRIM	Returns the characters of an argument with the trailing blanks removed	272
SECOND	Returns the second part of its argument	273
SIGN	Returns the sign of an argument	274
SIN	Returns the sine of an argument that is expressed as an angle in radians	275
SINH	Returns the hyperbolic sine of an argument that is expressed as an angle in radians	276
SMALLINT	Returns a small integer representation of its argument	277
SPACE	Returns a string that consists of the number of blanks the argument specifies	278
SQRT	Returns the square root of its argument	279
STDDEV	Returns the standard deviation of a set of numbers	184
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	280
SUBSTR	Returns a substring of a string	282
SUM	Returns the sum of a set of numbers	185
TAN	Returns the tangent of an argument that is expressed as an angle in radians	284
TANH	Returns the hyperbolic tangent of an argument that is expressed as an angle in radians	285
TIME	Returns a time derived from its argument	286
TIMESTAMP	Returns a timestamp derived from its arguments	287
# # # TIMESTAMP_FORMAT	Returns a timestamp for a character string expression, using a specified format to interpret the string	289
TRANSLATE	Returns a string with one or more characters translated	290
TRUNCATE or TRUNC	Returns a number truncated to the specified number of places to the right or left of the decimal point	293
# # UCASE or UPPER	Returns a string with the characters converted to uppercase	294
VARCHAR	Returns the varying-length character string representation of its argument	295

Table 24 (Page 5 of 5). Supported functions

	Function name	Description	Page
#	VARCHAR_FORMAT	Returns a character string representation of a timestamp, with the string in a specified format	299
#	VARGRAPHIC	Returns a graphic string representation of its argument	301
	VARIANCE or VAR	Returns the variance of a set of numbers	186
	WEEK	Returns an integer that represents the week of the year	304
	YEAR	Returns the year part of its argument	305

Column functions

The following information applies to all built-in column functions, except for the COUNT(*) and COUNT_BIG(*) variations of the COUNT and COUNT_BIG functions.

The argument of a column function is a set of values derived from an expression. The expression must include a column name and must not include another column function. The scope of the set is a group or an intermediate result table as explained in “Chapter 5. Queries” on page 307. 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 DSN8610.EMP
WHERE WORKDEPT = 'D11';
```

The keyword DISTINCT is not considered an argument of the function but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, duplicate values are eliminated. If ALL is implicitly or explicitly specified, duplicate values are not eliminated.

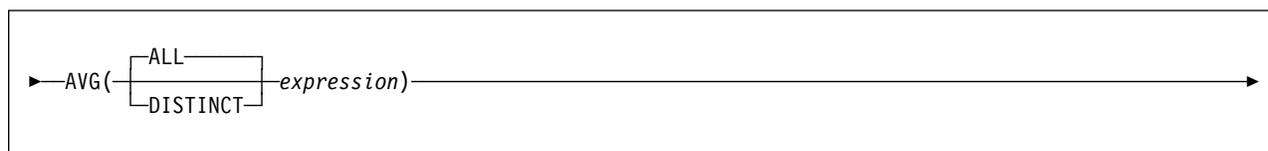
A column function can be used in a WHERE clause only if that clause is part of a subquery of a HAVING clause and the column name specified in the expression is a correlated reference to a group. If the expression includes more than one column name, each column name must be a correlated reference to the 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 a column 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.

Following in alphabetic order is a definition of each of the built-in column functions.

AVG



The schema is SYSIBM.

The AVG function returns the average of a set of numbers.

The argument values must 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 the same as the data type of the argument values, except that the result is a large integer if the argument values are small integers, and the result is double precision floating-point if the argument values are single precision floating-point. 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, duplicate values are also eliminated.

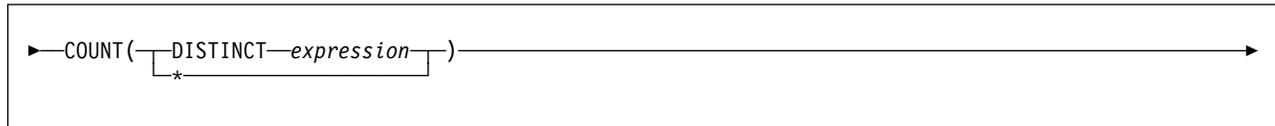
If the function is applied to an empty set, the result is the null value. Otherwise, the result is the average value of the set. If the type of the result is an integer, the fractional part of the average is lost. The order in which the summation part of the operation is performed is undefined but every intermediate result must be within the range of the result data type.

Example: Assuming DEC15, set the DECIMAL(15,2) variable AVERAGE to the average salary in department D11 of the employees in the sample table DSN8610.EMP.

```
EXEC SQL SELECT AVG(SALARY)
        INTO :AVERAGE
        FROM DSN8610.EMP
        WHERE WORKDEPT = 'D11';
```

COUNT

COUNT



The schema is SYSIBM.

The COUNT function returns the number of rows or values in a set of rows or values.

The argument values can be of any built-in data type other than a CLOB, DBCLOB, or BLOB. Character string arguments cannot have a maximum length greater than 255, and graphic string arguments cannot have a maximum length greater than 127.

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(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 duplicate values. The result is the number of values in the set.

Example 1: Set the integer host variable FEMALE to the number of females represented in the sample table DSN8610.EMP.

```
EXEC SQL SELECT COUNT(*)
        INTO :FEMALE
        FROM DSN8610.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 DSN8610.EMP
        WHERE SEX = 'F';
```

COUNT_BIG

▶ COUNT_BIG([DISTINCT] expression)

The schema is SYSIBM.

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 argument values can be of any built-in data type other than a CLOB, DBCLOB, or BLOB. Character string arguments cannot have a maximum length greater than 255, and graphic string arguments cannot have a maximum length greater than 127.

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(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 duplicate values. The result is the number of different nonnull values in the set.

Example 1: The examples for COUNT are also applicable for COUNT_BIG. Refer to the COUNT examples and substitute COUNT_BIG for the occurrences of COUNT. The results are the same except for the data type of the result.

Example 2: To count on a specific column, a sourced function must specify the type of the column. In this example, the CREATE FUNCTION statement creates a sourced function that takes any column defined as CHAR, uses COUNT_BIG to perform the counting, and returns the result 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 DSN8610.EMP;
```

Note that the input parameter for the sourced function is defined with empty parentheses to indicate that the input parameter inherits the attributes of the parameter of the source function.

MAX

MAX



The schema is SYSIBM.

The MAX function returns the maximum value in a set of values.

The argument values can be of any built-in data type other than a CLOB, DBCLOB, BLOB, or row ID. Character string arguments cannot have a maximum length greater than 255, and graphic string arguments cannot have a maximum length greater than 127.

The data type of the result and its other attributes (for example, the length and CCSID of a string) 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.

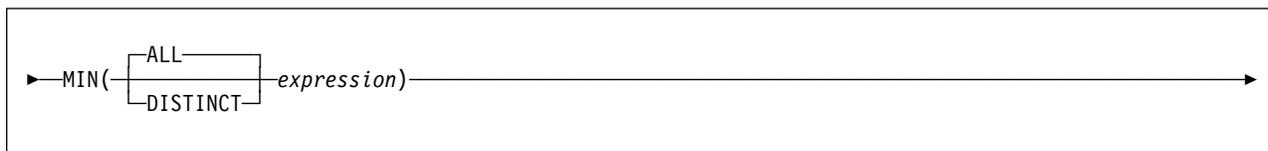
Example 1: Set the DECIMAL(8,2) variable MAX_SALARY to the maximum monthly salary of the employees represented in the sample table DSN8610.EMP.

```
EXEC SQL SELECT MAX(SALARY) / 12
        INTO :MAX_SALARY
        FROM DSN8610.EMP;
```

Example 2: Find the surname that comes last in the collating sequence for the employees represented in the sample table DSN8610.EMP. Set the VARCHAR(15) variable LAST_NAME to that surname.

```
EXEC SQL SELECT MAX(LASTNAME)
        INTO :LAST_NAME
        FROM DSN8610.EMP;
```

MIN



The schema is SYSIBM.

The MIN function returns the minimum value in a set of values.

The argument values can be of any built-in data type other than a CLOB, DBCLOB, BLOB, or row ID. Character string arguments cannot have a maximum length greater than 255, and graphic string arguments cannot have a maximum length greater than 127.

The data type of the result and its other attributes (for example, the length and CCSID of a string) 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.

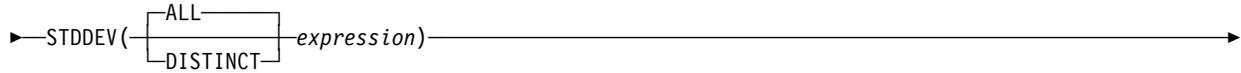
Example 1: Set the DECIMAL(15,2) variable MIN_SALARY to the minimum monthly salary of the employees represented in the sample table DSN8610.EMP.

```
EXEC SQL SELECT MIN(SALARY) / 12
          INTO :MIN_SALARY
          FROM DSN8610.EMP;
```

Example 2: Find the surname that comes first in the collating sequence for the employees represented in the sample table DSN8610.EMP. Set the VARCHAR(15) variable FIRST_NAME to that surname.

```
EXEC SQL SELECT MIN(LASTNAME)
          INTO :FIRST_NAME
          FROM DSN8610.EMP;
```

STDDEV



The schema is SYSIBM.

The STDDEV function returns the standard deviation (\sqrt{n}) of a set of numbers. The formula that is used to calculate STDDEV is:

$$\text{STDDEV} = \text{SQRT}(\text{VAR})$$

where SQRT(VAR) is the square root of the variance.

The argument values must each be the value of any built-in numeric data type, and their sum must be within the range of the data type of the result.

The result of the function is double precision floating-point number. 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, 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.

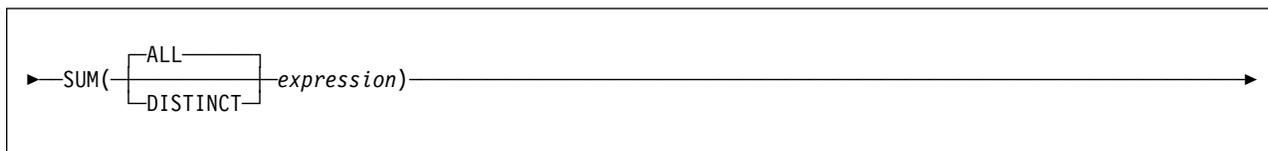
The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example: Using sample table DSN8610.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 DSN8610.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 schema is SYSIBM.

The SUM function returns the sum of a set of numbers.

The argument values must 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 the same as the data type of the argument values, except that the result is a large integer if the argument values are small integers, and the result is double precision floating-point if the argument values are single precision floating-point. 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, 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 DSN8610.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 DSN8610.EMP;
```

Scalar functions

A built-in scalar function can be used wherever an expression can be used. However, the restrictions that apply to the use of expressions and column functions also apply when an expression or column function is used within a scalar function. For example, the argument of a scalar function can be a column function only if a column 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.

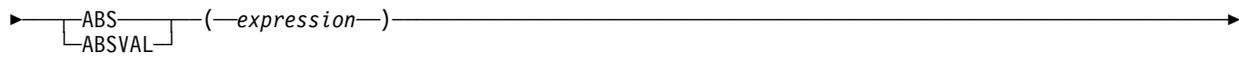
Example: The following SELECT statement calls for the employee number, last name, and age of each employee in department D11 in the sample table DSN8610.EMP. To obtain the ages, the scalar function YEAR is applied to the expression:

```
CURRENT DATE - BIRTHDATE
```

in each row of DSN8610.EMP for which the employee represented is in department D11:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BIRTHDATE)
FROM DSN8610.EMP
WHERE WORKDEPT = 'D11';
```

Following in alphabetic order is the definition of each of the built-in scalar functions.

ABS or ABSVAL

The schema is SYSIBM.

The ABS function returns the absolute value of the argument.

The argument is 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.

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 schema is SYSIBM.

The ACOS function returns the arccosine of the argument as an angle expressed in radians. The ACOS and COS functions are inverse operations.

The argument is an expression whose value is a number in the range of -1 to 1, and has any built-in numeric data type. 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.

ASIN

ASIN(*expression*)

The schema is SYSIBM.

The ASIN function returns the arcsine of the argument as an angle expressed in radians. The ASIN and SIN functions are inverse operations.

The argument is an expression whose value is a number in the range of -1 to 1, and has any built-in numeric data type. 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 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



▶—ATAN(*expression*)—▶

The schema is SYSIBM.

The ATAN function returns the arctangent of the argument as an angle expressed in radians. The ATAN and TAN functions are inverse operations.

The argument is an expression that returns the value of any built-in numeric data type. 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 ATANGENT is DECIMAL(10,9) with a value of 14.10141995. The following statement:

```
SELECT ATAN(:ATANGENT)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 1.50.

ATANH

▶—ATANH(*expression*)—▶

The schema is SYSIBM.

The ATANH function returns the hyperbolic arc tangent of the argument as an angle expressed in radians. The ATANH and TANH functions are inverse operations.

The argument is an expression whose value is a number in the range of -1 to 1, and has any built-in numeric data type. 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:

```
SELECT ATANH(:HATAN)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 1.50.

ATAN2



▶—ATAN2(*expression1*,*expression2*)—▶

The schema is SYSIBM.

The ATAN2 function returns the arctangent of *x* and *y* coordinates as an angle expressed in radians. The first and second arguments specify the *x* and *y* coordinates, respectively.

Each argument is an expression that returns the value of any built-in numeric data type. 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:

```
SELECT ATAN2 (:HATAN2A, :HATAN2B)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 1.1071487.

BLOB

The schema is SYSIBM.

The BLOB function returns a BLOB representation of a string of any type or a row ID type.

expression

An expression whose value is a character string, graphic string, binary string, or a row ID type.

integer

An integer value specifying the length attribute of the resulting BLOB data type. The value must be an integer between 0 and the maximum length of a BLOB.

Do not specify *integer* if *expression* is a row ID type.

If you do not specify *integer*, the length attribute of the result is the same as the length attribute of *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 *expression* (or twice the length of *expression* when the input is graphic data). If the length of *expression* is greater than the length specified, 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;
```

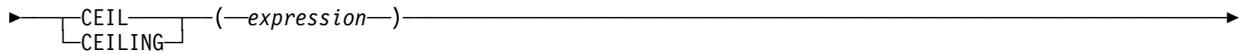
Example 2: The following function returns a BLOB for the large object that is identified by locator `myclob_locator`.

```
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 'Pellow 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('%Pellow Island%')
```

CEIL or CEILING



The schema is SYSIBM.

The CEIL or CEILING function returns the smallest integer value that is greater than or equal to the argument.

The argument is 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.

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;
```

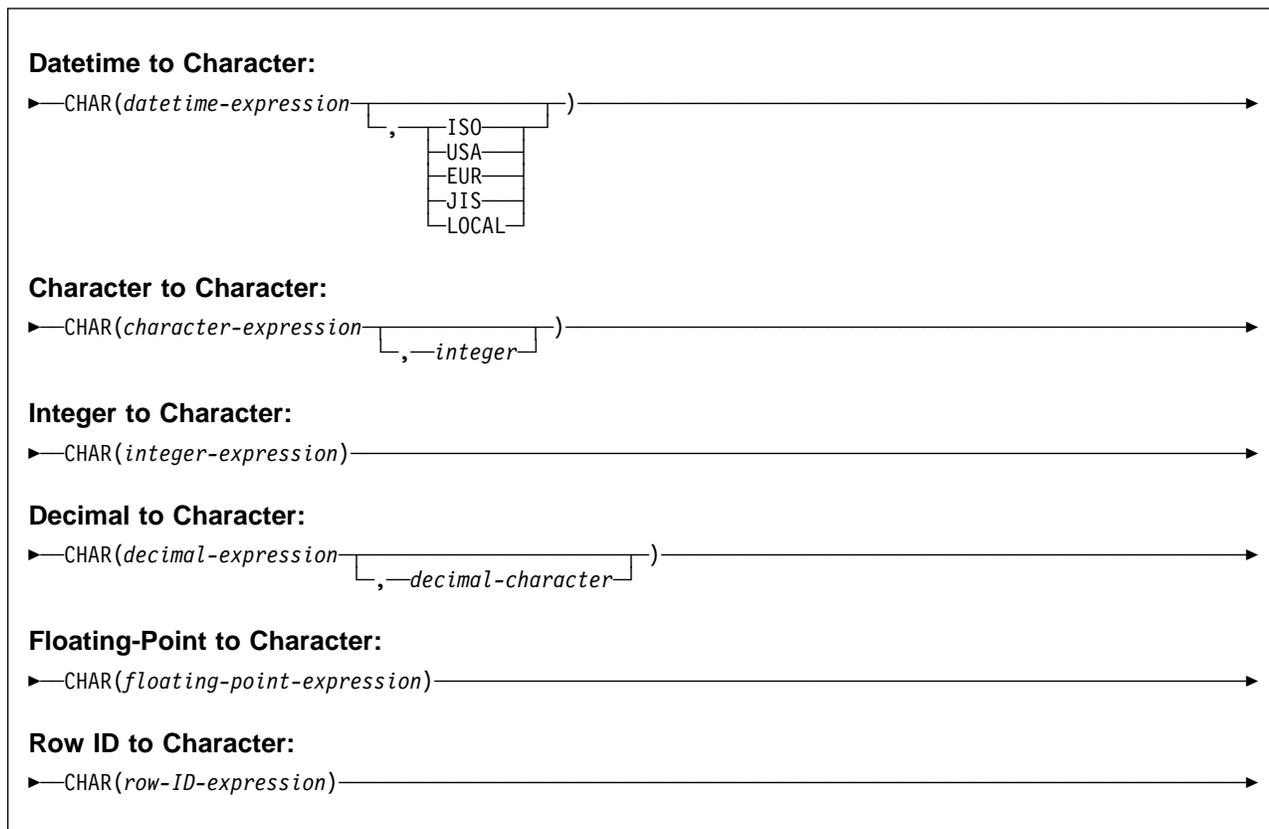
This example returns: 4., 4., -3., -3.

Example 2: Using sample table DSN8610.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 CEIL(MAX(SALARY)/12)
FROM DSN8610.EMP;
```

This example returns 4396. because the highest paid employee is Christine Haas who earns \$52750.00 per year. Her average monthly salary before applying the CEIL function is 4395.83.

CHAR



The schema is SYSIBM.

The CHAR function returns a fixed-length character string representation of one of the following values:

- Datetime value if the first argument is a date, time, or timestamp
- Character string value if the first argument is any type of character string
- Integer number if the first argument is a small or large integer
- Decimal number if the first argument is a decimal number
- Floating-point number if the first argument is a single or double precision floating-point number
- 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.

The CCSID of the resulting string depends on the rest of the data referenced in the statement that contains the CHAR function. If the statement references ASCII data, the CCSID of the resulting string is the ASCII CCSID of the server. Otherwise, the CCSID is the EBCDIC CCSID of the server.

Datetime to Character

datetime-expression

An expression whose value has one of the following three 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, 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 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 date 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 the character string representation of the time in the format specified by the second argument. If the second argument is omitted, 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 of the result is 8.

LOCAL denotes the local format at the DB2 that executes the SQL statement. If LOCAL is used for the format, a time exit routine must be installed at that DB2.

An error occurs if the second argument is specified and is not a valid value.

timestamp The result is the character string representation of the timestamp. The length of the result is 26. The second argument must not be specified.

The CCSID of the result is the SBCS CCSID of the server.

Character to Character

character-expression

An expression whose value is any type of character string.

integer

The length attribute for the resulting fixed-length character string. The value must be between 1 and 255 . If the length is not specified, the length of the result is the same as the length of *character-expression*, which must be 255 bytes or less but must not be an empty string.

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 an empty string, an error occurs.

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*.

Integer to Character

integer-expression

An expression whose value is an integer data type (SMALLINT or INTEGER).

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 with a preceding minus sign if the argument is negative. 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 right with blanks to a length of 11.

A positive value always includes one trailing blank.

The CCSID of the result is the SBCS CCSID of the server.

Decimal to Character

decimal-expression

An expression whose value is a decimal data type. To specify a different precision and scale, you can use the DECIMAL scalar function first to make the change.

decimal-character

Specifies the single-byte character constant (CHAR or VARCHAR only) that is used to delimit the decimal digits in the resulting character string. Do not specify a digit, plus ('+'), minus ('-') or blank. The default is the period ('.') or comma (','). For information on what factors govern the choice, see "Options affecting SQL" on page 164.

The result is the fixed-length character string representation of the argument in the form of an SQL decimal constant. The result includes a *decimal-character* and p digits, where p is the precision of the *decimal-expression*. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a blank, which means that a positive value always has one leading blank.

The length of the result is $2+p$, where p is the precision of the *decimal-expression*.

The CCSID of the result is the SBCS CCSID of the server.

Floating-Point to Character

floating-point-expression

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

The result is the fixed-length character string representation of the argument in the form of a 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 the SBCS CCSID of the server.

Row ID to Character*row-ID-expression*

An expression whose value is a row ID data type.

The result is the fixed-length character string representation of the argument. It is bit data and does not have an associated CCSID.

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 zeroes to length of 40.

Example 1: HIREDATE is a DATE column in sample table DSN8610.EMP. When it represents 15 December 1976 (as it does for employee 140):

```
EXEC SQL SELECT CHAR(HIREDATE, USA)
          INTO :DATESTRING
          FROM DSN8610.EMP
          WHERE EMPNO = '000140';
```

returns the string value '12/15/1976' in character-string variable DATESTRING.

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:

```
CHAR(STARTING+:HOURS, USA)
```

returns the value '10:30 PM'.

Example 3: Assume that RECEIVED is defined as a TIMESTAMP column in table TABLEY. When the value of the date portion of RECEIVED represents 10 March 1997 and the time portion represents 6 hours and 15 seconds after midnight, this example:

```
SELECT CHAR(RECEIVED)
       FROM TABLEY
       WHERE INTCOL = 1234;
```

returns the string value '1997-03-10-06.00.15.000000'.

CHAR

Example 4: For sample table DSN8610.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 DSN8610.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 DSN8610.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 displayed results to 10 characters.

```
SELECT CHAR(LASTNAME,10)
        FROM DSN8610.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: For the rows in sample table DSN8610.EMP, return the values in column EDLEVEL, which is defined as SMALLINT, as a fixed-length character string.

```
SELECT CHAR(EDLEVEL)
        FROM DSN8610.EMP;
```

An EDLEVEL of 18 is returned as CHAR(6) value '18 ' (18 followed by four blanks).

Example 7: In sample table DSN8610.EMP, the SALARY column is returned as DECIMAL(9,2). For those employees who have a salary of 18357.50, return the hire date and the salary, using a comma as the decimal character in the salary (18357,50).

```
SELECT HIREDATE, CHAR(SALARY, ',')
        FROM DSN8610.EMP
        WHERE SALARY = 18357.50;
```

The salary is returned as the string value '00018357,50'.

Example 8: Repeat the scenario in Example 7 except subtract the SALARY column from 20000.25 and return the salary with the default decimal character.

```
SELECT HIREDATE, CHAR (20000.25 - SALARY)
        FROM DSN8610.EMP
        WHERE SALARY = 18357.50;
```

The salary is displayed as the string value '0001642.75'.

Example 9: 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 10:* 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 '.

CLOB

▶ CLOB(*expression* [, *integer*])

The schema is SYSIBM.

The CLOB function returns a CLOB representation of a string.

expression

An expression whose value is a string. If *expression* is bit data, an error occurs.
Use a BLOB data type for bit data.

integer

An integer value specifying the length attribute of the resulting CLOB data type.
The value must be between 0 and the maximum length of a CLOB.

If you do not specify *integer*, the length attribute of the result is the same as the length attribute of *expression*.

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 *expression*. If the length of *expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are blanks, a warning is returned.

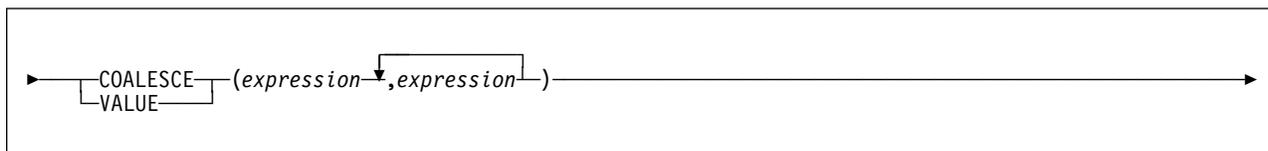
The subtype and CCSID of the result are determined as follows:

- # • If *expression* is character SBCS data, the result is SBCS data and the CCSID is the CCSID for the encoding scheme of the SQL statement.
- # • If the first argument is mixed data, the result is mixed data and the CCSID is the CCSID for the encoding scheme of the SQL statement.

Example: The following function returns a CLOB for the string 'This is a CLOB'.

```
SELECT CLOB('This is a CLOB')
FROM SYSIBM.SYSDUMMY1;
```

COALESCE



The schema is SYSIBM.

The COALESCE function returns the first argument that is not null. VALUE can be used as a synonym for COALESCE. Use COALESCE to conform to the SQL standard.

The arguments must be compatible. For more information on compatibility, refer to the compatibility matrix in Table 9 on page 85. 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 99. 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
```

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 DSN8610.EMP contains a DATE column named HIREDATE, and that nulls are allowed for this column. The following query selects all rows in DSN8610.EMP for which the date in HIREDATE is either unknown (null) or earlier than 1 January 1960.

```
SELECT * FROM DSN8610.EMP
WHERE VALUE(HIREDATE,DATE('1959-12-31')) < '1960-01-01';
```

²⁰ This function cannot be used as a source function when creating a user-defined function. Because it accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

COALESCE

The predicate could also be coded as `VALUE(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.

CONCAT

Note:

¹ The vertical bars (||) must be surrounded by the SQL escape character in effect (" or ').

The schema is SYSIBM.

You can use either the keyword CONCAT keyword or vertical bars (||) to invoke the concatenation function. Vertical bars (or the characters that must be used in place of vertical bars in some 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. Thus, CONCAT is the preferable syntax.

The CONCAT function, which is identical to the CONCAT operator, links two string arguments to form a string expression. The two arguments must be compatible strings. For more information on compatibility, refer to the compatibility matrix in Table 9 on page 85.

The result of the function is a string. If either argument can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first argument string followed by the second.

See “With the concatenation operator” on page 131 for more information.

Example: Using sample table DSN8610.EMP, concatenate column FIRSTNME with column LASTNAME. Both columns are defined as varying-length character strings.

```
SELECT CONCAT(FIRSTNME, LASTNAME)
FROM DSN8610.EMP;
```

COS

COS



► COS(*expression*) ◄

The schema is SYSIBM.

The COS function returns the cosine of the argument, where the argument is an angle expressed in radians. The COS and ACOS functions are inverse operations.

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

```
SELECT COS(:COSINE)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 0.07.

COSH



► COSH(*expression*) ◄

The schema is SYSIBM.

The COSH function returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.

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

```
SELECT COSH(:HCOS)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 2.35.

DATE



The schema is SYSIBM.

The DATE function returns a date derived from its argument.

The argument must be a date, a timestamp, a valid character string representation of a date or timestamp, a positive number with a built-in data type that is less than or equal to 3652059, or a character string of length 7. An argument with a character data type must not be a CLOB 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 76.)

If the argument is a character string of length 7, it must represent a valid date in the form *yyyynn*, where *yyyy* are digits denoting a year, and *nnn* are digits between 001 and 366 denoting a day of that year.

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

The other rules depend on the data type of the argument:

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 character string, the result is the date represented by the character string. If the CCSID of the string is not the same as the corresponding default CCSID at the application server, the string is first converted to that 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'. Then, for this value:

```
DATE(RECEIVED)
```

returns the internal representation of 25 December 1988.

Example 2: Assume that DATCOL is a CHAR(7) column in some table, and that one of its values is the character string '1989061'. Then, for this value:

```
DATE(DATCOL)
```

returns the internal representation of 2 March 1989.

Example 3: DB2 recognizes '1989-03-02' as the ISO representation of 2 March 1989. Therefore:

```
DATE('1989-03-02')
```

returns the internal representation of 2 March 1989.

DAY



The schema is SYSIBM.

The DAY function returns the day part of its argument.

The argument must be a date, timestamp, date duration, timestamp duration, or valid character string representation of a date or timestamp. A character string representation must not be a CLOB 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 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 other rules for the function depend on the data type of the argument:

If the argument is a date, timestamp, or character 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 DSN8610.EMP was hired.

```
EXEC SQL SELECT DAY(HIREDATE)
          INTO :DAYVAR
          FROM DSN8610.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

►—DAYOFMONTH(*expression*)—►

The schema is SYSIBM.

The DAYOFMONTH function returns the day part of its argument. The function is similar to the DAY function, except DAYOFMONTH does not support a date or timestamp duration as an argument.

The argument must be a date, a timestamp, or a valid character string representation of a date or timestamp. A character string representation must not be a CLOB 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 76.)

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: Set the INTEGER variable DAYVAR to the day of the month on which employee 140 in sample table DSN8610.EMP was hired.

```
SELECT DAYOFMONTH(HIREDATE)
       INTO :DAYVAR
       FROM DSN8610.EMP
       WHERE EMPNO = '000140';
```

DAYOFWEEK

DAYOFWEEK(*expression*)

The schema is SYSIBM.

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 argument must be a date, a timestamp, or a valid character string representation of a date or timestamp. A character string representation must not be a CLOB 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 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.

Example 1: Using sample table DSN8610.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 DSN8610.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;
```

DAYOFYEAR

► DAYOFYEAR(*expression*) ◀

The schema is SYSIBM.

The DAYOFYEAR function returns an integer in the range of 1 to 366 that represents the day of the year where 1 is January 1.

The argument must be a date, a timestamp, or a valid character string representation of a date or timestamp. A character string representation must not be a CLOB 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 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.

Example: Using sample table DSN8610.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 DSN8610.EMP;
```

The result is that AVG_DAY_OF_YEAR is set to 202.

DAYS



The schema is SYSIBM.

The DAYS function returns an integer representation of a date.

The argument must be a date, a timestamp, or a valid string representation of a date or timestamp. A character string representation must not be a CLOB 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 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 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.

Example: 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 DSN8610.EMP
           WHERE EMPNO = '000140';
```

DBCLOB

► DBCLOB(*expression* [, *integer*])

The schema is SYSIBM.

The DBCLOB function returns a DBCLOB representation of a graphic string type.

The length of the result is measured in double-byte characters.

expression

An expression whose value is a graphic string.

integer

An integer value specifying the length attribute of the resulting DBCLOB. The value must be between 0 and the maximum length of a DBCLOB.

If you do not specify *integer*, the length attribute of the result is the same as the length attribute of *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.

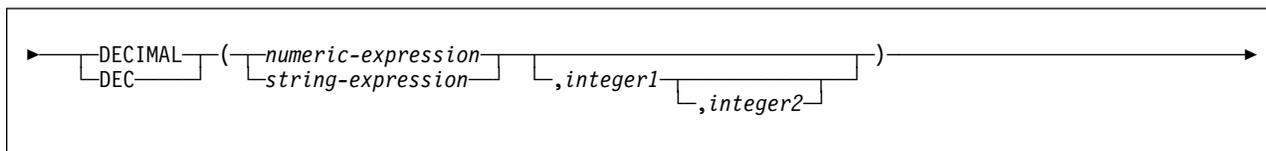
The actual length of the result is the minimum of the length attribute of the result and the actual length of *expression*. If the length of *expression* is greater than the length specified, the result is truncated. Unless all of the truncated characters are double-byte blanks, a warning is returned.

The CCSID of the result is the same as the CCSID of *expression*.

Example: Assume that GRAPHCOL contains a varying-length graphic string (VARGRAPHIC). The following function returns the string in GRAPHCOL as a DBCLOB.

```
SELECT DBCLOB(GRAPHCOL)
FROM SYSIBM.SYSDUMMY1;
```

DECIMAL or DEC



The schema is SYSIBM.

The DECIMAL or DEC function returns a decimal representation of a number or character string in the form of a numeric constant.

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result of the function is a decimal number. The result 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.

string-expression

An expression that returns any type of character string, except a CLOB, with a maximum length that is not greater than 255 bytes. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming an SQL integer or decimal constant.

The result of the function is a decimal number. The result is the same number that would occur if the corresponding integer or decimal constant 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.

integer1

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

integer2

An integer constant with a value in the range of 1 to p , where p is the value of the second argument. The value of this third argument specifies the scale of the result. The default value is 0.

The data type of the result is DECIMAL(p,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.

An error occurs if the number of significant digits required to represent the whole part of the number is greater than $p-s$.

DECIMAL or DEC

Example: Represent the average salary of the employees in DSN8610.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 DSN8610.EMP;
```

DEGREES



▶—DEGREES(*expression*)—▶

The schema is SYSIBM.

The DEGREES function returns the number of degrees converted from the argument expressed in radians.

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

```
SELECT DEGREES(:HRAD)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 180.0.

DIGITS

The schema is SYSIBM.

The DIGITS function returns a character string representation of its argument.

The argument is an expression that returns the value of any 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
- p if the argument is a decimal number with a precision of p

The CCSID of the resulting string depends on the rest of the data referenced in the statement that contains the DIGITS function. If the statement references ASCII data, the CCSID of the resulting string is the ASCII CCSID of the server. Otherwise, the CCSID is the EBCDIC CCSID of the server.

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. List all combinations of the first four digits in column INTCOL.

```
SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
FROM TABLEX;
```

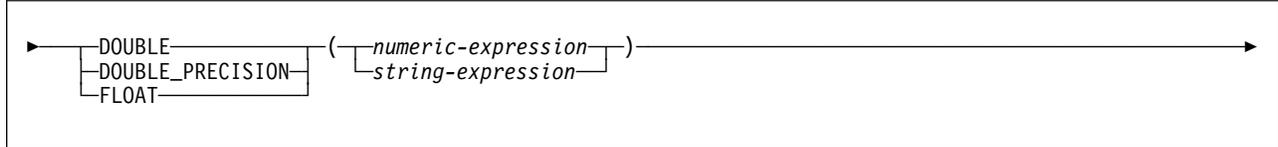
Example 2: Assume that COLUMNX has the data type DECIMAL(6,2), and that one of its values is -6.28. Then, for this value:

```
DIGITS(COLUMNX)
```

the value '000628' is returned.

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 or DOUBLE_PRECISION



The schema is SYSIBM.

The DOUBLE or DOUBLE_PRECISION function returns a double precision floating-point representation of a number or character string in the form of a numeric constant. FLOAT is a synonym for DOUBLE or DOUBLE_PRECISION.

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result of the function is a double precision floating-point number. The result is the same number that would occur if the expression were assigned to a double precision floating-point column or variable.

string-expression

An expression that returns any type of character string, except a CLOB, with an actual length that is not greater than 255 bytes. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming an SQL floating-point constant.

The result of the function is a double precision floating-point number. The result is the same number that would occur if the corresponding numeric constant were assigned to a double precision floating-point column or variable.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Using sample table DSN8610.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 DSN8610.EMP
WHERE COMM > 0;

```

EXP

EXP



► EXP(*expression*)

The schema is SYSIBM.

The EXP function returns the exponential function of the argument (a value that is the base of the natural logarithm (e) raised to a power specified by the argument). The EXP and LOG functions are inverse operations.

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

```
SELECT EXP(:E)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 31.62.

FLOAT



▶—FLOAT(*expression*)—▶

The schema is SYSIBM.

The FLOAT function returns a floating-point representation of its argument.

FLOAT is a synonym for the DOUBLE function. See “DOUBLE or DOUBLE_PRECISION” on page 219 for details.

FLOOR

The schema is SYSIBM.

The FLOOR function returns the largest integer value that is less than or equal to the argument.

The argument is 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. When the argument is DECIMAL, the scale of the result is 0 and not the scale of the input argument.

Example: Using sample table DSN8610.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 DSN8610.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.

GRAPHIC

Character to Graphic:

▶ GRAPHIC(*character-expression* [*,—integer*])

Graphic to Graphic:

▶ GRAPHIC(*graphic-expression* [*,—integer*])

The schema is SYSIBM.

The GRAPHIC function returns a graphic representation of a character string value, with the single-byte characters converted to double-byte characters, or a graphic string value if the first argument is a graphic string.

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 whose value must be an EBCDIC-encoded character string. 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 67 for these rules.)

The expression must not be an empty string or have the value X'0E0F'.

integer

The length of the resulting fixed-length graphic string. The value must be an integer 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 *integer* is not specified, the length of the result is the minimum of 127 and the length attribute of *character-expression*, excluding shift characters.

The CCSID of the result is the system EBCDIC CCSID for GRAPHIC data. If there is no system EBCDIC CCSID for GRAPHIC data, the CCSID of the result is X'FFFE'.

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 EBCDIC 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 EBCDIC CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is M.
- The argument is BIT 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 EBCDIC CCSID for mixed data, conversion is to the coded character set that the system EBCDIC 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'xx' for EBCDIC is X'42xx', except for X'40', whose DBCS equivalent is X'4040'.

Graphic to Graphic

graphic-expression

An expression whose value is a graphic string. The graphic string must not be an empty string.

integer

The length of the resulting fixed-length graphic string. The value must be an integer 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 *integer* is not specified, the length of the result is the minimum of 127 and the length attribute of *graphic-expression*.

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 schema is SYSIBM.

The HEX function returns a hexadecimal representation of its argument.

The argument can be of any built-in data type other than a character or binary string with a maximum length greater than 255 or a graphic string with a maximum length greater than 127.

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 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.

If the argument is not a varying-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 whose maximum length depends on the following considerations:

If the argument is not a varying-length string, the maximum length of the result string is the same as the length of the result.

If the argument is a varying-length character or binary string, the maximum length of the result string is twice the maximum length of the argument.

If the argument is a varying-length graphic string, the maximum length of the result string is four times the maximum length of the argument.

If the maximum length of the result is greater than 254 bytes, the result is subject to the restrictions that apply to long strings.

The CCSID of the resulting string depends on the rest of the data referenced in the statement that contains the HEX function. If the statement references ASCII data, the CCSID of the resulting string is the ASCII CCSID of the server. Otherwise, the CCSID is the EBCDIC CCSID of the server.

The CCSID of the result is the corresponding EBCDIC or ASCII CCSID for SBCS data defined at the server during system installation. The encoding scheme of the argument, EBCDIC or ASCII, determines which CCSID is used.

Example: Return the hexadecimal representation of START_RBA in the SYSIBM.SYSCOPY catalog table.

```
SELECT HEX(START_RBA) FROM SYSIBM.SYSCOPY;
```

HOUR

HOUR



► HOUR(*expression*) ◄

The schema is SYSIBM.

The HOUR function returns the hour part of its argument.

The argument must be a time, timestamp, time duration, timestamp duration, or valid character string representation of a time or timestamp. A character string representation must not be a CLOB and must have 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 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 other rules depend on the data type of the argument:

If the argument is a time, timestamp, or character string representation of either, the result is the hour part of the value, which is an integer between 0 and 24.

If the argument is a time duration or timestamp duration, the result is the hour part of the value, which is an integer between -99 and +99. A nonzero result has the same sign as the argument.

Example: 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 schema is SYSIBM.

#

#

#

The IDENTITY_VAL_LOCAL function is a nondeterministic function ²¹ that returns the most recently assigned value for an identity column. The function has no input parameters.

#

#

The result is a DECIMAL(31,0) regardless of the actual data type of the identity column that the result value corresponds to.

#

#

#

#

#

#

#

The value returned is the value that was assigned to the identity column of the table identified in the most recent single row INSERT statement with a VALUES clause for a table with an identity clause. The INSERT statement 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 could 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.

#

#

#

#

#

#

The function returns the null value in the following situations:

- When a single row INSERT statement with a VALUES clause 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 INSERT statement that assigned a value

#

#

#

#

#

#

The result of the function is not affected by the following statements:

- An INSERT statement with a VALUES clause for a table that does not contain an identity column ²²
- An INSERT statement with a subselect
- A ROLLBACK TO SAVEPOINT statement

#

#

#

#

#

#

#

²¹ Being nondeterministic affects what optimization (such as view processing and parallel processing) can be done when this function is used and in what contexts that the function can be invoked. For example, the RAND function is another built-in scalar that is not deterministic. Using nondeterministic functions within a predicate can cause unpredictable results.

²² On DB2 platforms that support multiple row INSERT statements with a VALUES clause, the previous bullet would be replaced with the following bullets:

- A single row INSERT statement with a VALUES clause for a table that does not contain an identity column
- A multiple row INSERT statement with a VALUES clause

Notes
The following notes explain the behavior of the function when it is invoked in
various situations:

Invoking the function within the VALUES clause of an INSERT statement
Expressions in the VALUES clause of an INSERT statement are evaluated
before values are assigned to the target columns of the INSERT statement.
Thus, when you invoke IDENTITY_VAL_LOCAL in a VALUES clause of an
INSERT statement, the value that is used is the most recently assigned value
for an identity column from a previous INSERT statement. The function
returns the null value if no such INSERT statement had been executed within
the same level as the invocation of the IDENTITY_VAL_LOCAL function.

Invoking the function following a failed INSERT statement
The function returns an unpredictable result when it is invoked after the
unsuccessful execution of a single row INSERT with a VALUES clause 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 INSERT
or the value that would have been assigned had the INSERT 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 assigned by one
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 single row INSERT statement invoked in
the same triggered action that had a VALUES clause for a table containing an
identity column. If a single row INSERT statement with a VALUES clause 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 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 VALUES clause of the INSERT
#   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.
```

Examples

```
# 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
```

IDENTITY_VAL_LOCAL

```
#          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 a identity  
#          column in a previous INSERT statement.
```

IFNULL

IFNULL



```
IFNULL(expression, expression)
```

The schema is SYSIBM.

The IFNULL function returns the first argument that is not null.

IFNULL is identical to the COALESCE and VALUE scalar functions except that IFNULL is limited to two arguments instead of multiple arguments. For a description, see “COALESCE” on page 203.

Example: For all the rows in sample table DSN8610.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 DSN8610.EMP;
```

INSERT

▶—INSERT(*expression1*,*expression2*,*expression3*,*expression4*)—▶

The schema is SYSIBM.

The INSERT function returns a string where, beginning at *expression2* in *expression1*, *expression3* bytes have been deleted and *expression4* has been inserted.

expression1

An expression that specifies the source string. The source string can be any type of character string except a CLOB or any type of graphic string except a DBCLOB. The actual length of the string must be greater than zero.

expression2

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 *expression1* plus one.

expression3

An expression that returns an integer. The integer specifies the number of bytes that are to be deleted from the source string, starting at the position identified by *expression2*. The value of the integer must be in the range of 0 to the length of *expression1*.

expression4

An expression that specifies the string to be inserted into the source string, starting at the position identified by *expression2*. The string to be inserted can be any type of character string except a CLOB or any type of graphic string except a DBCLOB.

expression1 and *expression4* must have the same string type. Both expressions must be character strings, or both expressions must be graphic strings. If the expressions are character strings, neither must be a CLOB. If the expressions are graphic strings, neither must be a DBCLOB.

The result of the function depends on the data type of the first and fourth arguments:

- VARCHAR if *expression1* and *expression4* are character strings
- VARGRAPHIC if *expression1* and *expression4* are graphic strings

The length attribute of the result depends on the arguments:

- If *expression2* and *expression3* are constants, the length attribute of the result is:

$$L1 - \text{MIN}((L1 - V2 + 1), V3) + L4$$

where:

L1 is the length attribute of *expression1*

V2 is the value of *expression2*

V3 is the value of *expression3*

INSERT

L4 is the length attribute of *expression4*

- Otherwise, the length attribute of the result is the length attribute of *expression1* plus the length attribute of *expression4*. In this case, the length attribute of *expression1* plus the length attribute of *expression4* must not exceed 4000 for a VARCHAR result or 2000 for a VARGRAPHIC result.

The actual length of the result is:

$$A1 - \text{MIN}((A1 - V2 + 1), V3) + A4$$

where:

A1 is the actual length of *expression1*

V2 is the value of *expression2*

V3 is the value of *expression3*

A4 is the actual length of *expression4*

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.

The subtype and CCSID of the result are determined as follows:

- If either *expression1* and *expression4* is character bit data, the result is bit data and does not have an associated CCSID.
- If *expression1* and *expression4* are both character SBCS data, the result is SBCS data and the CCSID is the ASCII or EBCDIC CCSID for SBCS data, depending on the encoding scheme of other data in the SQL statement.
- If *expression1* and *expression4* are both graphic data, the result is graphic data and the CCSID is the ASCII or EBCDIC CCSID for graphic data, depending on the encoding scheme of other data in the SQL statement.
- Otherwise, the result is mixed data, which is not necessarily well-formed. The CCSID is the EBCDIC or ASCII CCSID for mixed data, depending on the encoding scheme of other data in the SQL statement.

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 (*expression2*).

```
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;
```

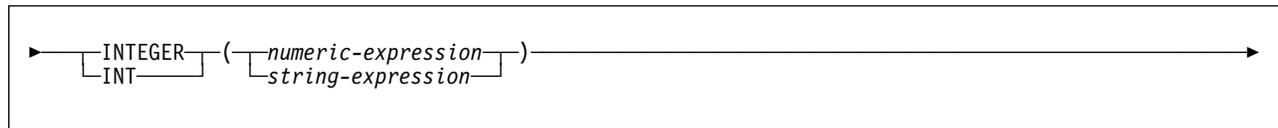
This example returns 'XXINSERTIN', 'XXNSERTING', 'XXSERTING ', and 'XXERTING '.

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 '.

INTEGER or INT



The schema is SYSIBM.

The INTEGER or INT function returns an integer representation of a number or character string in the form of a numeric constant.

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result of the function is a large integer. The result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. If present, the decimal part of the argument is truncated.

string-expression

An expression that returns any type of character string, except a CLOB, with an actual length that is not greater than 255 bytes. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming an SQL integer constant.

The result of the function is a large integer. The result is the same number that would occur if the corresponding numeric constant were assigned to a large integer column or variable.

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 DSN8610.EMP, find the average salary of the employees in department A00, rounding the result to the nearest dollar.

```
SELECT INTEGER(AVG(SALARY)+.5)
FROM DSN8610.EMP
WHERE WORKDEPT = 'A00';
```

Example 2: Using sample table DSN8610.EMP, select the EMPNO column, which is defined as CHAR(6), in integer form.

```
SELECT INTEGER(EMPNO)
FROM DSN8610.EMP;
```

JULIAN_DAY

►—JULIAN_DAY(*expression*)—►

The schema is SYSIBM.

The JULIAN_DAY function returns an integer value representing a number of days from January 1, 4712 B.C. (the start of the Julian date calendar) to the date specified in the argument.

The argument must be a date, a timestamp, or a valid character string representation of a date or timestamp. A character string representation must not be a CLOB 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 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.

Example 1: Using sample table DSN8610.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 DSN8610.EMP
      WHERE EMPNO = '000010';
```

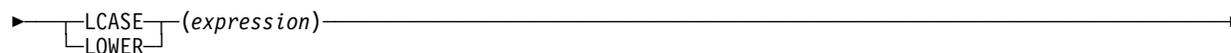
The result is that JDAY is set to 2438762.

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.

| LCASE or LOWER



The schema is SYSIBM.

The LCASE or LOWER function returns a string in which all the characters have been converted to lowercase characters.

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 must have an actual length that is not greater than 255. A graphic string argument must not be a DBCLOB and must have an actual length that is not greater than 127.

The alphabetic characters of the argument are translated to lowercase characters based on the value of the LC_CTYPE locale in effect for the statement. For example, characters A-Z are translated to a-z, and characters with diacritical marks are translated to their lowercase equivalent, if any. The locale is determined by special register CURRENT LOCALE LC_CTYPE. For information about the special register, see "CURRENT LOCALE LC_CTYPE" on page 107.

If the LC_CTYPE locale is blank when the function is executed, the result of the function depends on the data type of *expression*. For a character string expression, characters A-Z are translated to a-z and characters with diacritical marks are not translated. For a graphic string expression, an error occurs.

The string can contain mixed data. However, because the function operates on a strict byte-count basis, the result is not necessarily a properly formed mixed data character string.

The length attribute, data type, subtype, and CCSID of the result are the same as the argument. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example 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'.

LEFT



The schema is SYSIBM.

The LEFT function returns a string consisting of the specified number of leftmost *integer* characters of *string*. If *string* is a character or binary string, a character is a byte. If *string* is a graphic string, a character is a DBCS character.

The CCSID of the result is the same as that of the *string*.

string

An expression that specifies the string from which the result is derived. The string must be a character, graphic, or binary string. A substring of *string* is zero or more contiguous bytes of *string*.

The string can contain mixed data. However, because the function operates on a strict byte-count basis, 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*.

The *string* is effectively padded on the right with the necessary number of characters so that the specified substring of *string* 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 Japan (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 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* and a data type that depends on the data type of *string*:

- VARCHAR if *string* is CHAR or VARCHAR
- CLOB if *string* is CLOB
- VARGRAPHIC if *string* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string* is DBCLOB
- BLOB if *string* is BLOB

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.

LEFT

Example 1: Assume that host variable ALPHA has a value of 'ABCDEF'. The following statement:

```
SELECT LEFT(:ALPHA,3)
FROM SYSIBM.SYSDUMMY1;
```

returns 'ABC', which are the three leftmost characters in ALPHA.

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:

```
SELECT LEFT(:NAME, :FIRSTNAME_LEN)
FROM SYSIBM.SYSDUMMY1;
```

returns the value 'KATIE'.

Example 3: The following statement returns a zero length string.

```
SELECT LEFT('ABCABC',0)
FROM SYSIBM.SYSDUMMY1;
```

Example 4: The FIRSTNME column in sample EMP table is defined as VARCHAR(12). Find the first name for an employee whose last name is 'BROWN' and return the first name in a 10-byte string.

```
SELECT LEFT(FIRSTNME,10)
FROM DSN8610.EMP
WHERE LASTNAME='BROWN';
```

This function returns a VARCHAR(10) string that has the value of 'DAVID' followed by 5 blank characters.

LENGTH

The schema is SYSIBM.

The LENGTH function returns the length of its argument.

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

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 but does not include the length control field of varying-length strings. 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. The length of all other values is the number of bytes used to represent the value:

- 2 for small integer
- 4 for large integer
- 4 for single precision floating-point
- 8 for double precision floating-point
- $\text{INTEGER}(p/2)+1$ for decimal numbers with precision p
- 4 for date
- 3 for time
- 10 for timestamp
- The length of the string for character strings
- 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:

```
SELECT LENGTH(FIRSTNME)
       FROM DSN8610.EMP
       WHERE EMPNO = '000280';
```

returns the value 5.

Example 2: Assume that HIREDATE is a column of data type DATE. Then, regardless of value:

```
LENGTH(HIREDATE)
```

returns the value 4, and

```
LENGTH(CHAR(HIREDATE, EUR))
```

returns the value 10.

LN

LN



LN(*expression*)

The schema is SYSIBM.

The LN function returns the natural logarithm of the argument. The LN and EXP functions are inverse operations. LOG is a synonym for LN.

The argument is an expression that returns the value of any built-in numeric data type. 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 NATLOG is DECIMAL(4,2) with a value of 31.62. The following statement:

```
SELECT LN(:NATLOG)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 3.45.

LOCATE

▶ LOCATE(*search-string*, *source-string* [, *start*]) ▶

The schema is SYSIBM.

The LOCATE function returns the starting position of the first occurrence of one string (the *search-string*) within another string (the *source-string*). Numbers for the starting position begin at 1 and not 0. If *search-string* is not found in *source-string*, the function returns 0.

search-string

An expression that specifies the string that is to be searched for. The search string can be a character, graphic, or binary string 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)
- A scalar function whose arguments are any of the above (including nested function invocations)
- A CAST function 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 that is to be searched. The source string can be a character, graphic, or binary string. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable (including a LOB locator variable)
- A scalar function whose arguments are any of the above (including nested function invocations)
- 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 whose value is a positive integer. The integer specifies the position in the source string at which the search begins. If *start* is specified, the LOCATE function is equivalent to:

$$\text{POSSTR}(\text{SUBSTR}(\textit{source-string}, \textit{integer}), \textit{search-string}) + \textit{integer} - 1$$

If *start* is not specified, the search begins at the first character of the source string and the LOCATE function is equivalent to:

$$\text{POSSTR}(\textit{source-string}, \textit{search-string})$$

LOCATE

The first and second arguments must have compatible string type. Both arguments must be character strings, graphic strings, or binary strings.

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 “POSSTR” on page 255.

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;
```

LOG10



▶—LOG10(*expression*)—▶

The schema is SYSIBM.

The LOG10 function returns the base 10 logarithm of the argument.

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

```
SELECT LOG10(:HLOG)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 2.

| LTRIM



▶—LTRIM(*expression*)—▶

The schema is SYSIBM.

The LTRIM function removes blanks from the beginning of a string expression. The LTRIM function returns the same results as the STRIP function with LEADING specified:

```
STRIP(expression, LEADING)
```

expression must be any character string expression other than a CLOB or any graphic string expression other than a DBCLOB. The characters that are interpreted as leading blanks depend on the encoding scheme of the data and the data type:

- If the argument is a DBCS graphic string, the leading DBCS blanks are removed. For data that is encoded in ASCII, the ASCII CCSID determines the hex value that represents a double-byte blank. For example, for Japan (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.
- Otherwise, leading 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.

The result of the function depends on the data type of its argument:

- 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 *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 *expression*.

Example: 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'.

MICROSECOND

► MICROSECOND(*expression*) ◄

The schema is SYSIBM.

The MICROSECOND function returns the microsecond part of its argument.

The argument must be a timestamp, a timestamp duration, or a valid character string representation of a timestamp. A character string representation must not be a CLOB and must have 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 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 other rules depend on the data type of the argument:

If the argument is a timestamp or character 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: 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



►—MIDNIGHT_SECONDS(*expression*)—◄

The schema is SYSIBM.

The MIDNIGHT_SECONDS function returns an integer value in the range of 0 to 86400 that represents the number of seconds between midnight and the time specified by the argument.

The argument must be a time, a timestamp, or a valid character string representation of a time or timestamp. A character string representation must not be a CLOB 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 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.

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.

MINUTE



►—MINUTE(*expression*)—►

The schema is SYSIBM.

The MINUTE function returns the minute part of its argument.

The argument must be a time, timestamp, time duration, timestamp duration, or valid character string representation of a time or timestamp. A character string representation must not be a CLOB and must have 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 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 other rules depend on the data type of the argument:

If the argument is a time, timestamp, or character 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: 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

► MOD(*expression, expression*) ◄

The schema is SYSIBM.

The MOD function divides the first argument by the second argument and returns the remainder.

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 arguments must each 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 integers, the data type of the result is a large 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 operand, and the symbols p' and s' denote the precision and scale of the second operand.
- If either 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.

Example: Assume that M1 and M2 are two host variables. Find the remainder of dividing M1 by M2.

```
SELECT MOD(:M1, :M2)
FROM SYSIBM.SYSDUMMY1;
```

The following table shows the result for this function for various values of M1 and M2.

M1 data type	M1 value	M2 data type	M2 value	Result of MOD(:M1,:M2)
INTEGER	5	INTEGER	2	1
INTEGER	5	DECIMAL(3,1)	2.2	0.6
INTEGER	5	DECIMAL(3,2)	2.20	0.60
DECIMAL(4,2)	5.50	DECIMAL(4,1)	2.0	1.50

MONTH

MONTH

▶—MONTH(*expression*)—▶

The schema is SYSIBM.

The MONTH function returns the month part of its argument.

The argument must be a date, timestamp, date duration, timestamp duration, or valid character string representation of a date or timestamp. A character string representation must not be a CLOB and must have an actual length that is no greater than 255 bytes. (For the valid formats of string representations of dates and timestamps, see “String representations of datetime values” 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 other rules depend on the data type of the argument:

If the argument is a date, timestamp, or character 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: Select all rows in the sample table DSN8610.EMP for employees who were born in May:

```
SELECT * FROM DSN8610.EMP
WHERE MONTH(BIRTHDATE) = 5;
```


NULLIF

NULLIF

►—NULLIF(*expression*,*expression*)—►

The schema is SYSIBM.

The NULLIF function returns null if the two arguments are equal; otherwise, it returns the value of the first argument.

The two arguments must be compatible. (See the compatibility matrix in Table 9 on page 85.) Neither argument can be a CLOB, DBCLOB, or BLOB. The attributes of the result are the attributes of the first argument. Any numbers specified must be of a built-in numeric data type.

For example, if the result of the first argument is a character string, the result of the other must also be a character string; if the result of the first argument is number, the result of the other must also be a number.

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)
```

POSSTR

►—POSSTR(*source-string*,*search-string*)—►

The schema is SYSIBM.

The POSSTR function returns the starting position of the first occurrence of one string (the *search-string*) within another string (the *source-string*). Numbers for the starting position begin at 1 and not 0.

source-string

An expression that specifies the source string that is to be searched. The source string can be a character, graphic, or binary string. The expression can be specified by any of the following:

- # • A constant
- # • A special register
- # • A host variable (including a LOB locator variable)
- # • A scalar function whose arguments are any of the above (including nested function invocations)
- # • A column name
- # • A CAST function whos 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 that is to be searched for. The search string can be a character, graphic, or binary string 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)
- # • A scalar function whose arguments are any of the above (including nested function invocations)
- # • 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 arguments must have the same string type. Both arguments must be character strings, graphic strings, or binary strings.

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.

The strings can contain mixed data.

- # • 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.

POSSTR

- 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.

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 the search string is zero, the result is 1.
- If the length of the source string is zero, the result is 0.
- If the value of the search string is equal to an identical length substring of contiguous positions from the value of the source string, the result is the starting position of the first such substring within the value of the 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;
```

POWER



▶—POWER(*expression1*,*expression2*)—▶

The schema is SYSIBM.

The POWER function returns the value of *expression1* to the power of *expression2*.

Each argument is an expression that returns the value of any built-in numeric data type. An argument with a DECIMAL or REAL data type is converted to a double precision floating-point number 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.
- Otherwise, the result is DOUBLE.

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

Example: Assume that host variable HPOWER is INTEGER with a value of 3. The following statement:

```
SELECT POWER(2, :HPOWER)
FROM SYSIBM.SYSDUMMY1;
```

returns the value 8.

QUARTER

►—QUARTER(*expression*)—►

The schema is SYSIBM.

The QUARTER function returns an integer in the range of 1 to 4 that represents the quarter of the year in which the date occurs. For example, the function returns a 1 for any dates in January, February, or March.

The argument must be a date, a timestamp, or a valid character string representation of a date or timestamp. A character string representation must not be a CLOB 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 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.

Example 1: The following function returns 3 because August is in the third quarter of the year.

```
SELECT QUARTER('1996-08-25')
FROM SYSIBM.SYSDUMMY1
```

Example 2: Using sample table DSN8610.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 DSN8610.PROJ
WHERE PROJNO = 'AD3111' AND ACTNO = 70;
```

QUART is set to 4.

RADIANS



► RADIANS(*expression*) ◄

The schema is SYSIBM.

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

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

```
SELECT RADIANS(:HDEG)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 3.1415926536.

RAISE_ERROR

► RAISE_ERROR(*sqlstate*,*diagnostic-string*) ◄

The schema is SYSIBM.

The RAISE_ERROR function causes the statement that includes the function to return an error with the specified SQLSTATE (along with SQLCODE -438) and error condition. The RAISE_ERROR function always returns NULL with an undefined data type.

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.

To use this function in a context where “Rules for result data types” on page 99 do not apply, 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 DSN8610.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 DSN8610.EMP;
```

RAND

The schema is SYSIBM.

The RAND function returns a random floating-point value between 0 and 1. An argument can be used as an optional seed value.

If an argument is specified, it must be an integer (SMALLINT or INTEGER) 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.

Example: Assume that host variable HRAND is an INTEGER with a value of 100. The following statement:

```
SELECT RAND(:HRAND)
FROM SYSIBM.SYSDUMMY1;
```

returns a random floating-point number between 0 and 1, such as the approximate value .0121398.

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

REAL(*numeric-expression* | *string-expression*)

The schema is SYSIBM.

The REAL function returns a single precision floating-point representation of a number or character string in the form of a numeric constant.

numeric-expression

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

The result of the function is a single precision floating-point number. 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-expression

An expression that returns any type of character string, except a CLOB, with an actual length that is not greater than 255 bytes. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming an SQL floating-point constant.

The result of the function is a single precision floating-point number. The result is the same number that would occur if the corresponding numeric constant were assigned to a single precision floating-point column or variable.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Using sample table DSN8610.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 DSN8610.EMP
       WHERE COMM > 0;
```

REPEAT

▶ REPEAT(*expression*, *integer*) ◀

The schema is SYSIBM.

The REPEAT function returns a character string composed of *expression* repeated *integer* times.

expression

An expression that specifies the string to be repeated. The string must be any type of character string except a CLOB, or any type of graphic string except a DBCLOB. The actual length of the string must be 32767 bytes or less.

integer

An expression whose value is a positive integer. The integer specifies the number of times to repeat the string.

The result of the function depends on the data type of the first argument:

- VARCHAR if *expression* is a character string
- VARGRAPHIC if *expression* is graphic string

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 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.

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

The subtype and CCSID of the result are determined as follows:

- If *expression* is character bit data, the result is bit data and does not have an associated CCSID.
- If *expression* is character SBCS data, the result is SBCS data and the CCSID is the ASCII or EBCDIC CCSID for SBCS data, depending on the encoding scheme of other data in the SQL statement.
- If *expression* is graphic data, the result is graphic data and the CCSID is the ASCII or EBCDIC CCSID for graphic data, depending on the encoding scheme of other data in the SQL statement.
- Otherwise, the result is mixed data, which is not necessarily well-formed. The CCSID is the EBCDIC or ASCII CCSID for mixed data, depending on the encoding scheme of other data in the SQL statement.

REPEAT

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 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

► REPLACE(*expression1*,*expression2*,*expression3*) ◄

The schema is SYSIBM.

The REPLACE function replaces all occurrences of *expression2* in *expression1* with *expression3*. If *expression2* is not found in *expression1*, *expression1* is returned unchanged.

expression1

An expression that specifies the source string. The expression cannot be an empty string.

expression2

An expression that specifies the string to be removed from the source string. The expression cannot be an empty string.

expression3

An expression that specifies the replacement string.

The arguments must have the same string type. All the arguments must be character strings, or all the arguments must be graphic strings. If the arguments are character strings, none must be a CLOB. If the arguments are graphic strings, none must be a DBCLOB. The actual length of each string must be 32767 bytes or less.

The result of the function depends on the data type of the arguments:

- VARCHAR if the arguments are character strings
- VARGRAPHIC if the arguments are graphic strings

The length attribute of the result depends on the arguments:

- If the length attribute of *expression3* is less than or equal to the length attribute of *expression2*, the length attribute of the result is the length attribute of *expression1*.
- Otherwise, the length attribute of the result is:

$$(L3 * (L1/L2)) + \text{MOD}(L1, L2)$$

where:

L1 is the length attribute of *expression1*

L2 is the length attribute of *expression2*

L3 is the length attribute of *expression3*

If the result is a character string, the length attribute of the result must not exceed 4000. If the result is a graphic string, the length attribute of the result must not exceed 2000.

The actual length of the result is the actual length of *expression1* plus the number of occurrences of *expression2* that exist in *expression1* multiplied by the actual length of *expression3* minus the actual length of *expression2*. If the actual length of the result string exceeds the maximum for the return data type, an error occurs.

REPLACE

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

The subtype and CCSID of the result are determined as follows:

- If *expression1*, *expression2*, or *expression3* is bit data, the result is bit data and does not have an associated CCSID.
- If all three expressions are character SBCS data, the result is SBCS data and the CCSID is the ASCII or EBCDIC CCSID for SBCS data, depending on the encoding scheme of other data in the SQL statement.
- If all three expressions are graphic data, the result is graphic data and the CCSID is the ASCII or EBCDIC CCSID for graphic data, depending on the encoding scheme of other data in the SQL statement.
- Otherwise, the result is mixed data, which is not necessarily well-formed. The CCSID is the EBCDIC or ASCII CCSID for mixed data, depending on the encoding scheme of other data in the SQL statement.

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 may 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'.

RIGHT

▶—RIGHT(*string*,*integer*)—▶

The schema is SYSIBM.

The RIGHT function returns a string consisting of the specified number of rightmost *integer* characters of *string*. If *string* is a character or binary string, a character is a byte. If *string* is a graphic string, a character is a DBCS character.

The CCSID of the result is the same as that of the *string*.

string

An expression that specifies the string from which the result is derived. The string must be a character, graphic, or binary string. A substring of *string* is zero or more contiguous bytes of *string*.

The string can contain mixed data. However, because the function operates on a strict byte-count basis, 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*.

The *string* is effectively padded on the right with the necessary number of blank characters so that the specified substring of *string* 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 Japan (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 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* and a data type that depends on the data type of *string*:

- VARCHAR if *string* is CHAR or VARCHAR
- CLOB if *string* is CLOB
- VARGRAPHIC if *string* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string* is DBCLOB
- BLOB if *string* is BLOB

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 *expression*.

RIGHT

| *Example 1:* Assume that host variable ALPHA has a value of 'ABCDEF'. The
| following statement:

```
|         SELECT RIGHT(ALPHA,3)  
|         FROM SYSIBM.SYSDUMMY1;
```

| returns the value 'DEF', which are the three rightmost characters in ALPHA.

| *Example 2:* The following statement returns a zero length string.

```
|         SELECT RIGHT('ABCABC',0)  
|         FROM SYSIBM.SYSDUMMY1;
```

ROUND

▶ ROUND(*expression1*,*expression2*)

The schema is SYSIBM.

The ROUND function returns *expression1* rounded to *expression2* places to the
right of the decimal point if *expression2* is positive or the left of the decimal point if
expression2 is zero or negative. For example, ROUND(748.58,-3) returns 700.

expression1

An expression that returns a value of any built-in numeric data type.

expression2

An expression that returns a small or large integer. The value of integer specifies the number of places to the right of the decimal point for the result if *expression2* is not negative. If *expression2* is negative, *expression1* is rounded to the sum of the absolute value of *expression2*+1 number of places to the left of the decimal point.

If the absolute value of *expression2* 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 *expression1* is positive, a value of 5 is rounded to the next higher positive number. If *expression1* is negative, a 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, -2, -3, and -4 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, and 0900.000.

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.

ROUND

```
|          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, 5 was rounded up to the next higher number while in the
| negative value -3.5, 5 was rounded down to the next lower negative number.)

ROWID



►—ROWID(*expression*)—►

The schema is SYSIBM.

The ROWID function casts the input argument to a row ID.

The argument can be any type of character string, except 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 row ID value that was previously generated by DB2 to ensure a valid row ID 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 row ID.

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 row ID value cannot be used as the value for a row ID column in the database.

Example: Assume that table EMPLOYEE contains a ROWID column EMP_ROWID. Also assume that the table contains a row that is identified by a row ID 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');
```

RTRIM



▶—RTRIM(*expression*)—▶

The schema is SYSIBM.

The RTRIM function removes blanks from the end of a string expression. The RTRIM function returns the same results as the STRIP function with TRAILING specified:

```
STRIP(expression, TRAILING)
```

expression must be any character string expression other than a CLOB or any graphic string expression other than a DBCLOB. The characters that are interpreted as trailing blanks depend on the encoding scheme of the data and the data type:

- If the argument is a DBCS graphic string, then the trailing DBCS blanks are removed. If the data is encoded in ASCII, the ASCII CCSID determines the hex value that represents a double-byte blank. For example, for Japan (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.
- If the argument is an SBCS graphic string, 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.

The result of the function depends on the data type of its argument:

- 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 *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 *expression*.

Example: 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'.

SECOND

▶—SECOND(*expression*)—▶

The schema is SYSIBM.

The SECOND function returns the seconds part of its argument.

The argument must be a time, timestamp, time duration, timestamp duration, or valid character string representation of a time or timestamp. A character string representation must not be a CLOB and must have 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 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 other rules depend on the data type of the argument:

If the argument is a time, timestamp, or character string representation of either, 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. Then, when TIME_DUR has the value 153045:

```
SECOND(:TIME_DUR)
```

returns the value 45.

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'. Then, for this value:

```
SECOND(RECEIVED)
```

returns the value 30.

SIGN

SIGN



►—SIGN(*expression*)—►

The schema is SYSIBM.

The SIGN function returns an indicator of the sign of the argument. The returned value is:

-1	if the argument is less than zero
0	if the argument is zero
1	if the argument is greater than zero

The argument is an expression that returns a value of any built-in numeric data
type, except DECIMAL(31,31).

The result of the function has the same data type and length attribute as the argument except that when the argument is DECIMAL, the precision is increased by one if the argument's precision and scale are equal. For example, an argument with a data type of DECIMAL(5,5) results 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

►—SIN(*expression*)—►

The schema is SYSIBM.

The SIN function returns the sine of the argument, where the argument is an angle expressed in radians. The SIN and ASIN functions are inverse operations.

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

```
SELECT SIN(:SINE)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 0.99.

SINH

SINH



►—SINH(*expression*)—►

The schema is SYSIBM.

The SINH function returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

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

```
SELECT SINH(:HSINE)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 2.12.

SMALLINT

SMALLINT(*numeric-expression* | *string-expression*)

The schema is SYSIBM.

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

numeric-expression

An expression that returns a value of any built-in numeric data type.

The result of the function is a small integer. 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-expression

An expression that returns any type of character string, except a CLOB, with an actual length that is not greater than 255 bytes. Leading and trailing blanks are removed from the string, and the resulting substring must conform to the rules for forming an SQL integer constant.

The result of the function is a small integer. The result is the same number that would occur if the corresponding numeric constant were assigned to a small integer column or variable.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Using sample table DSN8610.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 DSN8610.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.

SPACE

►—SPACE(*expression*)—►

The schema is SYSIBM.

The SPACE function returns a character string that consists of the number of SBCS blanks that the argument specifies.

The argument is an expression that results in an integer. The integer 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. The CCSID is the EBCDIC or ASCII CCSID for SBCS data, depending on the encoding scheme of other data in the SQL statement.

If *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 *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



► SQRT(*expression1*)

The schema is SYSIBM.

The SQRT function returns the square root of the argument.

The argument can be any built-in numeric data type. If the argument is not double precision floating point, it is converted to a double precision floating-point number 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.

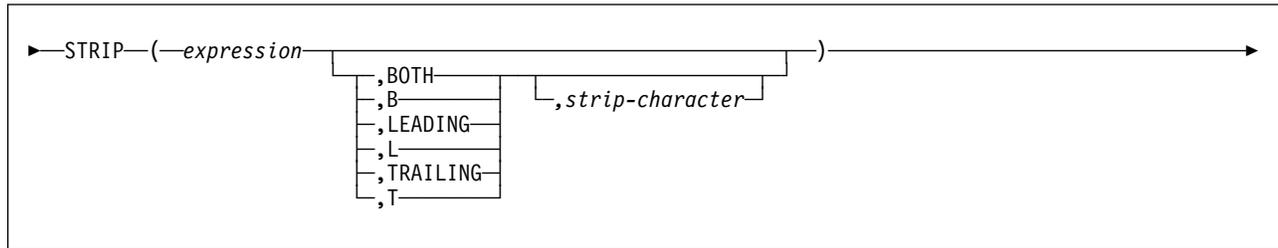
Example: Assume that host variable SQUARE is defined as DECIMAL(2,1) and has a value of 9.0. Find the square root of SQUARE.

```
SELECT SQRT(:SQUARE)
FROM SYSIBM.SYSDUMMY1;
```

This example returns a double precision floating-point number with an approximate value of 3.

STRIP

STRIP



The schema is SYSIBM.

The STRIP function removes blanks or another specified character from the end, the beginning, or both ends of a string expression.

The first argument is an expression that returns any type of string except a CLOB, DBCLOB, or BLOB.

The second argument indicates whether characters are removed from the beginning, the end, or both ends of the string. If you do not specify a second argument, blanks are removed from both the beginning and end of the string.

The third argument is a single-character constant that indicates the SBCS or DBCS character that is to be removed. If the first argument is a DBCS graphic or DBCS-only string, the third argument must be a graphic constant consisting of a single DBCS character. If the data type is not appropriate or the value contains more than one character, an error is returned.

If you do not specify the third argument, the following occurs:

- If the first argument is a DBCS graphic string, then the default strip character is a DBCS blank. The hex representation of a DBCS blank depends on the encoding scheme and CCSID of the data. For example, for data encoded in ASCII, a DBCS blank for Japan (CCSID 301) is X'8140', while for simplified Chinese it is X'A1A1'. For EBCDIC DBCS, X'4040' is interpreted as a DBCS blank.
- The default strip character is an SBCS blank. If the data is encoded in ASCII, then X'20' represents a blank. Otherwise, X'40' represents an EBCDIC blank.

The result of the function is a varying-length string with the same maximum length as the length attribute of the string. 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.

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, too.

Rewrite the example so that no beginning blanks are removed.

```
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'.

SUBSTR

▶ SUBSTR(*string*, *start*, *length*) ◀

The schema is SYSIBM.

The SUBSTR function returns a substring of a string.

string

An expression that specifies the string from which the result is derived. The string must be a character, graphic, or binary string. If *string* 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* is zero or more contiguous characters of *string*. If *string* is a graphic string, a character is a DBCS character. If *string* 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* to be the first character of the result. The value of integer must be between 1 and the length attribute of *string*. (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*.

length

An expression that specifies the length of the resulting substring. The length must be an integer in the range 0 to n , where n is equal to $L-S+1$ (L is the length attribute of *string* and S is the value of *start*). The specified length must not be the integer constant 0.

If *string* is a varying-length string and if *length* is explicitly specified, *string* is effectively padded on the right with the necessary number of characters so that the specified substring of *string* always exists. Hexadecimal zeroes are used as the padding character when *string* is BLOB data. Otherwise, a blank is used as the padding character. If *string* is a fixed-length string, omission of *length* is an implicit specification of $\text{LENGTH}(\textit{string}) - \textit{start} + 1$, which is the number of characters from the character specified by *start* to the last character of *string*. If *string* is a varying-length string, omission of *length* is an implicit specification of zero or $\text{LENGTH}(\textit{string}) - \textit{start} + 1$, whichever is greater.

If *length* is explicitly specified by an integer constant that is 255 or less, the result is a fixed-length string. If *length* is not explicitly specified, but *string* is a fixed-length string and *start* is an integer constant, the result is a fixed-length string. In all other cases, the result is a varying-length string with a maximum length that is the same as the length attribute of *string*. The result is subject to the restrictions that apply to long strings if its maximum length is greater than 255. These restrictions also apply if it is a graphic string whose maximum length is greater than 127.

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*.

Example 1: FIRSTNME is a VARCHAR(12) column in sample table DSN8610.EMP. One of its values is the 5-character string 'MAUDE'. When FIRSTNME has this value:

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 DSN8610.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 DSN8610.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);
```

TAN

TAN



►—TAN(*expression*)—►

The schema is SYSIBM.

The TAN function returns the tangent of the argument, where the argument is an angle expressed in radians. The TAN and ATAN functions are inverse operations.

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

```
SELECT TAN(:TANGENT)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 14.10.

TANH



►TANH(*expression*)◄

The schema is SYSIBM.

The TANH function returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians. The TANH and ATANH functions are inverse operations.

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

```
SELECT TANH(:HTANGENT)
FROM SYSIBM.SYSDUMMY1;
```

returns a double precision floating-point number with an approximate value of 0.90.

TIME

TIME



►—TIME(*expression*)—►

The schema is SYSIBM.

The TIME function returns a time derived from its argument.

The argument must be a time, a timestamp, or a valid character string representation of a time or timestamp. A character string representation must not be a CLOB and must have 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 76.)

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.

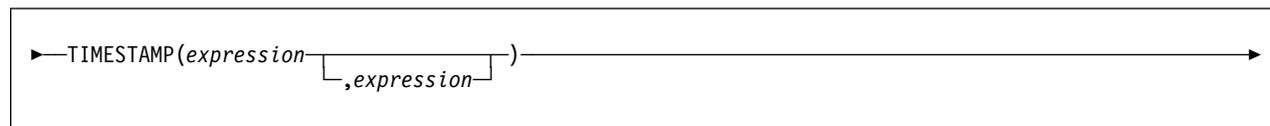
If the argument is a timestamp, the result is the time part of the timestamp.

If the argument is a character string, the result is the time or time part of the timestamp represented by the character string. If the CCSID of the string is not the same as the corresponding default CCSID at the application server, the string is first converted to that 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 schema is SYSIBM.

The TIMESTAMP function returns a timestamp derived from its argument or arguments.

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

If only one argument is specified, it must be a timestamp, a valid string representation of a timestamp, a character string of length 8, or a character string of length 14. The argument cannot be a CLOB. (String representations for a timestamp are described in “String representations of datetime values” on page 76.)

A character string of length 8 is assumed to be a Store Clock value.

A character string of length 14 must be a string of digits that represents a valid date and time in the form *yyyymmddhhmmss*, where *yyyy* is the year, *mm* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.

If both arguments are specified, the first argument must be a date or a valid string representation of a date and the second argument must be a time or a valid string representation of a time. Neither argument can be a CLOB. (Table 4 on page 77 and Table 5 on page 77 lists the valid formats for string representations for dates and times.)

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 character string, the result is the timestamp represented by that character string. The timestamp represented by a string of length 14 has a microsecond part of zero. The interpretation of a character string as a Store Clock value will yield a timestamp with a year between 1900 to 2042 as described in *ESA/390 Principles of Operation*.

If an argument is a character string with a CCSID that is not the same as the corresponding default CCSID at the application server, the string is first converted to that CCSID.

TIMESTAMP

Example: 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 1988 and TIMECOL represents 17 hours, 12 minutes, and 30 seconds after midnight. Then, for this row:

```
TIMESTAMP (DATECOL, TIMECOL)
```

returns the value '1988-12-25-17.12.30.000000'.

TIMESTAMP_FORMAT

```
#
# ──TIMESTAMP_FORMAT(string-expression,format-string)(1)───▶
#
# Note:
# 1 TO_DATE can be specified as synonym for TIMESTAMP_FORMAT.
```

```
# The schema is SYSIBM.
```

```
# The TIMESTAMP_FORMAT function returns a timestamp.
```

```
# string-expression
```

```
# An expression that returns any type of character string, except a CLOB, with a
# maximum length that is not greater than 255 bytes. Leading and trailing blanks
# are removed from the string, and the resulting substring is interpreted as a
# timestamp using the format specified by format-string.
```

```
# format-string
```

```
# An expression that returns a character string constant with a maximum length
# that is not greater than 255 bytes. format-string contains a template of how
# string-expression is to be interpreted as a timestamp value. Leading and trailing
# blanks are removed from the string, and the resulting substring must be a valid
# template for a timestamp. The only valid format for the function is:
```

```
# 'YYYY-MM-DD HH24:MI:SS'
```

```
# where:
```

```
# YYYY 4-digit year
```

```
# MM Month (01-12, January = 01)
```

```
# DD Day of month (01-31)
```

```
# HH24 Hour of day (00–24, when the value is 24, the minutes and seconds
# must be 0).
```

```
# MI Minutes (00–59)
```

```
# SS Seconds (00–59)
```

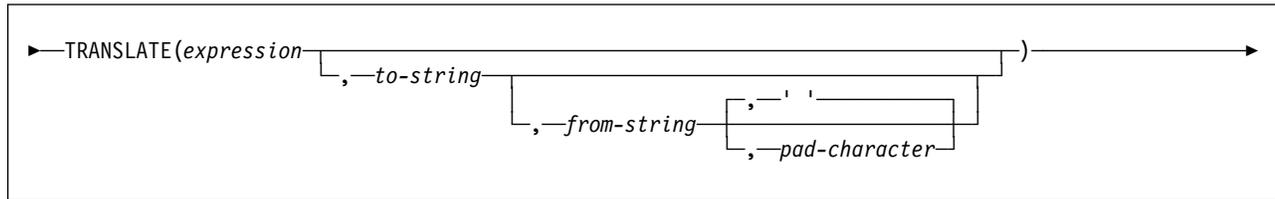
```
# The result of the function is a timestamp.
```

```
# If the argument can be null, the result can be null; if the argument is null, the result
# is the null value.
```

```
# Example: Set the character variable TVAR to the value of CREATEDTS from
# SYSIBM.SYSDATABASE if it is equal to one second before the beginning of the
# year 2000 ('1999-12-31 23:59:59'). The character string should be interpreted in the
# only format that can be specified for the function.
```

```
# SELECT VARCHAR_FORMAT(CREATEDTS, 'YYYY-MM-DD HH24:MI:SS')
# INTO :TVAR
# FROM SYSIBM.SYSDATABASE;
# WHERE CREATEDTS =
# TIMESTAMP_FORMAT('1999-12-31 23:59:59', 'YYYY-MM-DD HH24:MI:SS');
```

TRANSLATE



The schema is SYSIBM.

The TRANSLATE function translates one or more characters of *expression*.

expression

An expression that specifies the string to be translated. The string must be a character or graphic string. A character string argument must not be a CLOB and must have an actual length that is not greater than 255. A graphic string argument must not be a DBCLOB and must have an actual length that is not greater than 127.

to-string

A string that specifies the characters to which certain characters in *expression* are to be translated. This string is sometimes called the *output translation table*. The string must be a character or graphic string. A character string argument must not be a CLOB and must have an actual length that is not greater than 255. A graphic string argument must not be a DBCLOB and must have an actual length that is not greater than 127.

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-character* 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

A string that specifies the characters that if found in *expression* are to be translated. This string is sometimes called the *input translation table*. When a character in *from-string* is found, the character in *expression* is translated to the character in *to-string* that is in the corresponding position of the character in *from-string*.

from-string must be a character or a graphic string. A character string argument must not be a CLOB and must have an actual length that is not greater than 255. A graphic string argument must not be a DBCLOB and must have an actual length that is not greater than 127.

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-character

A string that specifies the character with which to pad *to-string* if its length is less than *from-string*. The string must be a character string (except a CLOB) or a graphic string (except a DBCLOB) with 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.

All of the arguments must have the same string type. They must all be character strings or all be graphic strings.

If *expression* is the only argument that is specified, the characters of its value are translated 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 translated to A-Z, and characters with diacritical marks are translated to their uppercase equivalent, if any. (For a description of the uppercase tables that are used for this translation, see *IBM National Language Support Reference Volume 2*. Special register CURRENT LOCAL LC_CTYPE determines the locale. If the LC_CTYPE locale is blank when the function is executed, the result of the function depends on the string type of *expression*. For a character string expression, characters a-z are translated to A-Z and characters with diacritical marks are not translated. For a graphic string expression, an error occurs.

If more than one argument is specified, the result string is built character-by-character from *expression* with each character in *from-string* being translated to the corresponding character in *to-string*. For each character in *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-character*. If the character is not found in *from-string*, it is moved to the result string without being translated.

The string can contain mixed data. However, because the function operates on a strict byte-count basis, the result is not necessarily a properly formed mixed data character string.

The length attribute, data type, subtype, and CCSID of the result are the same as *expression*. 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 was specified.

```
SELECT TRANSLATE (:SITE, 'j', 'B')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Hanauma jay'.

```
SELECT TRANSLATE (:SITE, 'ei', 'aa')
FROM SYSIBM.SYSDUMMY1
```

TRANSLATE

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'.

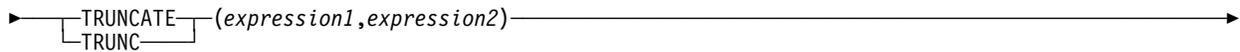
```
SELECT TRANSLATE (:SITE, '$$', 'L1')  
FROM SYSIBM.SYSDUMMY1
```

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 schema is SYSIBM.

The TRUNCATE or TRUNC function returns *expression1* truncated to *expression2* places to the right of the decimal point. If *expression2* is negative, *expression1* is truncated to the absolute value of *expression2* places to the left of the decimal point. If the absolute value of *expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example, TRUNCATE(748.58, -4) returns 0.

expression1

An expression that results in a value of any built-in numeric data type.

expression2

An expression that results in a small or large integer.

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 DSN8610.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 DSN8610.EMP;
```

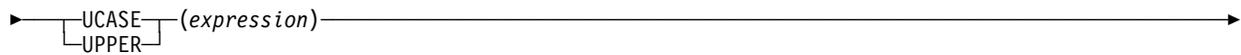
Because the highest paid employee in the sample employee table earns \$52750.00 per year, the example returns the value 4395.83.

Example 2: Return the number 873.726 truncated to 2, 1, 0, -1, and -2 decimal places respectively.

```
SELECT TRUNC(873.726,2),
       TRUNC(873.726,1),
       TRUNC(873.726,0),
       TRUNC(873.726,-1),
       TRUNC(873.726,-2)
FROM TABLEX
WHERE INTCOL = 1234;
```

This example returns the values 873.720, 873.700, 873.000, 870.000, and 800.000.

| UCASE or UPPER



The schema is SYSIBM.

The UCASE or UPPER function returns a string in which all the characters have been converted to uppercase characters.

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 must have an actual length that is not greater than 255. A graphic string argument must not be a DBCLOB and must have an actual length that is not greater than 127.

The alphabetic characters of the argument are translated to uppercase characters based on the value of the LC_CTYPE locale in effect for the statement. For example, characters a-z are translated to A-Z, and characters with diacritical marks are translated to their uppercase equivalent, if any. The locale is determined by special register CURRENT LOCALE LC_CTYPE. For information about the special register, see "CURRENT LOCALE LC_CTYPE" on page 107.

If the LC_CTYPE locale is blank when the function is executed, the result of the function depends on the data type of *expression*. For a character string expression, characters a-z are translated to A-Z and characters with diacritical marks are not translated. For a graphic string expression, an error occurs.

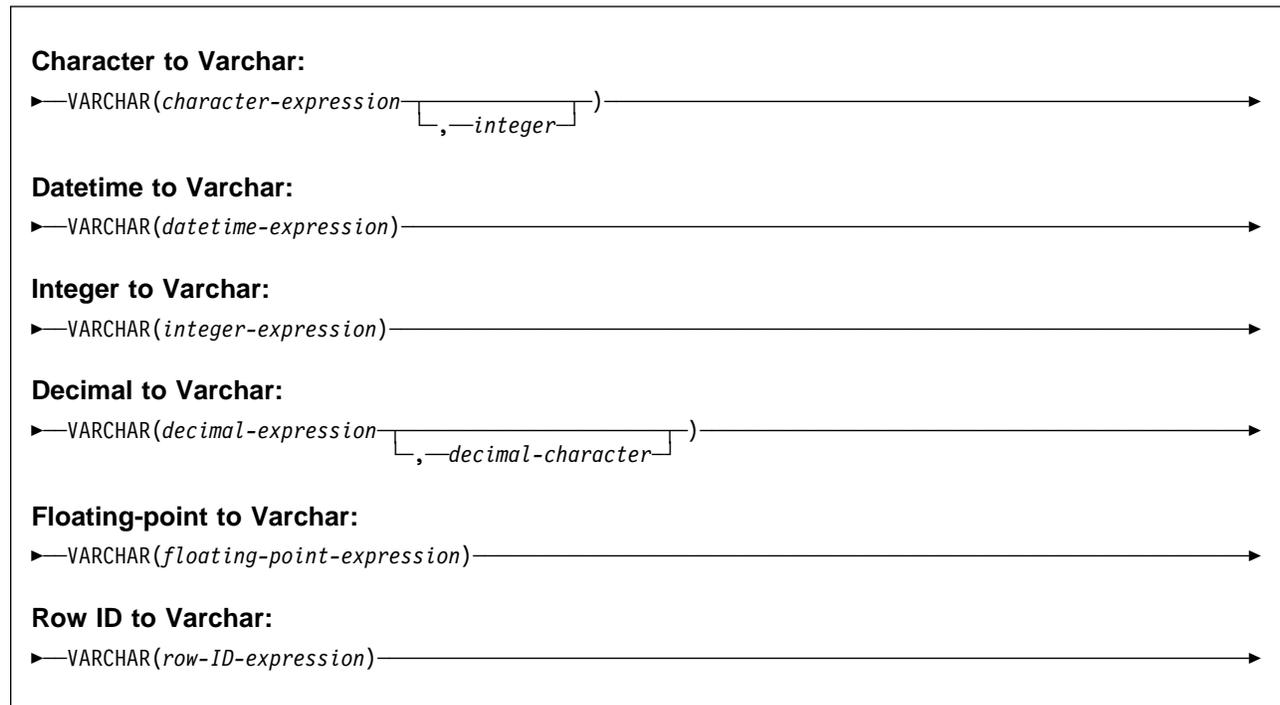
The length attribute, data type, subtype, and CCSID of the result are the same as the expression. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: 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'.

VARCHAR



The schema is SYSIBM.

The VARCHAR function returns a varying-length character string representation of a character string, datetime value, integer number, decimal number, floating-point number, or row ID value.

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.

The CCSID of the resulting string depends on the rest of the data referenced in the statement that contains the VARCHAR function. If the statement references ASCII data, the CCSID of the resulting string is the ASCII CCSID of the server. Otherwise, the CCSID is the EBCDIC CCSID of the server. Row ID values are not translated.

Character to Varchar

character-expression

An expression that returns a value with a character string data type.

integer

The length attribute for the resulting varying-length character string. The value must be between 1 and 32767. If the length is not specified, the length of the result is the same as the length of *character-expression*. If the first argument is mixed data, the second argument cannot be less than 4.

If the second argument 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

VARCHAR

empty string. Otherwise, the length attribute of the result is the same as the
length attribute of the first argument.

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*.

Datetime to Varchar

datetime-expression

An expression whose value has one of the following three 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.

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.

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 the SBCS CCSID of the server.

Integer to Varchar

integer-expression

An expression that returns a value with a small or large integer data type.

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.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. 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 server.

Decimal to Varchar

decimal-expression

An expression that returns a value that is a decimal data type. To change the precision and scale of the expression's value, apply the DECIMAL function to the expression before applying the VARCHAR function.

decimal-character

Specifies a single-byte character constant (CHAR or VARCHAR) that represents the decimal point in the resulting character string. The character cannot 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 "Options affecting SQL" on page 164.

The result is a varying-length character string representation (VARCHAR) of the argument in the form of an SQL decimal constant. The result includes a character that represents the decimal point and p digits where p is the precision of *decimal-expression*.

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. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit.

The CCSID of the result is the SBCS CCSID of the server.

Floating-Point to Varchar

floating-point-expression

An expression that returns a value that is a 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 the SBCS CCSID of the server.

Row ID to Varchar

VARCHAR

row-ID-expression

An expression whose value must be of a row ID data type.

The result is a varying-length character string representation (VARCHAR) of the argument. It is bit data and does not have an associated CCSID.

The length attribute of the result is 40. The actual length of the result is the length of *row-ID-expression*.

Example: Assume that host variable JOB_DESC is defined as VARCHAR(8). Using sample table DSN8610.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 DSN8610.EMP
      WHERE LASTNAME = 'QUINTANA';
```

VARCHAR_FORMAT

```
#
# ──VARCHAR_FORMAT(timestamp-expression,format-string)(1)───▶
# Note:
# 1 TO_CHAR can be specified as a synonym for VARCHAR_FORMAT.
```

```
#
# The schema is SYSIBM.
#
# The VARCHAR_FORMAT function returns a character representation of a
# timestamp in the format indicated by format-string.
#
# timestamp-expression
# An expression that returns a timestamp.
#
# format-string
# An expression that returns a character string constant with a maximum length
# that is not greater than 255 bytes. format-string contains a template of how
# timestamp-expression is to be formatted. Leading and trailing blanks are
# removed from the string, and the resulting substring must conform to the rules
# for formatting a timestamp. The only valid format that can be specified for the
# function is:
# 'YYYY-MM-DD HH24:MI:SS'
#
# where:
#
# YYYY 4-digit year
#
# MM Month (01-12, January = 01)
#
# DD Day of month (01-31)
#
# HH24 Hour of day (00–24, when the value is 24, the minutes and seconds
# must be 0).
#
# MI Minutes (00–59)
#
# SS Seconds (00–59)
#
# The result is the varying-length character string that contains the argument in the
# format specified by format-string. format-string also determines the length attribute
# and actual length 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 resulting string depends on the rest of the data referenced in the
# statement that contains the VARCHAR_FORMAT function. If the statement
# references ASCII data, the CCSID of the resulting string is ASCII CCSID of the
# server. Otherwise, the CCSID is the EBCDIC CCSID of server.
#
# Example: Set the character variable TVAR to the timestamp value of CREATEDTS
# from SYSIBM.SYSDATABASE, using the character string format supported by the
# function to specify the format of the value for TVAR.
```

VARCHAR_FORMAT

```
#          SELECT VARCHAR_FORMAT(CREATEDTS,'YYYY-MM-DD HH24:MI:SS')  
#          INTO :TVAR  
#          FROM SYSIBM.SYSDATABASE;
```

VARGRAPHIC

Character to Vargraphic:

► VARGRAPHIC(*character-expression* [*,integer*])

Graphic to Vargraphic:

► VARGRAPHIC(*graphic-expression* [*,integer*])

The schema is SYSIBM.

The VARGRAPHIC function returns a varying-length graphic string representation of a character string value, with the single-byte characters converted to double-byte characters, or a graphic string value.

The result of the function is a varying-length graphic string (VARGRAPHIC). If the length attribute of the result is greater than 127 double-byte characters, the result is a long string and subject to the restrictions that apply to long strings.

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 whose value must be an EBCDIC-encoded character string. 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 67 for these rules.)

integer

The length attribute of the resulting varying-length graphic string. The value of *integer* must be between 1 and 16352.

If the second argument is not specified and if *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.

The actual length of the result is the minimum of the length attribute of the result
and the actual 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 system EBCDIC CCSID for GRAPHIC data. If there is no system EBCDIC CCSID for GRAPHIC data, the CCSID of the result is X'FFFE'.

VARGRAPHIC

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 EBCDIC 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 EBCDIC CCSID for SBCS data. In this case, the operation proceeds as if the CCSID of the argument is M.
- The argument is BIT 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 EBCDIC CCSID for mixed data, conversion is to the coded character set that the system EBCDIC 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'.

Graphic to Vargraphic

graphic-expression

An expression that returns a value that is an EBCDIC-encoded graphic string.

integer

The length attribute for the resulting varying-length graphic string. The value
must be an integer between 1 and 16352.

If the second argument is not specified and if the *graphic-expression* is an
empty sting 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..

| If the first argument is an empty string, the result is an empty string.

| 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: 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;
```

WEEK

WEEK



► WEEK(*expression*) ◄

The schema is SYSIBM.

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.

The argument must be a date, a timestamp, or a valid character string representation of a date or timestamp. A character string representation must not be a CLOB and must have an actual length that is not greater than 255 bytes. (Table 4 on page 77 lists the valid formats for string representations of dates.)

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

Example: Using sample table DSN8610.PROJ, set the integer host variable WEEK to the week of the year that project 'AD2100' ended.

```
SELECT WEEK(PRENDATE)
       INTO :WEEK
       FROM DSN8610.PROJ
       WHERE PROJNO = 'AD2100';
```

The result is that WEEK is set 6.

YEAR



►—YEAR(*expression*)—◄

The schema is SYSIBM.

The YEAR function returns the year part of its argument.

The argument must be a date, timestamp, date duration, timestamp duration, or a valid character string representation of a date or timestamp. A character string representation must not be a CLOB 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 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 other rules depend on the data type of the argument specified:

If the argument is a date, a timestamp, or character 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 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: From the table DSN8610.EMP, select all rows for employees who were born in 1941.

```
SELECT *
  FROM DSN8610.EMP
 WHERE YEAR(BIRTHDATE) = 1941;
```

YEAR

Chapter 5. Queries

Authorization	309
subselect	311
select-clause	312
from-clause	315
where-clause	320
group-by-clause	321
having-clause	322
Examples of subselects	322
fullselect	327
Character conversion in unions and concatenations	328
Selecting the result CCSID	328
Examples of fullselects	330
select-statement	331
order-by-clause	331
update-clause	332
read-only-clause	333
optimize-for-clause	333
with-clause	334
queryno-clause	335
Examples of select statements	335

A *query* specifies a result table. A query is a component of certain SQL statements. There are three forms of a query:

- A *subselect*
- A *fullselect*
- A *select-statement*

A subselect is a subset of a fullselect, and a fullselect is a subset of a select-statement.

There is another SQL statement called SELECT INTO, which is described in “SELECT INTO” on page 806. SELECT INTO is not a subselect, fullselect, or a select-statement.

Authorization

The privilege set that is defined below 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)
- SYSADM authority
- SYSCTRL authority (catalog tables only)

For each user-defined function that is referenced in a query, the EXECUTE privilege on the user-defined function is also required.

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.

For dynamically prepared statements, the privilege set depends on the dynamic SQL statement behavior, which is specified by bind 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.

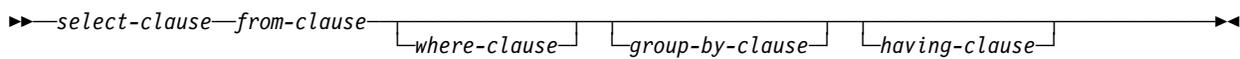
For a list of the DYNAMICRULES values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.

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 627 for the authorization rules that apply to the subselect component of CREATE VIEW.

Queries

If your installation uses the access control authorization exit (DSNX@XAC), that exit may be controlling the authorization rules instead of the rules that are listed here.

subselect



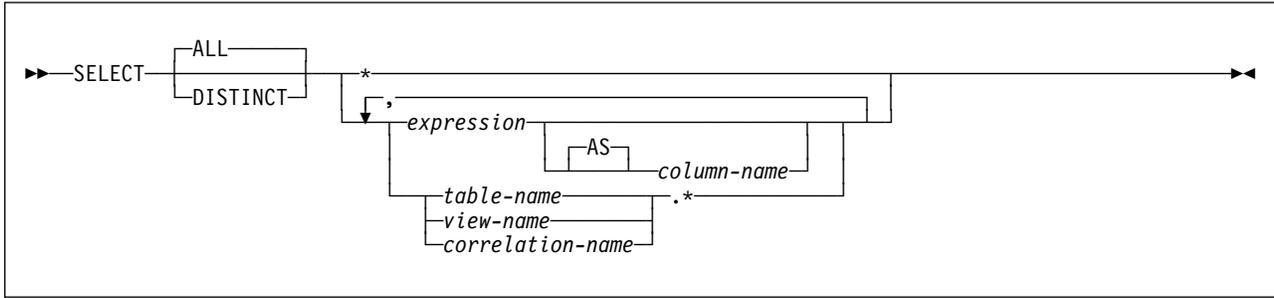
The *subselect* is a component of the fullselect, the CREATE VIEW statement, and the INSERT statement. It is also a component of certain predicates which, in turn, are components of a subselect. A subselect that is a component of a predicate is called a subquery.

A subselect specifies a result table derived from the result of its first FROM clause. The derivation 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 used to perform the derivation may be quite different from this description.)

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

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.

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. **DISTINCT** must not be used more than once in a subselect, with the exception of its use with a column function whose expression is a column. The same **DISTINCT** column function with the same column expression can be referred to more than once in a subselect. This restriction includes **SELECT DISTINCT** and the use of **DISTINCT** in a column function of the select list or **HAVING** clause. It does not include occurrences of **DISTINCT** in subqueries of the subselect.

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.

Select list notation:

- * Represents a list of names that identify the columns of table R. The first name in the list identifies the first column of R, the second name identifies the second column of R, and so on.

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.

expression

Can be any expression of the type that is described in “Expressions” on page 131. Each *column-name* in the expression 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 identified in an outer subselect.

AS column-name

Names or renames the result column. The name must not be qualified and does not have to be unique.

#

*name.**

Represents a list of names that identify the columns of *name*. *name* can be a table name, view name, or correlation name, and must designate a table or view named in the FROM clause. If a table is specified, it must not be an auxiliary table. 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 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.

SQL statements can be implicitly or explicitly rebound (prepared again). The effect of a rebound on statements that include * or *name.** is that the list of names is re-established. Therefore, the number of columns returned by the statement may 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.

Limitation on long string columns: The result of an expression must not be a character string with a maximum length greater than 255 or a graphic string with a maximum length greater than 127 if:

- SELECT DISTINCT is used.
- The subselect is a subquery.
- The subselect is an operand of UNION.

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 next two separate lists describe the results.

If neither GROUP BY nor HAVING is used:

- The select list must not include column functions, or it must be entirely a list of column functions.
- If the select list does not include column 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 is a list of column 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 GROUP BY or HAVING is used:

- Each *column-name* in the select list must either identify a grouping column or be specified within a column function.
- The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the column functions in the select list.

subselect

- 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 on page 323 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 column 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
- A result of a UNION if at least one of the corresponding items in the select list is nullable

Names of result columns: The name of a result column of a subselect is determined as follows:

- If the AS clause is specified, the name of the result column is the name specified on the AS clause. The name need not be unique.
- If the AS clause is not specified and the result column is derived from a column name, the result column name is the unqualified name of that column.
- All other result columns are unnamed.

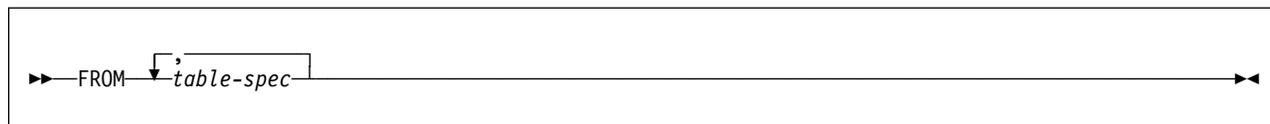
Names of result columns are placed into the SQL descriptor area (SQLDA) when the DESCRIBE statement is executed. This allows an interactive SQL processor such as SPUFI or QMF to use the column names when displaying the results. The names in the SQLDA include those specified by the AS clause.

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 25. 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.
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 131.
Any function	The data type of the result of the function. For a built-in function, see “Chapter 4. Built-in functions” on page 173 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 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.
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.

from-clause

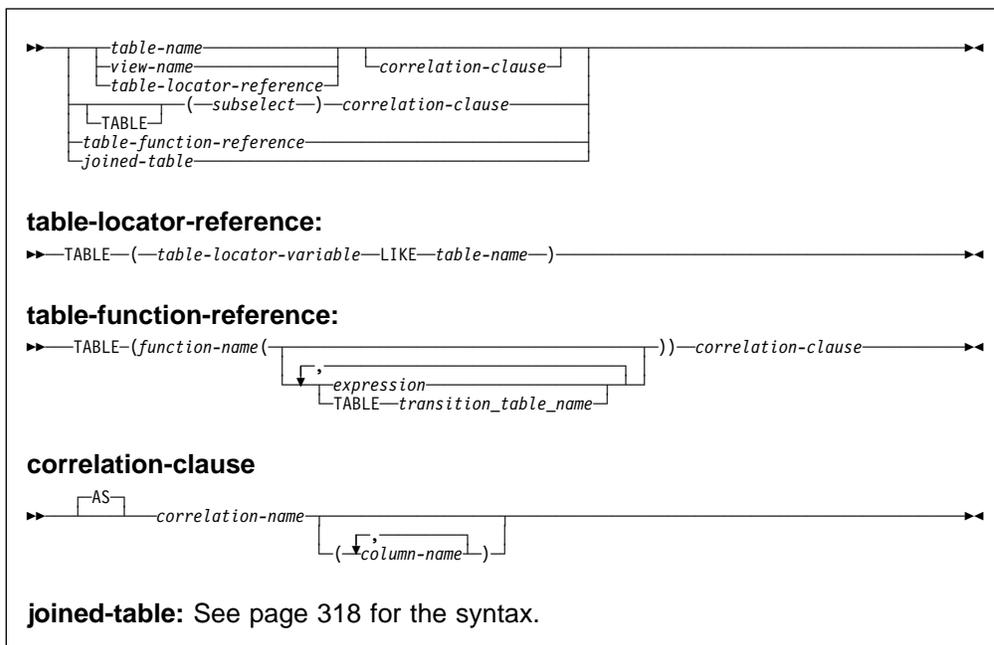


The FROM clause specifies an intermediate result table, R. If a single *table-spec* is specified, R is the result of that *table-spec*. If more than one *table-spec* is specified, R consists of all possible combinations of the rows of the result of each *table-spec*. Each row of R is a row from the result of the first *table-spec* concatenated with a row from the result of the second *table-spec*, concatenated

subselect

with a row from the result of the third *table-spec*, and so on. The number of rows in R is the product of the number of rows in the result of each *table-spec*. Thus, if the result of any *table-spec* is empty, R is empty.

table-spec



A *table-spec* specifies an intermediate result table:

- If a single table or view is identified, the intermediate 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*.
- A *subselect* 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.
- If a *function-name* is specified, the intermediate result table is the set of rows returned by the table function.
- If a *joined-table* is specified, the intermediate result table is the result of one or more join operations as explained below.

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. The tables that are identified must not be auxiliary tables. The tables, table functions, or underlying tables of the views that are identified must have the same encoding scheme—either all ASCII or all EBCDIC. 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.

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.

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 on page 127, 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.

Each *correlation-name* in a *correlation-clause* defines a designator for the immediately preceding intermediate 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. Using *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 a list of *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-spec*, the number of columns in the result of the *table-spec* 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, 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 columns names should be used to reference the columns. For more information, see "Column name qualifiers in correlated references" on page 116.

Correlated references in table-specs: 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 subselect 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-spec* 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.

subselect

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:

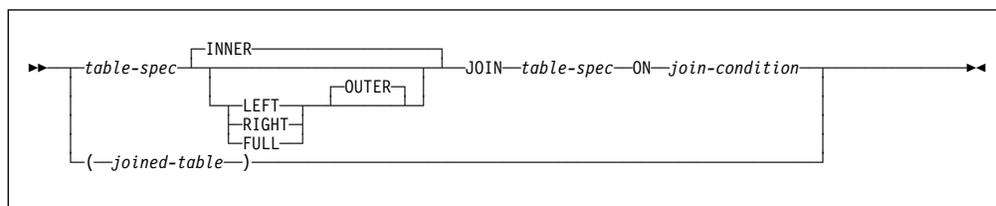
- Cannot participate in a FULL OUTER JOIN or a RIGHT OUTER JOIN
- Can participate in LEFT OUTER JOIN or an INNER JOIN if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause

Table 26 shows some examples of valid and invalid correlated references. TABF1 and TABF2 represent table functions.

Table 26. 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
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



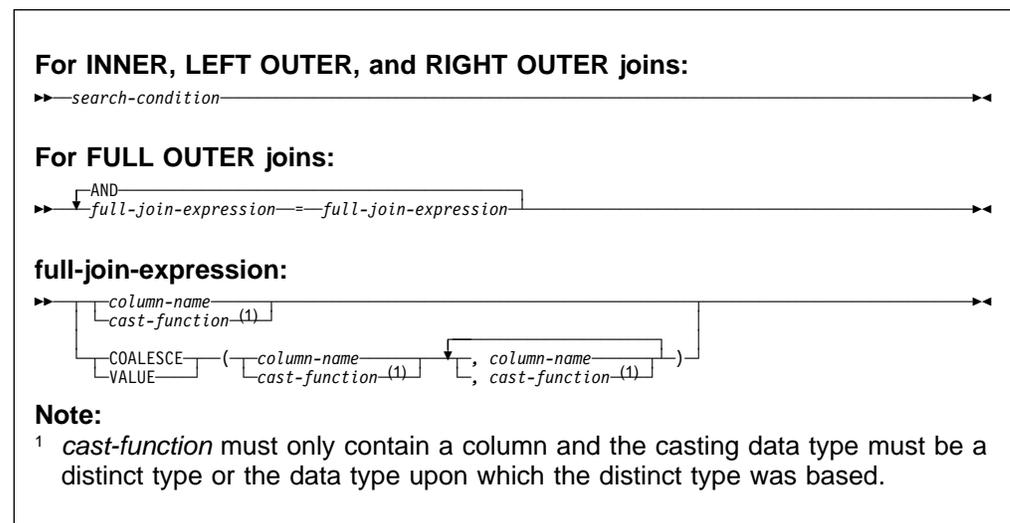
A *joined-table* specifies an intermediate result table that is the result of either an inner equi-join or an outer join. The table is derived by applying one of the join-operators: INNER, RIGHT OUTER, LEFT OUTER, or FULL OUTER to its operands. If a join-operator is not specified, INNER is implicit. The order in which a LEFT OUTER JOIN or RIGHT OUTER JOIN is performed can affect the result.

As described in more detail under “Join operations” on page 320 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 may be missing rows of 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.

join-condition



For INNER, LEFT OUTER, and RIGHT OUTER joins, the *join-condition* is a *search-condition* that must conform to these rules:

- It cannot contain any 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

subselect

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 and VALUE functions are 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 117 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

►—WHERE—*search-condition*—◄

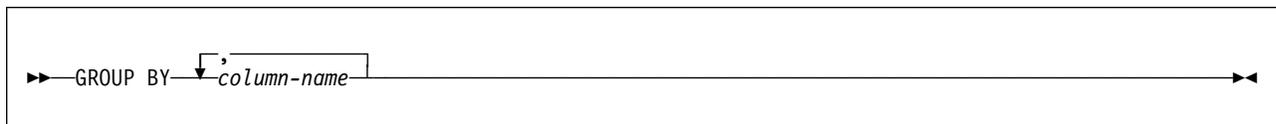
The WHERE clause specifies an intermediate result table that consists of those rows of R for which the search condition is true. R is the result of the FROM clause of the subselect.

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 identified in an outer subselect.
- A column 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 once, whereas a subquery with a correlated reference may have to be executed once for each row.

group-by-clause



The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause.

Each *column-name* must unambiguously identify a column of R other than a long string column. Each identified column is called a *grouping column*.

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 column are equal; and all rows with the same set of values of the grouping columns are in the same group. For grouping, all null values within a grouping column are considered equal.

Because every row of a group contains the same value of any grouping column, the name of a grouping column can be used in a search condition in a HAVING clause or an expression in a SELECT clause. In each case, the reference specifies only one value for each group. However, if the grouping column contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and may not all have the same length. In that case, a reference to the grouping column 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.

#

The Group By clause must not be used in a subquery of a basic predicate or must not be used if R is derived from a view whose outer subselect includes GROUP By or HAVING clauses.

GROUP BY must not be used in a subquery of a basic predicate.

having-clause

▶▶—HAVING—*search-condition*—◀◀

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the search-condition is true. R is the result of the previous clause. If this clause is not GROUP BY, R is considered a single group with no grouping columns.

Each *column-name* in *search-condition* must:

- Unambiguously identify a grouping column of R, or
- Be specified within a column function²³, or
- 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.

A group of R 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 R, 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” below.

A correlated reference to a group of R must either identify a grouping column or be contained within a column function.

The HAVING clause must not be used in a subquery of a basic predicate. When HAVING is used without GROUP BY, any column name in the select list must appear within a column function.

Examples of subselects

Example 1: Show all rows of the table DSN8610.EMP.

```
SELECT * FROM DSN8610.EMP;
```

Example 2: Show the job code, maximum salary, and minimum salary for each group of rows of DSN8610.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 DSN8610.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 DSN8610.EMPPROJACT: employee number, activity number, activity start date, and activity end date. Using the CHAR function, convert the start

²³ See “Chapter 4. Built-in functions” on page 173 for restrictions that apply to the use of column functions.

and end dates to their USA formats. Get the needed department information from the table DSN8610.EMP.

```
SELECT EMPNO, ACTNO, CHAR(EMSTDATE,USA), CHAR(EMENDATE,USA)
FROM DSN8610.EMPPROJACT
WHERE EMPNO IN (SELECT EMPNO FROM DSN8610.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 once.)

```
SELECT WORKDEPT, MAX(SALARY)
FROM DSN8610.EMP
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                      FROM DSN8610.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 DSN8610.EMP Q
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                      FROM DSN8610.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 DSN8610.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 3 on page 145 for CASE expressions.

Example 8: Get the employee number and employee name for all the employees in DSN8610.EMP. Order the results by the date of hire.

```
SELECT EMPNO, FIRSTNME, LASTNAME
FROM DSN8610.EMP
ORDER BY HIREDATE;
```

Example 9: 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.

subselect

```
SELECT EMPNO, FIRSTNAME, LASTNAME
FROM DSN8610.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 10: 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, to eliminate the suppliers that begin with the letter A from the table of parts, suppliers, product numbers and products, write a query like this:

```
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 11: 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 12: 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 13: 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:

subselect

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 14: The result of Example 11 (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, a synonym for the VALUE function, 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

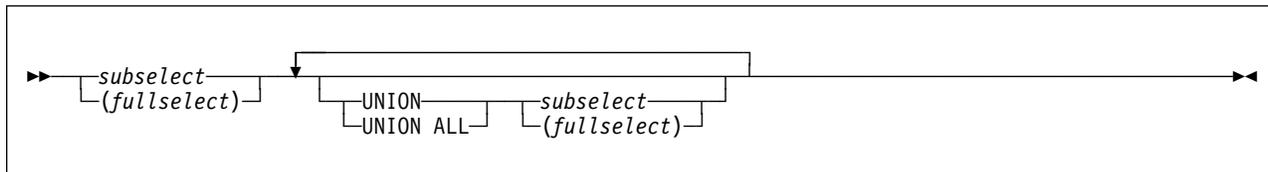
Example 15: For all parts that are used in product numbers less than 200, show the part, the part supplier, the product number, and the product name. Use a nested table expression.

```
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)

fullselect



fullselect specifies a result table. If UNION is not used, the result of the fullselect is the result of the specified subselect.²⁴

UNION or UNION ALL

Derives a result table by combining two other result tables, R1 and R2. If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with duplicate rows eliminated.

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 R has the same result column name. If the *n*th column of R1 and the *n*th column of R2 do not have the same name, the result column in R is unnamed.

Qualified column names cannot be used in the ORDER BY clause when UNION or UNION ALL is also specified.

Duplicate rows: Two rows are duplicates if each value in the first is equal to the corresponding value of the second. For determining duplicates, two null values are considered equal.

UNION and UNION ALL are associative operations. However, when UNION and UNION ALL are used in the same statement, the result depends on the order in which the operations are performed. Operations within parentheses are performed first. When the order is not specified by parentheses, operations are performed in order from left to right.

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. If UNION is specified without the ALL option, R1 and R2 must not include a long string column.

The *n*th column of the result of UNION and UNION ALL is derived from the *n*th columns of R1 and R2.

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 99.

²⁴ DB2 allows SELECT INTO as the operand of UNION. This is a deprecated feature with undefined results.

Character conversion in unions and concatenations

The SQL operations that combine strings are concatenation, UNION, and UNION ALL. Within an SQL statement, concatenation combines two or more strings into a new string. Within a fullselect, UNION and UNION ALL can combine two or more string columns resulting from the subselects into a 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.” These rules also apply to the COALESCE (or VALUE) scalar function.

Selecting the result CCSID

The result CCSID is selected at bind 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. The rules shown here apply when neither CCSID is X'FFFF'. When a CCSID is X'FFFF', the result CCSID is always X'FFFF', and no character conversions take place.

If one CCSID is for SBCS data and the other is for mixed data, the operand selected depends on the value of the MIXED DATA field on installation panel DSNTIPF at the DB2 where the operation takes place:

- If this value is YES, the operand MIXED furnishes the result CCSID.
- If this value is NO, the operand SBCS furnishes the result CCSID.

If both CCSIDs are the same type (both SBCS, both MIXED, or both GRAPHIC CCSIDs), then the operand that furnishes the result CCSID is as shown in Table 27.

For example, assume a concatenation of the form:

```
string-constant CONCAT derived-value
```

The value in the second row and fourth column shows that the first operand (*string-constant*) supplies the result CCSID.

Table 27. Operand that supplies the CCSID for character conversion

First operand	Second operand				
	Column value	String constant	Special register	Derived value	Host variable
Column value	first	first	first	first	first
String constant	second	first	first	first	first
Special register	second	first	first	first	first
Derived value	second	second	second	first	first
Host variable	second	second	second	second	neither ¹

Note: 1. Both operands are converted, if necessary, to the system CCSID of the server.

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 (or VALUE) scalar function is converted, if necessary, to the coded character set of the result string. Each string of an operand of UNION or UNION ALL 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 Language Environment. A warning occurs if a character of a string is converted to the substitution character.

#

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 DSN8610.EMP.

```
SELECT * FROM DSN8610.EMP;
```

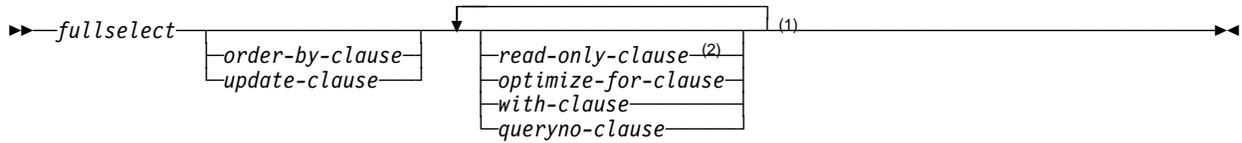
Example 3: Using sample tables DSN8610.EMP and DSN8610.EMPROJACT, list the employee numbers of all employees for which either of the following statements are true:

- Their department numbers begin with 'D'.
- They are assigned to projects whose project numbers begin with 'AD'.

```
SELECT EMPNO FROM DSN8610.EMP
WHERE WORKDEPT LIKE 'D%'
UNION
SELECT EMPNO FROM DSN8610.EMPPROJACT
WHERE PROJNO LIKE 'AD%';
```

The result is the union of two result tables, one formed from the sample table DSN8610.EMP, the other formed from the sample table DSN8610.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'.

select-statement



Notes:

- ¹ The same clause must not be specified more than once.
- ² Must not be specified if *update-clause* is specified.

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 causing a result table to be displayed at your terminal. In any case, the table specified by *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 OS/390 or remote queries in which the server and requester are DB2 for OS/390, if a table is encoded as ASCII, the retrieved data is encoded in EBCDIC. For information on retrieving such data encoded in ASCII, see Section 7 of *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 you execute a select statement on a table, no INSERT, UPDATE, or DELETE statement at a lower level of nesting must be executed on 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(...);
```

order-by-clause



The ORDER BY clause specifies an ordering of the rows of the result table. If a single column is identified, the rows are ordered by the values of that column. If

select-statement

more than one column is identified, the rows are ordered by the values of the first identified column, then by the values of the second identified column, and so on. A long string column must not be identified.

A named column can be identified by an integer or a column name. An unnamed column must be identified by an integer. A column is unnamed if the AS clause is not specified and it is derived from a constant, an expression with operators, or a function. If the fullselect includes a UNION operator, the fullselect rules on named columns apply.

column-name

Must unambiguously identify a column of the result table, with an exception that makes it possible to name a column that is not in the result table. If the query is a subselect, *column-name* can identify the column name of a table, view, or nested table expression identified in the FROM clause and not in the result table when the subselect does not use:

- DISTINCT in the select list
- Column functions in the select list
- GROUP BY

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.

ASC

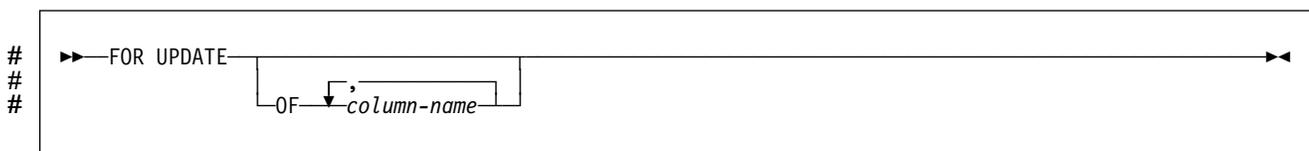
Uses the values of the column in ascending order. This is the default.

DESC

Uses the values of the column in descending order.

Ordering is performed in accordance with the comparison rules described in Chapter 3. Language elements, beginning on page 94. 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 column have an arbitrary order. If you do not specify ORDER BY, the rows of the result table have an arbitrary order.

update-clause



The optional UPDATE clause identifies the columns that can be updated in a later positioned UPDATE statement. 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 634. 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 UPDATE clause is specified without a list of columns, 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.

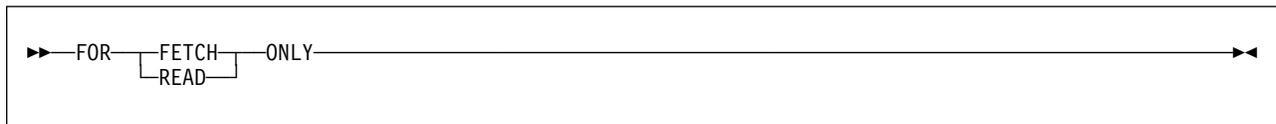
The declaration of a cursor referred to in a positioned UPDATE statement need not include an UPDATE clause if the STDSQL(YES) or NOFOR option is specified when the program is precompiled. For more on the subject, see “Positioned updates of columns” on page 172.

When FOR UPDATE OF 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 KEEP UPDATE LOCKS is specified in the SQL statement, an X lock, instead of a U lock, is acquired at FETCH time.

No locks are acquired on declared temporary tables. For a discussion of U locks and S locks, see Section 5 (Volume 2) of *DB2 Administration Guide*.

read-only-clause

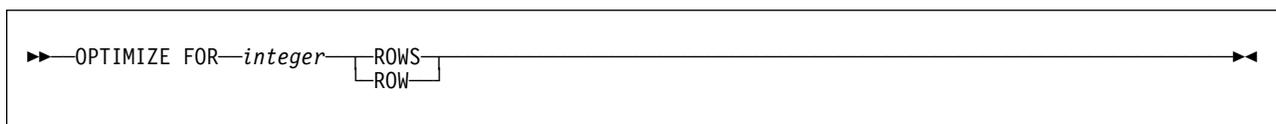


The clause FOR FETCH ONLY²⁵ declares that the result table is read-only and therefore the cursor cannot be referred to in positioned UPDATE and DELETE statements.

Some result tables are read-only by nature. (For example, a table based on a read-only view.) FOR FETCH ONLY can still be specified for such tables, but the specification has no effect. For result tables for which updates and deletes are possible, specifying FOR FETCH ONLY can possibly improve the performance of FETCH operations and distributed operations.

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 FETCH ONLY.

optimize-for-clause



²⁵ Or, FOR READ ONLY is equivalent.

select-statement

The OPTIMIZE FOR 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 OPTIMIZE FOR clause does not limit the number of rows that can be fetched or affect the result in any way other than performance. In general, if you are retrieving only a few rows, use 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*.

with-clause



#

The WITH 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.)

CS	Cursor stability
UR	Uncommitted read
RR	Repeatable read
RR KEEP UPDATE LOCKS	Repeatable read keep update locks
RS	Read stability
RS KEEP UPDATE LOCKS	Read stability keep update locks

WITH UR can be specified only if the result table is read-only.

WITH RR KEEP UPDATE LOCKS or WITH RS KEEP UPDATE LOCKS can be specified only if the FOR UPDATE OF clause is also specified. KEEP UPDATE LOCKS tells DB2 to acquire and hold an X lock instead of an U or S lock on all qualified pages and rows. Although this option 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

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

See “Notes” on page 636 for a list of the characteristics that make a result table read-only. A simple way to ensure that a result table is read-only is to specify FOR FETCH ONLY or FOR READ ONLY in the SQL statement.

queryno-clause

►—QUERYNO—*integer*—◄

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.

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, if hints are used. For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, see Section 5 (Volume 2) of *DB2 Administration Guide*.

Examples of select statements

Example 1: Select all the rows from DSN8610.EMP.

```
SELECT * FROM DSN8610.EMP;
```

Example 2: Select all the rows from DSN8610.EMP, arranging the result table in chronological order by date of hiring.

```
SELECT * FROM DSN8610.EMP ORDER BY HIREDATE;
```

select-statement

Example 3: Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the table DSN8610.EMP. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
  FROM DSN8610.EMP
  GROUP BY WORKDEPT
  ORDER BY 2;
```

#

Example 4: Change various salaries, bonuses, and commissions in the table DSN8610.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 DSN8610.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 DSN8610.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 DSN8610.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 FETCH ONLY;
```

Chapter 6. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements listed in the following table.

Table 28 (Page 1 of 4). SQL statements

SQL statement	Function	Page
ALLOCATE CURSOR	Defines and associates a cursor with a result set locator variable	346
ALTER DATABASE	Changes the description of a database	348
ALTER FUNCTION	Changes the description of an external user-defined function	351
ALTER INDEX	Changes the description of an index	364
ALTER PROCEDURE (external)	Changes the description of a stored procedure	376
ALTER PROCEDURE (SQL))	Changes the description of an SQL procedure	384
ALTER STOGROUP	Changes the description of a storage group	390
ALTER TABLE	Changes the description of a table	393
ALTER TABLESPACE	Changes the description of a table space	412
ASSOCIATE LOCATORS	Gets the result set locator value for each result set returned by a stored procedure	424
BEGIN DECLARE SECTION	Marks the beginning of a host variable declaration section	427
CALL	Calls a stored procedure	428
CLOSE	Closes a cursor	436
COMMENT ON	Replaces or adds a comment to the description of an object	438
COMMIT	Ends a unit of recovery and commits the database changes made by that unit of recovery	444
CONNECT (Type 1)	Connects the process to a server	449
CONNECT (Type 2)	Connects the process to a server	454
CREATE ALIAS	Defines an alias	457
CREATE AUXILIARY TABLE	Defines an auxiliary table for storing LOB data	457
CREATE DATABASE	Defines a database	462
CREATE DISTINCT TYPE	Defines a distinct type (user-defined data type)	465
CREATE FUNCTION (external scalar)	Defines a user-defined external scalar function	473
CREATE FUNCTION (external table)	Defines a user-defined external table function	492
CREATE FUNCTION (sourced)	Defines a user-defined function that is based on an existing scalar or column function	508
CREATE GLOBAL TEMPORARY TABLE	Defines a created temporary table at the current server	520
CREATE INDEX	Defines an index on a table	525
CREATE PROCEDURE (external)	Defines an external stored procedure	541
CREATE PROCEDURE (SQL)	Defines an SQL procedure	555

Statements

Table 28 (Page 2 of 4). SQL statements

SQL statement	Function	Page
CREATE STOGROUP	Defines a storage group	565
CREATE SYNONYM	Defines an alternate name for a table or view	568
CREATE TABLE	Defines a table	570
CREATE TABLESPACE	Defines a table space, which includes allocating and formatting the table space	597
CREATE TRIGGER	Defines a trigger	615
CREATE VIEW	Defines a view of one or more tables or views	627
DECLARE CURSOR	Defines an SQL cursor	634
# DECLARE GLOBAL # TEMPORARY TABLE	Defines a declared temporary table at the current server	639
DECLARE STATEMENT	Declares names used to identify prepared SQL statements	649
DECLARE TABLE	Provides the programmer and the precompiler with a description of a table or view	650
DELETE	Deletes one or more rows from a table	653
DESCRIBE (prepared statement or TABLE)	Describes the result columns of a prepared statement or the columns of a table or view	659
DESCRIBE CURSOR	Puts information about the result set associated with a cursor into a descriptor	666
DESCRIBE INPUT	Puts information about the input parameters (markers) of a prepared statement into a descriptor	668
DESCRIBE PROCEDURE	Puts information about the result sets returned by a stored procedure into a descriptor	671
DROP	Deletes objects	674
END DECLARE SECTION	Marks the end of a host variable declaration section	687
EXECUTE	Executes a prepared SQL statement	689
EXECUTE IMMEDIATE	Prepares and executes an SQL statement	692
EXPLAIN	Obtains information about how an SQL statement would be executed	694
FETCH	Assigns values of a row to host variables	707
FREE LOCATOR	Removes the association between a LOB locator variable and its value	710
GRANT (collection privileges)	Grants authority to create a package in a collection	714
GRANT (database privileges)	Grants privileges on a database	715
GRANT (distinct type privileges)	Grants the usage privilege on a distinct type (user-defined data type)	718
GRANT (function or procedure privileges)	Grants privileges on a user-defined function or a stored procedure	720
GRANT (package privileges)	Grants authority to bind, execute, or copy a package	725
GRANT (plan privileges)	Grants authority to bind or execute an application plan	727
GRANT (schema privileges)	Grants privileges on a schema	728
GRANT (system privileges)	Grants system privileges	728
GRANT (table or view privileges)	Grants privileges on a table or view	733

Table 28 (Page 3 of 4). SQL statements

SQL statement	Function	Page
GRANT (use privileges)	Grants authority to use specified buffer pools, storage groups, or table spaces	736
HOLD LOCATOR	Allows a LOB locator variable to retain its association with its value beyond a unit of work	738
INCLUDE	Inserts declarations into a source program	740
INSERT	Inserts one or more rows into a table	742
LABEL ON	Replaces or adds a label on the description of a table, view, alias, or column	749
LOCK TABLE	Locks a table or table space partition in shared or exclusive mode	751
OPEN	Opens a cursor	753
PREPARE	Prepares an SQL statement (with optional parameters) for execution	757
RELEASE (connection)	Places one or more connections in the release pending status	765
# RELEASE SAVEPOINT #	Releases a savepoint and any subsequently set savepoints within a unit of recovery	768
RENAME TABLE	Renames an existing table	769
REVOKE (collection privileges)	Revokes authority to create a package in a collection	777
REVOKE (database privileges)	Revokes privileges on a database	778
REVOKE (distinct type privileges)	Revokes the usage privilege on a distinct type (user-defined data type)	781
REVOKE (Function or Procedure privileges)	Revokes privileges on a user-defined function or a stored procedure	783
REVOKE (package privileges)	Revokes authority to bind, execute, or copy a package	788
REVOKE (plan privileges)	Revokes authority to bind or execute an application plan	790
REVOKE (schema privileges)	Revokes privileges on a schema	791
REVOKE (system privileges)	Revokes system privileges	793
REVOKE (table or view privileges)	Revokes privileges on a table or view	796
REVOKE (use privileges)	Revokes authority to use specified buffer pools, storage groups, or table spaces	799
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	801
# SAVEPOINT	Sets a savepoint within a unit of recovery	804
SELECT INTO	Specifies a result table of no more than one row and assigns the values to host variables	806
SET CONNECTION	Establishes the application server of the process by identifying one of its existing connections	809
SET CURRENT DEGREE	Assigns a value to the CURRENT DEGREE special register	812
SET CURRENT LOCALE LC_CTYPE	Assigns a value to the CURRENT LOCALE LC_CTYPE special register	814

Statements

Table 28 (Page 4 of 4). SQL statements

SQL statement	Function	Page
SET CURRENT OPTIMIZATION HINT	Assigns a value to the CURRENT OPTIMIZATION HINT special register	816
SET CURRENT PACKAGESET	Assigns a value to the CURRENT PACKAGESET special register	817
SET CURRENT PATH	Assigns a value to the CURRENT PATH special register	819
SET CURRENT PRECISION	Assigns a value to the CURRENT PRECISION special register	822
SET CURRENT RULES	Assigns a value to the CURRENT RULES special register	823
SET CURRENT SQLID	Assigns a value to the CURRENT SQLID special register	824
# SET host-variable # Assignment	Assigns values to host variables	826
# SET transition-variable # Assignment	Assigns values to transition variables	828
SIGNAL SQLSTATE	Signals an error with a user-specified SQLSTATE and description	
UPDATE	Updates the values of one or more columns in one or more rows of a table	833
VALUES	Provides a method to invoke a user-defined function from a trigger	842
VALUES INTO	Assigns values to host variables	843
WHENEVER	Defines actions to be taken on the basis of SQL return codes	845

How SQL statements are invoked

The SQL statements described in this chapter are classified as *executable* or *nonexecutable*. The section on invocation in the description of each statement 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. See Appendix B, “Characteristics of SQL statements in DB2 for OS/390” on page 873 for a list of executable statements.

A *nonexecutable statement* can only be embedded in an application program.

In addition to the statements described in this chapter, there is one more SQL statement construct: the *select-statement*. (See “select-statement” on page 331.) It is not included in this chapter because 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. You must precede each embedded statement with EXEC SQL.

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 <i>input</i> | The current value of the host variable is used in the execution of the statement. |
| As <i>output</i> | 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.²⁶ 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 statement is *never* executed, and acts as a “no-operation” if placed among executable statements of the application program. Therefore, you must not follow such statements by a test of the SQLCODE or SQLSTATE field in SQLCA.

²⁶ SQLCODE and SQLSTATE cannot be in the SQLCA when the precompiler option STDSQL(YES) is in effect. See “SQL standard language” on page 170.

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 terminal). 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 Section 7 of *DB2 Application Programming and SQL Guide*. Alternatively, you can use the (embedded) statement EXECUTE IMMEDIATE to prepare and execute a statement in one step.

The statement may 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 on page 759 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 “EXECUTE” on page 689 for rules concerning this replacement.) Once prepared, a statement can be executed several times with different values of host variables.

Parameter markers are not allowed in EXECUTE IMMEDIATE.

The successful or unsuccessful execution of the statement is indicated by setting the SQLCODE and SQLSTATE fields in SQLCA after the EXECUTE (or EXECUTE IMMEDIATE) statement. You should check the fields as described above for embedded statements.

As explained in “Authorization IDs and dynamic SQL” on page 61, the DYNAMICRULES behavior in effect determines the privilege set that is used for authorization checking when dynamic SQL statements are processed. Table 29 summarizes those privilege sets. (See Table 2 on page 61 for a list of the DYNAMICRULES bind option values that determine which behavior is in effect).

Table 29 (Page 1 of 2). DYNAMICRULES behaviors and authorization checking

DYNAMICRULES behavior	Privilege set
Run behavior	The union of the set of privileges held by each authorization ID of the process if the dynamically prepared statement is other than an ALTER, CREATE, DROP, GRANT, RENAME, or REVOKE statement. The privileges that are held by the SQL authorization ID of the process if the dynamic SQL statement is a CREATE, GRANT, or REVOKE statement.
Bind behavior	The privileges that are held by the primary authorization ID of the owner of the package or plan.

Table 29 (Page 2 of 2). DYNAMICRULES behaviors and authorization checking

DYNAMICRULES behavior	Privilege set
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.

Static invocation of a SELECT statement

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 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 setting the SQLCODE and SQLSTATE fields in 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 may 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 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 757 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 753 for rules concerning this replacement.)

The successful or unsuccessful execution of the SELECT statement is indicated by the setting of the SQLCODE and SQLSTATE fields in SQLCA after the OPEN. You should check the fields as described above for embedded statements.

Interactive invocation

IBM relational database management systems allow you to enter SQL statements from a terminal. DB2 for OS/390 provides SPUFI to prepare and execute these 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.

Checking the execution of SQL statements

An application program that contains executable SQL statements must include one or both of the following stand-alone host variables:

- SQLCODE (SQLCOD in Fortran)
- SQLSTATE (SQLSTT in Fortran)

Or,

- An SQLCA, which can be provided by using the INCLUDE SQLCA statement

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 function call.

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.

When you specify STDSQL(NO), which indicates conformance to DB2 rules, you must include an SQLCA explicitly.

SQLCODE

Regardless of whether the application program provides an SQLCA or a stand-alone variable for SQLCODE, DB2 sets SQLCODE after each SQL statement is executed. DB2 conforms to the SQL standard as follows:

- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

SQLCODE +100 indicates "no data". For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.

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.

SQLSTATE

Regardless of whether the application program provides an SQLCA or a stand-alone variable for SQLSTATE, DB2 sets SQLSTATE after each SQL statement is executed. DB2 returns values that conform to the error specification in the SQL standard.

If the application is using DB2 ODBC , the SQLSTATE returned conforms to the ODBC Version 2.0 specification.

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.

ALLOCATE CURSOR

The ALLOCATE CURSOR statement specifies 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

Identifies 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.

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 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 in the stored procedure.

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

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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 DATABASE database-name
  [ BUFFERPOOL bpname ]
  [ INDEXBP bpname ]
  [ STOGROUP stogroup-name ]
  [ CCSID ccsid-value ]
  (1)

```

Note:

¹ The same clause must not be specified more than once.

Description

DATABASE *database-name*

Identifies the database to be altered. The name must identify a database that exists at the current server.

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, 8KB and 16KB buffer pools cannot be specified.

See “Naming conventions” on page 50 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 must identify a 4KB buffer pool. See “Naming conventions” on page 50 for more details about *bpname*.

| INDEXBP cannot be specified for a work file database.

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.

STOGROUP cannot be specified for a work file 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 data base or a TEMP 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 codepoint that supports the Euro symbol replaces an existing codepoint, 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 or EBCDIC), DB2 supports only one CCSID. 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 Appendix A of *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
```

ALTER DATABASE

```
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

The ALTER FUNCTION statement changes the description of an external scalar 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 for the schema or all schemas
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

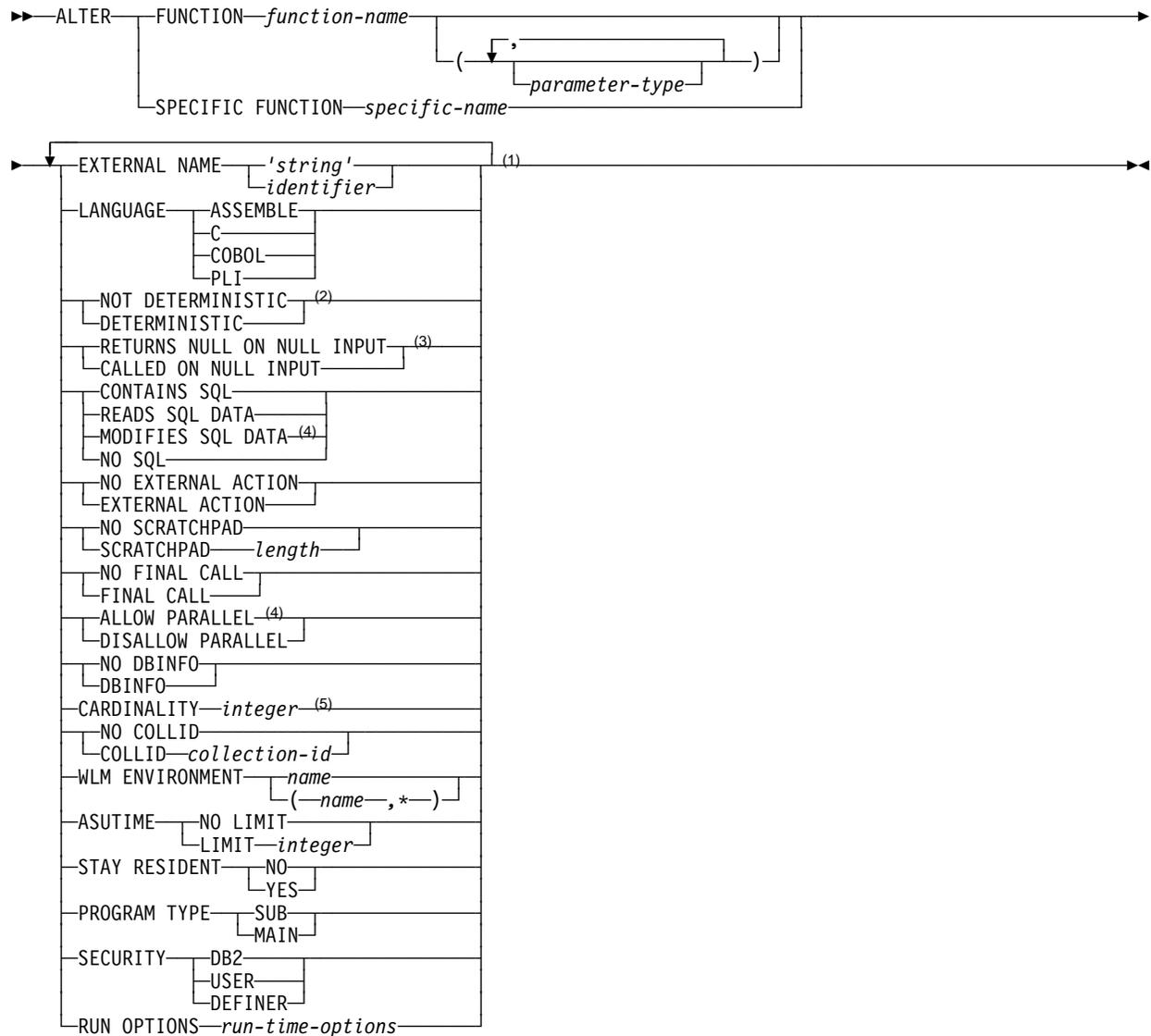
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package.

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.

Syntax

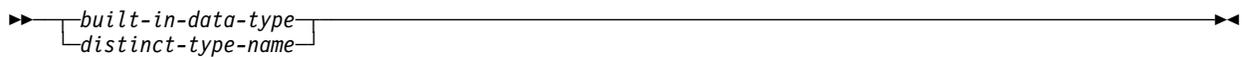
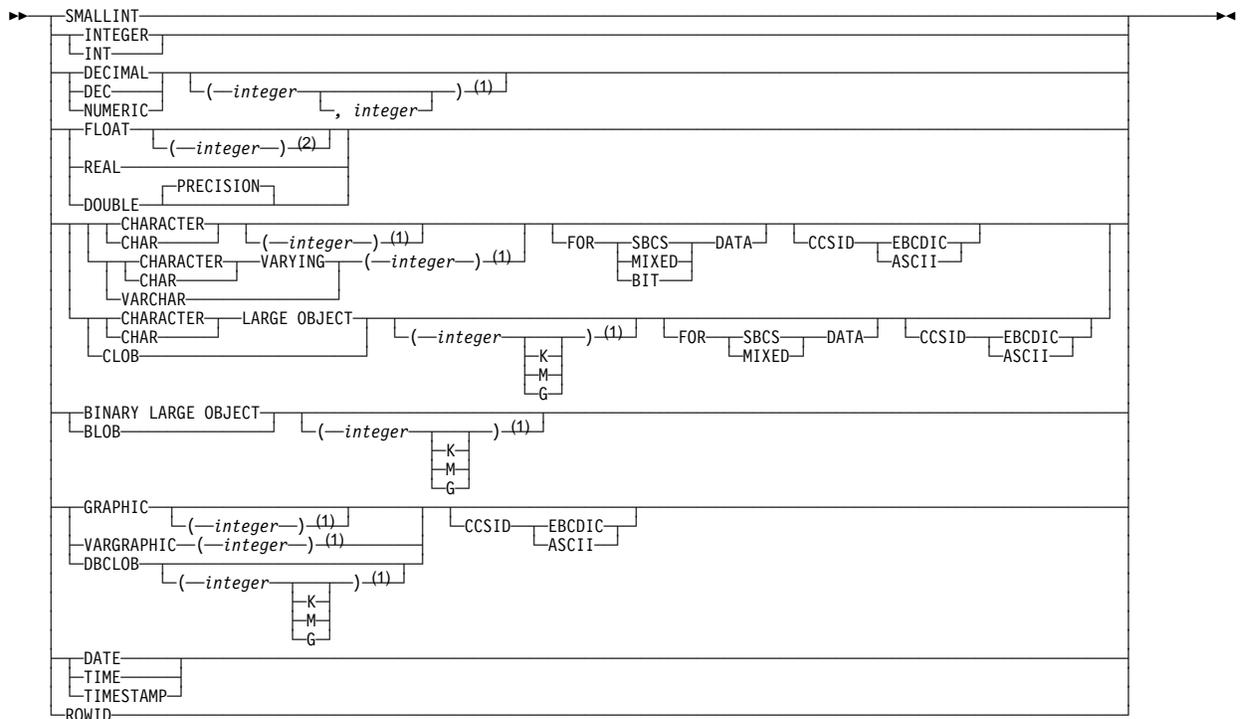


Notes:

- 1 The same clause must not be specified more than once.
- 2 Synonyms for this clause include VARIANT for NOT DETERMINISTIC, and NOT VARIANT for DETERMINISTIC.
- 3 Synonyms for this clause include NOT NULL CALL for RETURNS NULL ON NULL INPUT, and NULL CALL for CALLED ON NULL INPUT.
- 4 MODIFIES SQL DATA and ALLOW PARALLEL are not supported for *external table functions*.
- 5 CARDINALITY is not supported for *external scalar functions*.

parameter-type:**Note:**

- 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-data-type:****Notes:**

- The values that are specified for length, precision, or scale attributes must match the values that were specified when the function was created. Coding specific values is optional. Empty parentheses, (), can be used instead to indicate that DB2 ignores the attributes when determining whether data types match.
- The value that is specified does not have to match the value that was specified when the function was created because matching is based on data type (REAL or DOUBLE). $1 \leq integer \leq 21$ indicates REAL and $22 \leq integer \leq 53$ indicates DOUBLE. Coding a specific value is optional. Empty parentheses cannot be used.

Description

One of the following three clauses identifies the function to be changed.

FUNCTION *function-name*

Identifies the external function by its function name. The name is implicitly or explicitly qualified with a schema name. If the name is not explicitly qualified, it 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 plan or package was created or last rebound. If QUALIFIER was not specified, the schema name is the owner of the plan or package.
- If the statement is prepared dynamically, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

The identified function must be an external 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 external function by its function signature, which uniquely identifies the function.

function-name

Gives the function name of the external function. If the function name is not qualified, it is implicitly qualified with a schema name as described in the preceding description for FUNCTION *function-name*.

(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 specify a value or use a set of empty parentheses:

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match.
 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 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. For example:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. For a complete list of the default lengths of data types, see “CREATE TABLE” on page 570.

For data types with a subtype or encoding scheme attribute, specifying the FOR DATA clause or 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.

See “CREATE FUNCTION” on page 472 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; otherwise, an error occurs.

SPECIFIC FUNCTION *specific-name*

Identifies the external 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; otherwise, an error occurs.

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*.

The following clauses change the description of the function that has been identified to be changed.

EXTERNAL NAME *'string'* or *identifier*

Identifies the name of the MVS load module that contains the user-written code that implements the logic of the function. The external program name can be a string constant that is no longer than 8 characters or a short identifier, and must conform to the naming conventions for MVS load modules.

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 ALTER FUNCTION statement is executed. However, it must exist and be accessible by the current server when the function is invoked.

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.

PLI

The function is written in PL/I.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the function returns the same results for identical input arguments.

NOT DETERMINISTIC

The function might not return the same result for identical input arguments. The function depends on some state values that affect the results. DB2 uses this information when processing a SELECT, UPDATE, DELETE, or INSERT statement to disable merging of views that refer to the function. An example of a function that is not deterministic is one that generates random numbers, or any function that contains SQL statements.

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.

If a view refers to the function, the function cannot be changed to NOT DETERMINISTIC. To change the function, drop any views that refer to the function first.

DETERMINISTIC

The function always returns the same result for identical input arguments. DB2 can use this information to optimize view processing for SELECT, UPDATE, DELETE, or INSERT statements. An example of a deterministic function is a function that calculates the square root of the input.

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 is 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.

NO SQL, MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL

Indicates whether the function issues any SQL statements and, if so, what type. See Table 56 on page 877 for a detailed list of the SQL statements that can be executed under each data access indication.

NO SQL

The function cannot execute any SQL statements.

MODIFIES SQL DATA

The function can execute any SQL statement except those statements that are not supported in any function. Do not specify MODIFIES SQL DATA for external table functions or with ALLOW PARALLEL.

READS SQL DATA

The function cannot execute SQL statements that modify data. SQL statements that are not supported in any function return a different error.

CONTAINS SQL

The function cannot execute any SQL statements that read or modify data. SQL statements that are not supported in any function return a different error.

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 can use this information to optimize the processing of views for SELECT, UPDATE, DELETE or INSERT statements.

EXTERNAL ACTION

The function can take an action that changes the state of an object that DB2 does not manage.

Some functions with external actions can receive 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 in SELECT, UPDATE, DELETE or INSERT statements that refer to function.
- 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 refers to the function, the function cannot be changed to EXTERNAL ACTION. To change the function, drop any views 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 SCRATCHPAD or **SCRATCHPAD**

Specifies whether DB2 provides 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

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'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 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.

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.

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 473.

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 492.

ALLOW or DISALLOW PARALLEL

Specifies whether the function can be executed in 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. Do not specify ALLOW PARALLEL for external table functions or with MODIFIES SQL DATA.

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 specific information that DB2 knows is passed to the function when it is invoked.

NO DBINFO

Additional information is not passed.

DBINFO

An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run-time authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

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—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.

Do not specify CARDINALITY for external scalar functions.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is 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, the package collection is the value of the CURRENT PACKAGESET special register.

COLLID *collection-id*

The name of the package collection that is used when the function is executed.

WLM ENVIRONMENT

A long SQL identifier that identifies the *name* of the WLM (MVS 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 MVS 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 WLM environment name is a long 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 on page 553.

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 *OS/390 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 2GB. 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 remains 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.

SECURITY

Specifies how the function interacts with an external security product, such as RACF, to control access to non-SQL resources.

ALTER FUNCTION

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 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 *run-time-options*

Specifies the Language Environment run-time options to be used for the function. You must specify *run-time-options* as a character string that is no longer than 254 bytes. To replace any existing run-time options with no options, specify an empty string with RUN OPTIONS. When you specify an empty string, DB2 does not pass any run-time options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment run-time options, see *OS/390 Language Environment for OS/390 & VM Programming Reference*.

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.

Examples

Example 1: Assume that there are two functions CENTER 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 PELLOW.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 PELLOW.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 PELLOW.CENTER (CHAR(), DEC(), INTEGER)  
| RETURNS NULL ON NULL INPUT;
```

| If you use empty parentheses, DB2 ignores the length, precision, and scale
| attributes when looking for matching data types to find the function.

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

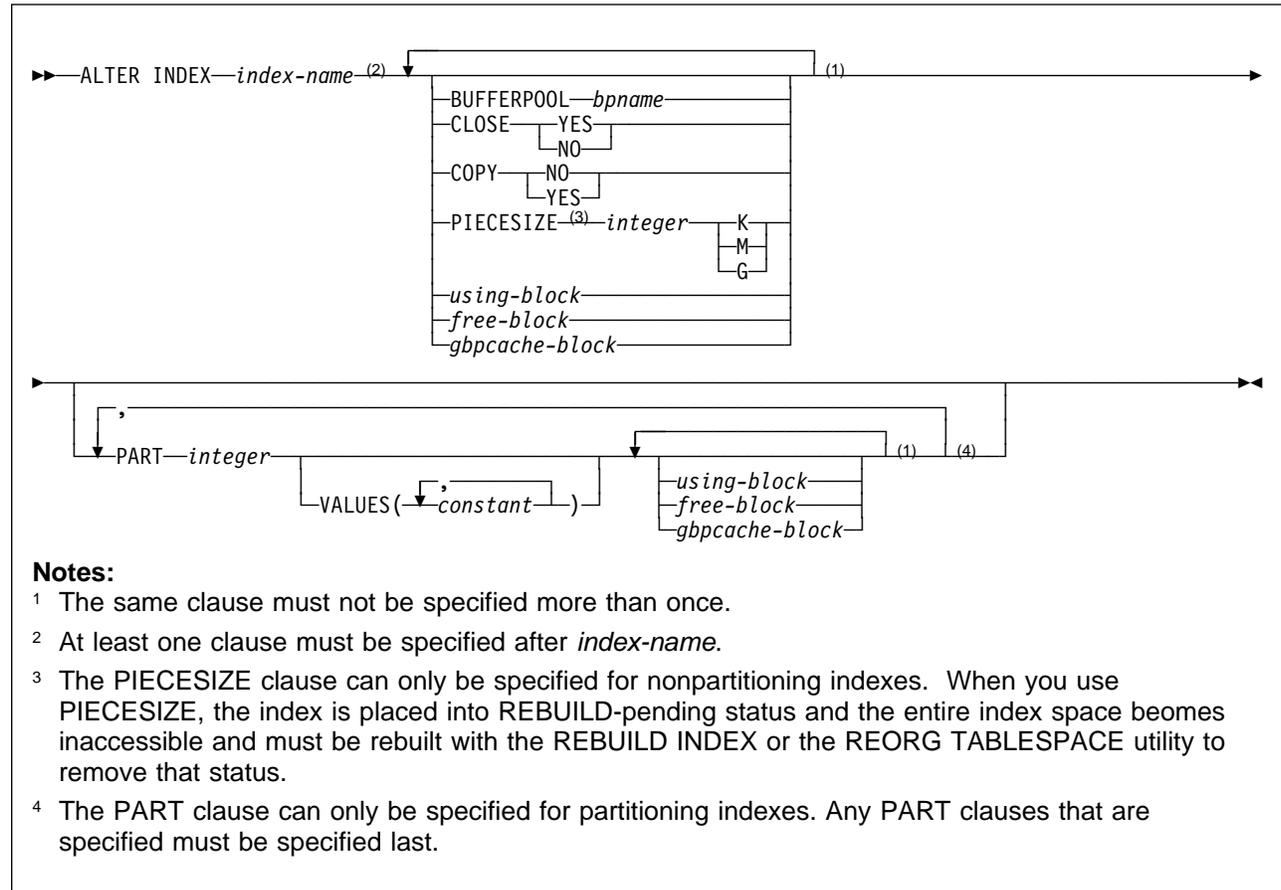
The privilege set that is defined below must include one of the following:

- 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

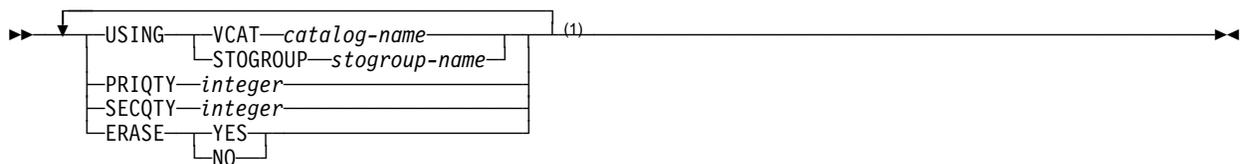
If BUFFERPOOL or USING STOGROUP is specified, additional privileges could be needed, as explained in the description of those clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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



using-block:



Note:

- The same clause must not be specified more than once.

free-block:



Note:

- The same clause must not be specified more than once.

ALTER INDEX



Description

index-name

Identifies the index to be altered. 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.

BUFFERPOOL *bpname*

Identifies the buffer pool to be used for the index. The *bpname* must identify an activated 4KB buffer pool, and the privilege set must include SYSADM authority, SYSCTRL authority, or the USE privilege for the buffer pool. See “Naming conventions” on page 50 for more details about *bpname*.

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.

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.

PIECESIZE *integer*

Specifies the maximum addressability of each piece (data set) for a nonpartitioning index. Be very aware that when you alter the PIECESIZE value, the index is placed into REBUILD-pending (PSRBD) status and 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 specified in *integer*.

K Indicates that the *integer* value is to be multiplied by 1 024 to specify the maximum piece 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 piece 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 piece size in bytes. The integer must be a power of two between 1 and 64.

Table 30 shows the valid values for piece size, which depend on the size of the table space.

Table 30. Valid values of *PIECESIZE* clause

K units	M units	G units	Size attribute of table space
254 K	-	-	-
512 K	-	-	-
1024 K	1 M	-	-
2048 K	2 M	-	-
4096 K	4 M	-	-
8192 K	8 M	-	-
16384 K	16 M	-	-
32768 K	32 M	-	-
65536 K	64 M	-	-
131072 K	128 M	-	-
262144 K	256 M	-	-
524288 K	512 M	-	-
1048576 K	1024 M	1 G	-
2097152 K	2048 M	2 G	-
4194304 K	4096 M	4 G	LARGE, DSSIZE 4 G (or greater)
8388608 K	8192 M	8 G	DSSIZE 8 G (or greater)
16777216 K	16384 M	16 G	DSSIZE 16 G (or greater)
33554432 K	32768 M	32 G	DSSIZE 32 G (or greater)
67108864 K	65536 M	64 G	DSSIZE 64 G

using-block

The components of the *using-block* are discussed below, first for nonpartitioning indexes and then for partitioning indexes.

Using Block for Nonpartitioning Indexes

For nonpartitioning 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. (A nonpartitioning index can have more than one data set if PRIQTY+118 × SECQTY is at least 2 gigabytes.)

If you specify USING, the index must be in the stopped state when the ALTER INDEX statement is executed. See Altering storage attributes on page 374 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 a short 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 set forth in Section 2 (Volume 1) of *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. 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. 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 MVS 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 omit the PRIQTY, SECQTY, or ERASE clause, the implicit value of the omitted clause is its current value when the current data set is DB2-managed. If the current data set is being changed from being user-managed to DB2-managed, the implicit value of the omitted clause is its default value, which are:

- PRIQTY 12 when PRIQTY is omitted
- SECQTY 12 when PRIQTY and SECQTY are omitted. When SECQTY is omitted but PRIQTY is specified, SECQTY is either 10% of PRIQTY or 3 times the index page size (4K), whichever is larger.
- ERASE NO when ERASE is omitted

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, the primary space allocation is at least *n* kilobytes, where *n* is:

12	If <i>integer</i> is less than 12
<i>integer</i>	If <i>integer</i> is between 12 and 4194304
4194304	If <i>integer</i> is greater than 4194304

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.)

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 are 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 is specified, the secondary space allocation is at least n kilobytes, where n is:

<i>integer</i>	If <i>integer</i> is not greater than 4194304
4194304	If <i>integer</i> is greater than 4194304

If *integer* is 0, no data set for the index can be extended.

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.)

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 *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 *DFSMS/MVS: Access Method Services for the Integrated Catalog* 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 on page 374 to determine how and when changes take effect.

USING Block for Partitioning Indexes:

For a partitioning index, there is an optional PART clause for each partition. A *using-block* can be specified at the global level or at the partition level. A *using-block* within a PART clause applies only to that partition. A *using-block* specified before any PART clauses applies to every partition except those with a PART clause with a *using-block*.

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-block* within the PART clause for the partition. Do not use more than one *using-block* in any PART clause.
- The values of PRIQTY, SECQTY, and ERASE given in a *using-block* before any PART clauses
- 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-block* within the PART clause for the partition. Do not use more than one *using-block* in any PART clause.
- The values of PRIQTY, SECQTY, and ERASE given in a *using-block* before any PART clauses
- The default values of PRIQTY, SECQTY, and ERASE, which are:
 - PRIQTY 12
 - SECQTY 12, if PRIQTY is not specified in either *using-block*, 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 on page 374 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 a short 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 *n*th data set of the index, conforming to the DB2 naming convention for data sets set forth in Section 2 (Volume 1) of *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 for nonpartitioning indexes.

End of using-block

free-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.

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.

If the index is partitioning, 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 PART clause for that partition. Do not use more than one *free-block* in any PART clause.
- The values given in a *free-block* before any PART clauses.
- The current values of FREEPAGE and PCTFREE for that partition.

End of free-block

gbpcache-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 R/W interest on the index 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 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 R/W 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.

Hiperpools are not used for indexes or partitions that are defined with GBPCACHE ALL.

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 PART clause for that partition. Do not use more than one *gbpcache-block* in any PART clause.
2. The value given in a *gbpcache-block* before any PART 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 the following indexes on catalog table SYSIBM.SYSINDEXES: DSNDXX01, DSNDXX02, DSNDXX03.

End of gbpcache-block

PART integer

Identifies a partition of the index. 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 VALUES clause.

VALUES(*constant*,...)

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 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.

You must use at least one constant after VALUES in each PART clause. You can use as many constants as there are columns in the key. The concatenation of all the constants 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.

The use of the constants to define key values is subject to these rules:

- The first constant corresponds to the first column of the key, the second constant to the second column, and so on. Each constant must have the same data type as its corresponding column.
- If a key includes a ROWID column (or a column with a distinct type that is sourced 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 constant that is specified for the corresponding ROWID column are considered.
- The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding 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'.
- Using fewer constants 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. For a descending index, it has the same effect as using the lowest possible values for all omitted columns.
- 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 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 table spaces created with the LARGE or DSSIZE options, the constants you specify after VALUES are enforced. The value specified by the constants is the highest value of the key that can be placed in the table. Any keys that are made invalid after the ALTER statement is executed are placed in a discard data set when you run REORG. If the last partition is in REORG pending, regardless of whether you changed its limiting key values, you must specify a discard data set when you run REORG.

Notes

Running utilities: You cannot execute the ALTER INDEX statement while a DB2 utility has control of the index or its associated table space.

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 Section 2 (Volume 1) of *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 915.

Examples

Example 1: Alter the index DSN8610.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 DSN8610.XEMP1
  CLOSE NO;
```

| *Example 2:* Alter the index DSN8610.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 DSN8610.XPROJ1
  BUFFERPOOL BP1
  COPY YES
  PIECESIZE 8M;
```

| *Example 3:* Alter partitioned index DSN8610.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 R/W interest on the partition.

```
ALTER INDEX DSN8610.XDEPT1
  BUFFERPOOL BP1
  CLOSE YES
  COPY YES
  USING VCAT CATLGG
  FREEPAGE 6
  PCTFREE 11
  GBPCACHE ALL
  PART 3
    USING VCAT CATLGG
    FREEPAGE 13
    PCTFREE 13,
  PART 4
    USING VCAT CATLGG
    GBPCACHE CHANGED,
  PART 5
    USING VCAT CATLGG
    FREEPAGE 25
    PCTFREE 25;
```

ALTER PROCEDURE (external)

The ALTER PROCEDURE statement changes the description of a 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 for the schema or all schemas
- SYSADM or SYSCTRL authority

The authorization ID that matches the schema name implicitly has the ALTERIN privilege on the schema.

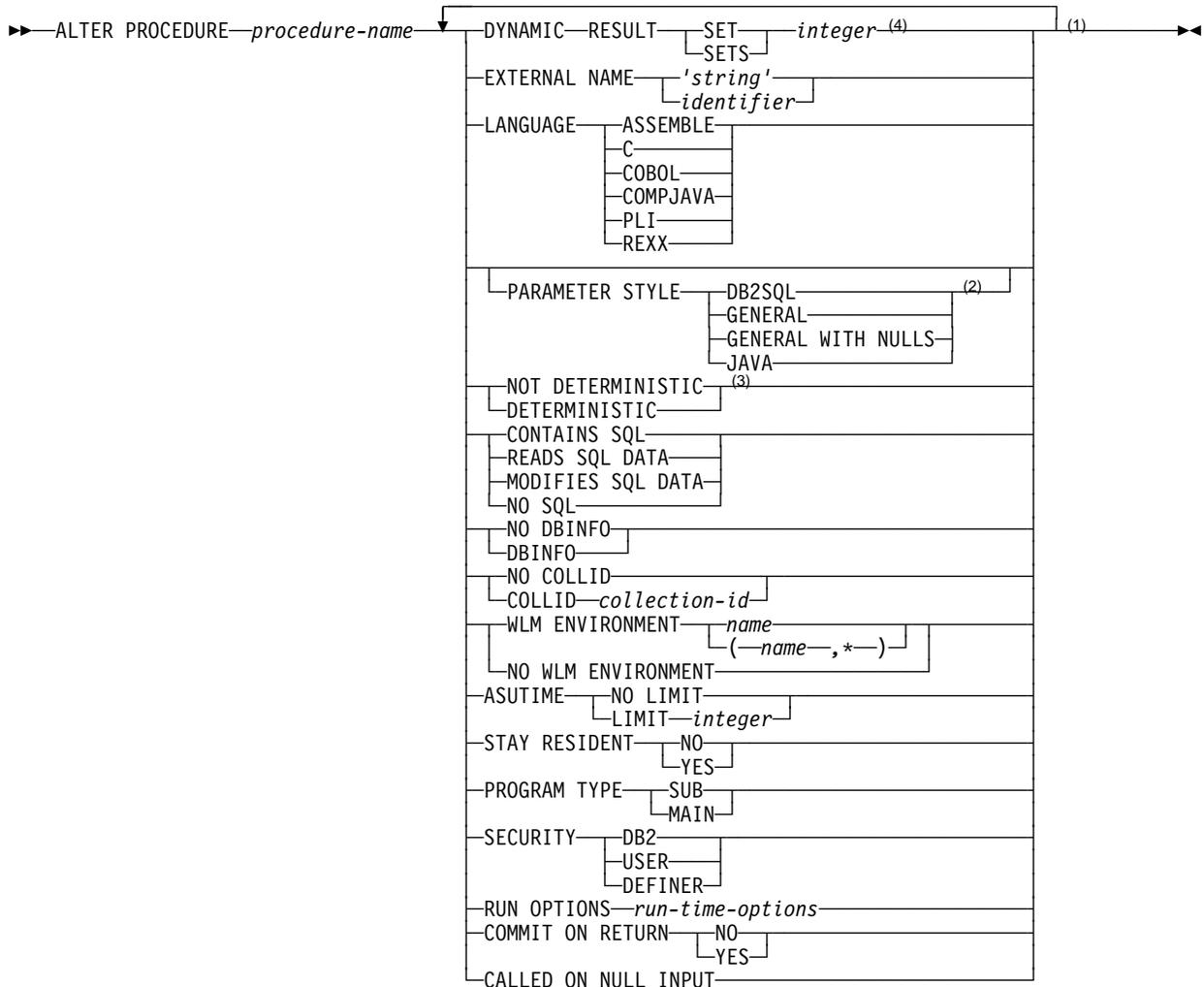
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package.

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 or
DB2-established stored procedure address space. This authorization is obtained
from an external security product, such as RACF.

Syntax



Notes:

- 1 The same clause must not be specified more than once.
- 2 Synonyms for the clause include STANDARD CALL for DB2SQL, SIMPLE CALL for GENERAL, and SIMPLE CALL WITH NULLS for GENERAL WITH NULLS.
- 3 Synonyms for the clause include VARIANT for NOT DETERMINISTIC, and NOT VARIANT is a synonym for DETERMINISTIC.
- 4 Synonyms include RESULT SET for DYNAMIC RESULT SET and RESULT SETS for DYNAMIC RESULT SETS.

Description

procedure-name

Identifies the stored procedure to be altered. The name is implicitly or explicitly qualified by a schema name. If the name is not explicitly qualified, it is implicitly qualified with a schema name according to the following rules.

ALTER PROCEDURE (external procedure)

- If the statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER option when the plan or package was created or last rebound. If QUALIFIER was not specified, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

DYNAMIC RESULT SET *integer* or **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 '*string*' or *identifier*

Identifies the program that runs when the procedure name is specified in a CALL statement.

If LANGUAGE is COMPJAVA, the name is a string constant in the format *class-name.method-name* or *package-name.class-name.method-name* that is no longer than 254 bytes. For other values of LANGUAGE, the name can be a string constant that is no longer than 8 characters or a short identifier, and must conform to the naming conventions for MVS load modules.

The program does not need to exist when the ALTER PROCEDURE statement is executed. However, it must exist and be accessible by the current server when a CALL statement for the stored procedure is issued.

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.

COMPJAVA

The stored procedure is written in Java and is a compiled program.

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 DB2SQL or NO WLM ENVIRONMENT is in effect.

PARAMETER STYLE

Identifies the linkage convention used to pass parameters to 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*.

DB2SQL

In addition to the parameters on the CALL statement, the following arguments are also passed to the stored procedure:

- A null indicator for each parameter on the CALL statement
- The SQLSTATE to be returned to DB2
- 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, an additional parameter, the DB2INFO structure, might also be passed. Do not specify DB2 SQL when LANGUAGE REXX is in effect.

GENERAL

Only the parameters on the CALL statement are passed to the stored procedure. The parameters cannot be null.

GENERAL WITH NULLS

In addition to the parameters on the CALL statement, another argument is also passed to the stored procedure. The additional argument contains a vector of null indicators for each of the parameters on the CALL statement that enables the stored procedure to accept or return null parameter values.

JAVA

The stored procedure uses a convention for passing parameters that conforms to the Java and SQLJ specifications. INOUT and OUT parameters are passed as single-entry arrays. The DBINFO structure is not passed.

JAVA can be specified only if LANGUAGE is COMPJAVA.

For REXX stored procedures (LANGUAGE REXX), GENERAL and GENERAL WITH NULLS are the only valid values for PARAMETER STYLE. For LANGUAGE COMPJAVA, JAVA is the only valid value.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the stored procedure returns the same result from successive calls with identical input arguments.

NOT DETERMINISTIC

The stored procedure might not return the same result from successive calls with identical input arguments.

DETERMINISTIC

The stored procedure returns the same result from successive calls with identical input arguments.

DB2 does not verify that the stored procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

NO SQL, MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL DATA

Indicates whether the stored procedure issues any SQL statements and, if so, what type. See Table 56 on page 877 for a detailed list of the SQL statements that can be executed under each data access indication.

NO SQL

The stored procedure cannot execute any SQL statements.

ALTER PROCEDURE (external procedure)

MODIFIES SQL DATA

The stored procedure can execute any SQL statement except those statements that are not supported in any stored procedure.

READS SQL DATA

The stored procedure cannot execute SQL statements that modify data. SQL statements that are not supported in any stored procedure return a different error.

CONTAINS SQL

The stored procedure cannot execute any SQL statements that read or modify data. SQL statements that are not supported in any stored procedure return a different error.

NO DBINFO or DBINFO

Specifies whether specific information known by DB2 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*.

NO COLLID or COLLID *collection-id*

Identifies the package collection that is 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 calling program does not use a package, the package collection is set to the value of the CURRENT PACKAGESET special register.

COLLID *collection-id*

Identifies the package collection that is 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 MVS workload manager (WLM) 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 a long identifier.

name

The WLM environment in which the stored procedure runs. 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 MVS 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 on page 553.

NO WLM ENVIRONMENT

Indicates that the stored procedure is to run in the DB2-established stored procedure address space.

Do not specify NO WLM ENVIRONMENT if the definition of the stored procedure implicitly or explicitly includes the following clauses or parameters:

- The PROGRAM TYPE SUB clause
- The SECURITY USER or SECURITY DEFINER clause
- The LANGUAGE REXX clause
- Parameters with a LOB data type or a distinct type based on a LOB data type

To change the procedure to run in the DB2-established stored procedure address space, you must have appropriate authority for the DB2-established stored procedure address space. For an example of a RACF command that provides this authorization, see Running stored procedures on page 553.

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 *OS/390 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 2GB. 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 remains 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.

ALTER PROCEDURE (external procedure)

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.

Specify PROGRAM TYPE SUB for stored procedures with a LANGUAGE
value of REXX. Do not specify PROGRAM TYPE SUB when NO WLM
ENVIRONMENT is in effect.

MAIN

The stored procedure runs as a main routine.

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.

Do not specify SECURITY USER when NO WLM ENVIRONMENT is in
effect.

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.

Do not specify SECURITY DEFINER when NO WLM ENVIRONMENT is in
effect.

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.

If you specify the RUN OPTIONS parameter for a stored procedure with a
LANGUAGE value of COMPJAVA, DB2 ignores the run-time options.

For a description of the Language Environment run-time options, see *OS/390 Language Environment for OS/390 & VM 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.

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.

CALLED ON NULL INPUT

Specifies that the stored procedure will be called even if any of the input arguments is null, making the procedure responsible for testing for null argument values. The result is the null value.

Notes

Changes are immediate: Any changes that the ALTER PROCEDURE statement cause to the definition of a procedure take effect immediately. The changed definition is used the next time that the procedure is called.

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.

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)

The ALTER PROCEDURE statement changes the description of an SQL 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 for the schema or all schemas
- # • 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 authorization ID of 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.

The authorization ID used to alter the stored procedure definition must have
appropriate authority for the specified environment (WLM environment or
DB2-established stored procedure address space) in which the procedure is
currently defined to run. This authorization is obtained from an external security
product, such as RACF.

ALTER PROCEDURE (SQL procedure)

```
# NOT DETERMINISTIC
# The stored procedure might not return the same result from successive
# calls with identical input arguments.
#
# DETERMINISTIC
# The stored procedure returns the same result from successive calls with
# identical input arguments.
#
# DB2 does not verify that the stored procedure code is consistent with the
# specification of DETERMINISTIC or NOT DETERMINISTIC.
#
# MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL DATA
# Indicates whether the stored procedure can execute any SQL statements and,
# if so, what type. See Table 56 on page 877 for a detailed list of the SQL
# statements that can be executed under each data access indication.
#
# MODIFIES SQL DATA
# The stored procedure can execute any SQL statement except those
# statements that are not supported in any stored procedure.
#
# READS SQL DATA
# The stored procedure cannot execute SQL statements that modify data.
# SQL statements that are not supported in any stored procedure return a
# different error.
#
# CONTAINS SQL
# The stored procedure cannot execute any SQL statements that read or
# modify data. SQL statements that are not supported in any stored
# procedure return a different error.
#
# NO COLLID or COLLID collection-id
# Identifies the package collection that is 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
# Indicates that the package collection for the stored procedure is the same
# as the package collection of the calling program. If the calling program
# does not use a package, the package collection is set to the value of
# special register CURRENT PACKAGESET.
#
# COLLID collection-id
# Indicates the package collection for the stored procedure is the one
# specified.
#
# WLM ENVIRONMENT
# Identifies the MVS workload manager (WLM) 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 a long 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 MVS 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 change the environment in which the stored 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 on page 553 .
#
# NO WLM ENVIRONMENT
#           Indicates that the stored procedure is to run in the DB2-established stored
#           procedure address space.
#
#           Do not specify NO WLM ENVIRONMENT if you implicitly or explicitly define the
#           stored procedure with the SECURITY USER or SECURITY DEFINER clause.
#
#           To change the procedure to run in the DB2-established stored procedure
#           address space, you must have appropriate authority for the DB2-established
#           stored procedure address space. For an example of a RACF command that
#           provides this authorization, see Running stored procedures on page 553 .
#
# 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.
#
#           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 OS/390 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 GB.
#           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 remains resident in
#           memory when the stored procedure ends.
#
# NO
#           The load module is deleted from memory after the stored procedure ends.
#
# 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.
#
#           Specify PROGRAM TYPE SUB for stored procedures with a LANGUAGE

```

ALTER PROCEDURE (SQL procedure)

```
#           value of REXX. Do not specify PROGRAM TYPE SUB when NO WLM
#           ENVIRONMENT is in effect.

#           MAIN
#           The stored procedure runs as a main routine.

#           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. You must specify run-time-options as a character string that is no
#           longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty
#           string, DB2 does not pass any run-time options to Language Environment, and
#           Language Environment uses its installation defaults.

#           For a description of the Language Environment run-time options, see OS/390
#           Language Environment for OS/390 & VM 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.

#           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.
```

```
# CALLED ON NULL INPUT  
#     Specifies that the stored procedure will be called even if any of the input  
#     arguments is null, making the procedure responsible for testing for null  
#     argument values. The result is the null value.
```

Notes

```
# Changes are immediate: Any changes that the ALTER PROCEDURE statement  
# cause to the definition of a procedure take effect immediately. The changed  
# definition is used the next time that the procedure is called.
```

```
# 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.
```

Examples

```
# 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 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.

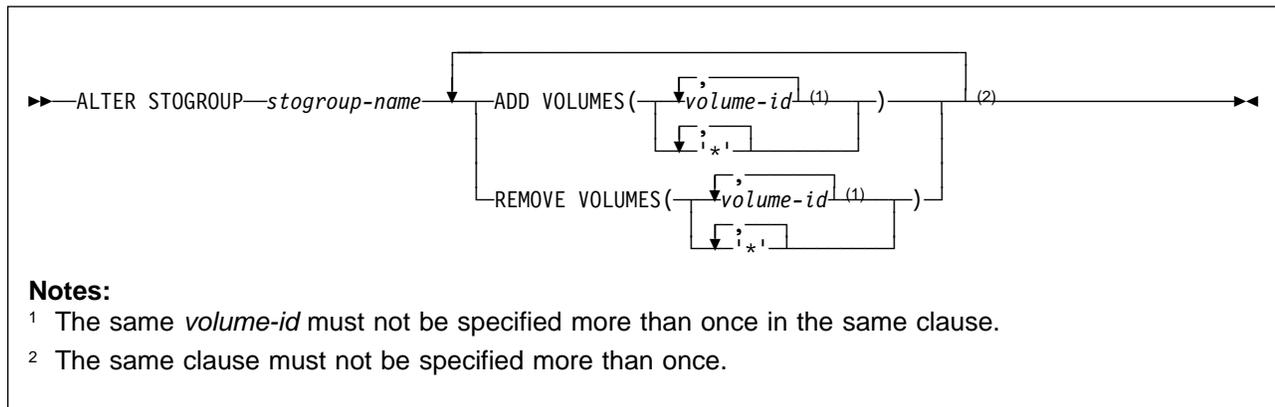
Authorization

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 authorization ID of 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). Contact your site's storage administrator to determine if the SMS Guaranteed Space attribute applies. If SMS Guaranteed Space does not apply for SMS-managed data sets, it is recommended that the VOLUMES clause be specified with one asterisk, VOLUMES(' * '). If SMS Guaranteed Space does apply, contact your site storage manager and refer to *DFSMS/MVS®: Access Method Services for the Integrated Catalog* and *DFSMS/MVS: Storage Administration Reference for DFSMSdftp™* for information on how to specify the VOLUMES clause.

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). For information about how to specify the clause with asterisks, see the above description of the ADD VOLUMES clause.

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 MVS 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. Thus, there is no point in creating a storage group with more than 133 volumes.

MVS imposes a limit on the number of volumes that can be allocated per data set: 59 at this writing. For the latest information on that restriction, see *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

Verifying volume IDs: When processing the ADD VOLUMES or REMOVE VOLUMES clause, DB2 does not check the existence of the volumes or determine the types of devices that they identify. Later, when the storage group is used to allocate or deallocate data sets, the list of volumes is passed in the specified order to Data Facilities (DFSMSdftp), which does the actual work. See Section 2 (Volume

ALTER STOGROUP

1) of *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 Section 2 (Volume 1) of *DB2 Administration Guide* for considerations for using SMS to manage data sets.

Examples

Example 1: Alter storage group DSN8G610. Add volumes DSNV04 and DSNV05.

```
ALTER STOGROUP DSN8G610
  ADD VOLUMES (DSNV04,DSNV05);
```

Example 2: Alter storage group DSN8G610. Remove volumes DSNV04 and DSNV05.

```
ALTER STOGROUP DSN8G610
  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:

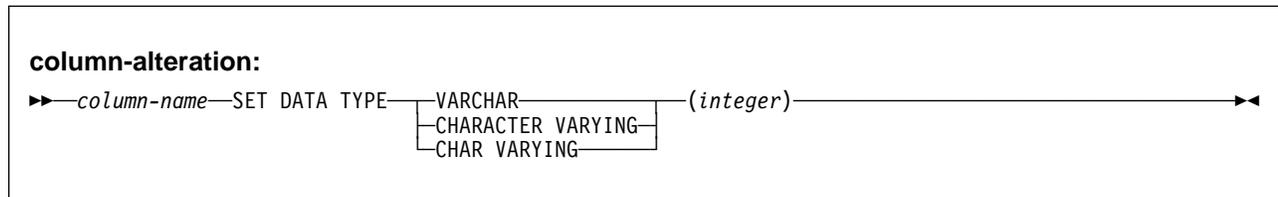
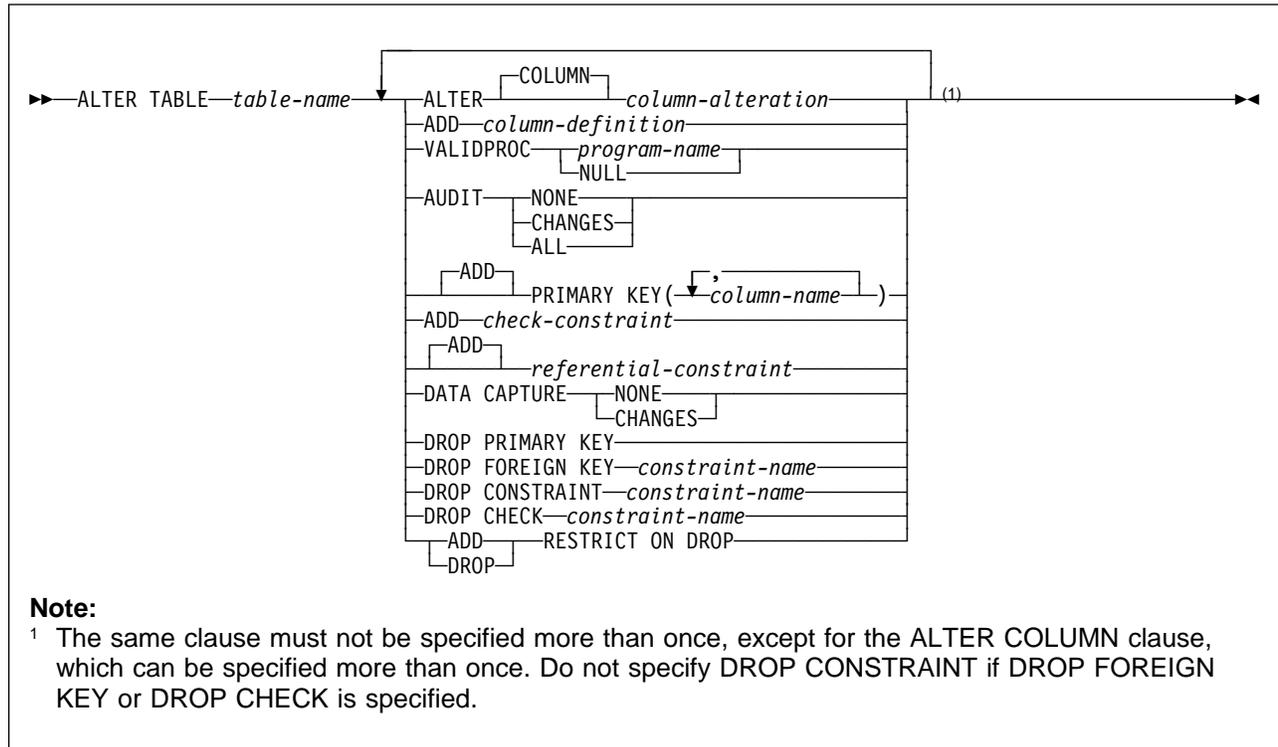
- The ALTER privilege on the table
- Ownership of the table
- DBADM authority for the database
- SYSADM or SYSCTRL authority

Additional privileges might be required if FOREIGN KEY, 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. See the description of the appropriate clauses for the details about these privileges.

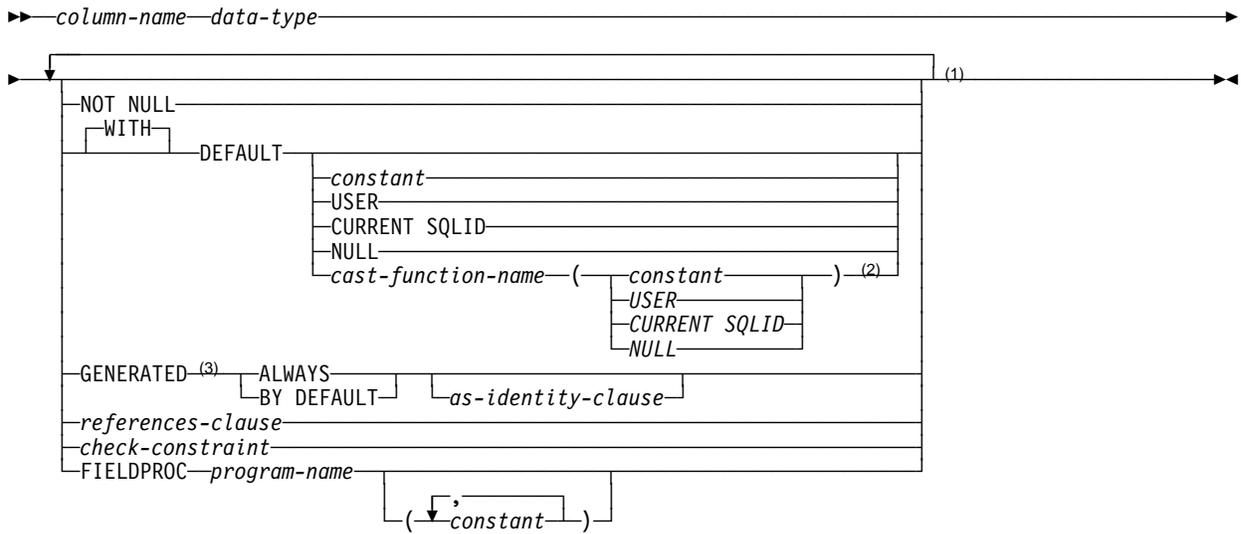
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package. If the statement is dynamically prepared, the privilege set is the union of the privilege sets that are held by each authorization ID of the process.

ALTER TABLE

Syntax



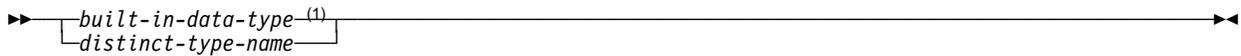
column-definition:



Notes:

- 1 The same clause must not be specified more than once.
- 2 This form of the DEFAULT value can only be used with columns that are defined as a distinct type.
- 3 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), or the column is to be an identity column.

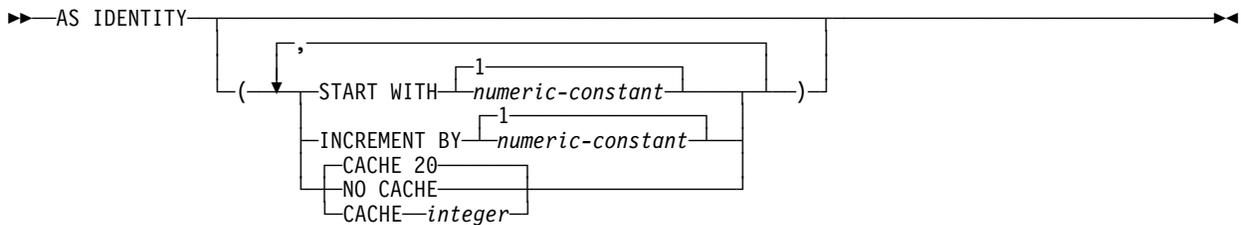
data-type:



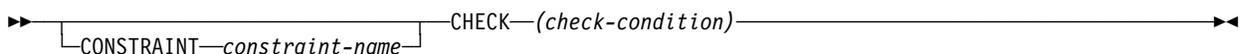
Note:

- 1 For the syntax, see built-in-data-type on page 573.

as-identity-clause:

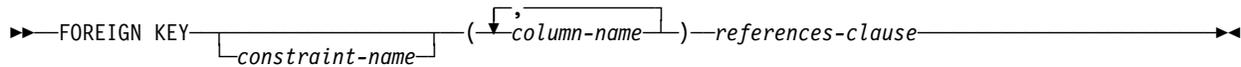


check-constraint:

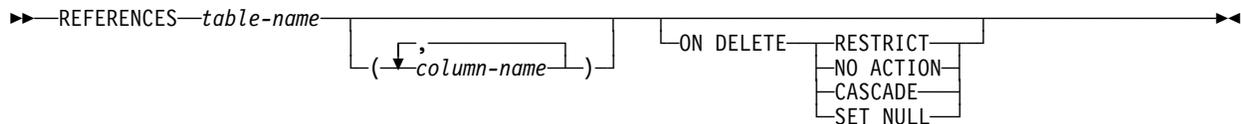


ALTER TABLE

referential-constraint:



references-clause:



Description

table-name

Identifies the table to be altered. The name must identify a table that exists at the current server. The name must not identify an auxiliary table, declared temporary table, or view. If the name identifies a catalog table, DATA CAPTURE CHANGES is the only clause that can be specified.

#

column-alteration

ALTER COLUMN *column-alteration*

Alters the definition of a column. Only the length attribute of an existing column with a VARCHAR data type can be changed. A column cannot be altered if it is used in a referential constraint or a view or has a field procedure routine. It also cannot be altered if it belongs to a table that has edit or validation routine, a table that is defined with DATA CAPTURE CHANGES, or a created temporary table.

column-name

Identifies the column to be altered. The name must not be qualified and must identify an existing column in the table that has a VARCHAR data type. The name must not identify a column that is being added in the same ALTER TABLE statement.

SET DATA TYPE VARCHAR (*integer*)

Specifies the new length for the column. The value of *integer* must be equal to or greater than the current maximum length of the column.

The new length must not make the total byte count of all columns in a row exceed the maximum row size. (For information on byte counts of columns, see Byte counts on page 592.) If the column is used in an index, the new length must not make the sum of the length attributes of the specified index columns greater than 255.

The length of more than one column can be changed in a single ALTER TABLE statement if each ALTER COLUMN clause identifies a unique column

of the table. The ALTER COLUMN clause and ADD CHECK CONSTRAINT clause can identify the same column.

End of column-alteration

column-definition

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 on page 592. A column also cannot be added to a table that has an edit procedure.

You can add a LOB column only if the table has a ROWID column. A table can have only one ROWID column. You cannot add a LOB or ROWID column to a created temporary table. 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 on page 591. For more information about adding a ROWID column, see Adding a ROWID column on page 408.

You cannot add an identity column to a table that has an identity column, or to
a created temporary table. For more information about adding an identity
column, see Adding an identity column on page 408.

column-name

Is the name 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-data-type

Specifies the data type of the column is one of the built-in data types. See “built-in-data-type” on page built-in-data-type on page 575 for detail.

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.

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

ALTER TABLE

DEFAULT must not be specified. For an identity column, although NOT
NULL can be specified, DEFAULT must not be specified.

DEFAULT

The default value assigned to the column in the absence of a value
specified on INSERT or LOAD. Do not specify DEFAULT for a ROWID
column or an identity column (a column that is defined AS IDENTITY); DB2
generates default values. If a value is not specified after the DEFAULT
keyword and the column is not nullable, 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 string	Blanks
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.

| 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 only have the default value of a
| zero-length string as listed above or 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.

USER

Specifies the value of the USER special register at the time of INSERT or LOAD as the default for the column. If USER is specified, the data type of the column must be a character string with a length greater than or equal to the length attribute of the USER special register. For existing rows, the value is that of the 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 INSERT 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 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.

In a given column definition:

- DEFAULT and FIELDPROC cannot both be specified.
- NOT NULL and DEFAULT NULL cannot both be specified.
- DEFAULT cannot be specified for a ROWID column or an identity column.
- 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.

GENERATED

Indicates that DB2 generates values for the column. You must specify GENERATED if the column is to be considered an identity column (defined with the AS IDENTITY clause) or the data type of the column is a ROWID (or a distinct type that is based on a ROWID).

ALWAYS

Indicates 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

Indicates that DB2 will generate a value for the column when a row is inserted unless a value is specified. into the table only if a value is not specified. Otherwise, DB2 uses the value that you specify.

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 statement and the LOAD utility cannot be used to add rows to the table. If the value of special register CURRENT RULES is 'STD' when the ALTER TABLE statement is processed, 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

ALTER TABLE

has a unique, single-column index; without a unique index, DB2 can
guarantee unique values only among the set of system-generated
values.
|
|

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, DECIMAL with a scale of zero, or a
distinct type based on one of these types).
An identity column is implicitly NOT NULL.

START WITH *numeric-constant*
Specifies the first value for the identity column. The value can be
any positive or negative value that could be assigned to the column
without non-zero digits existng to the right of the decimal point.
The default is 1.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity
column. The value can be any positive or negative value that is
not 0, does not exceed the value of a large integer constant, and
could be assigned to the column without any non-zero digits
existng to the right of the decimal point. The default is 1.

If the value is positive, the sequence of values for the identity
column ascends. If the value is negative, the sequence of values
descends.

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.

CACHE *integer*
Specifies the number of values of the identity column sequence
that DB2 preallocates and keeps in memory. The minimum
value that can be specified is 2, and the maximum is the
largest value that can be represented as an integer. The
default is 20.

During a system failure, all cached identity column values that
are yet to be assigned are lost, and thus, will never be used.
Therefore, the value specified for CACHE also represents the
maximum number of values for the identity column that could
be lost during a system failure.

In a data sharing environment, each member gets its own
range of consecutive values to assign. For example, if CACHE
20 is specified, DB2A might get values 1-20 for a particular
sequence, and DB2B might get values 21-40. Therefore, if
transactions from different members generate values for the
same identity column, the values that are assigned might not
be in the order in which they are requested.

The minimum value is 2. The maximum is the largest value that
can be represented as an integer. The default is CACHE 20.

NO CACHE

Specifies that values for the identity column are not
preallocated.

In a data sharing environment, use NO CACHE if you need to
guarantee that the identity values are generated in the order in
which they are requested.

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 a *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 the *references-clause* in the definition of a LOB or ROWID column because a LOB or ROWID column cannot be a foreign key.

check-constraint

The *check-constraint* of a *column-definition* has the same effect as specifying a table check constraint in a separate ADD *check-constraint* clause. For conformance with the SQL standard, a table check constraint specified in the definition of column C should not reference any columns other than C.

Do not specify a table check constraint in the definition of a LOB or ROWID 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 Appendix B (Volume 2) of *DB2 Administration Guide*. Field procedures can only be specified for short string columns that do not have a nonnull default value.

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.

End of column-definition

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 Appendix B (Volume 2) of *DB2 Administration Guide*.

The validation procedure can inhibit a load, insert, update, or delete operation on any row of the table. Before the operation takes place, the row is passed to the procedure. The values represented by any LOB columns in the table are not passed. 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.

AUDIT

Alters the auditing attribute of the table. For information about audit trace classes, see Section 3 (Volume 1) of *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 performed by each unit of recovery. 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 each unit of work of 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 AUDIT attribute is changed to CHANGES or ALL, subsequent ALTER TABLE statements will be audited if the appropriate audit trace class is active.

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 or ROWID column, and the same column must not be identified more than once. The number of identified columns must not exceed 64 and the sum of their length attributes must not exceed 255. The table must not have a primary key and the identified columns must be defined as NOT NULL.

The table must have a unique index with a 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.

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.

check-constraint

ADD check-constraint

Designates the values that specific columns of the table can contain.

CONSTRAINT *constraint-name*

Names the table check constraint. The constraint name must be different from the names of any existing referential or check constraints 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 table check constraint.

CHECK (*check-condition*)

Defines a table check constraint. 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 or ROWID columns.
- 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 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
 - Quantified predicates
 - EXISTS predicates
- If a check-condition refers to a long string column, 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

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 Section 2 (Volume 1) of *DB2 Administration Guide*.

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.

_____ End of check-constraint _____

_____ referential-constraint _____

FOREIGN KEY *constraint-name (column-name,...)* **references-clause**

Specifies a referential constraint with the specified *constraint-name*. A name is generated if a *constraint-name* is not specified. The generated name is derived from the name of the first column of the foreign key in the same way that the name of an implicitly created table space is derived from the name of a table except that the scope of uniqueness of a *constraint-name* is the table. If specified, *constraint-name* must be different from the names of any existing referential or check constraints on the table.

Let T1 denote the object table of the ALTER TABLE statement.

The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T1 except a LOB or ROWID column, and the same column must not

be identified more than once. 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.

End of referential-constraint

references-clause

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. Let T2 denote the identified parent table and let T1 denote the table being altered (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 or a ROWID column. The same column must not be identified more than once.

The list of column names must be identical to the list of column names in a unique index (UNIQUE RULE in SYSINDEXES will be R, P, C, or U). The column names must be specified in the *same order* as in the unique index on T2.

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.²⁷ A table in a segmented table space is populated if the table is not empty. A table in a nonsegmented table space is considered populated if the table space has ever contained any records.

²⁷ The check pending status prevents further updating or reading by other SQL applications. It does not affect the application process that issues ALTER TABLE. However, we do not recommend that a process create or alter a permanent table and then access it.

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 integrity” on page 23.

If T1 and T2 are the same table, CASCADE or NO ACTION must be specified. SET NULL must not be specified unless some column of the foreign key allows null values. Also, SET NULL must not be specified if any nullable column of the foreign key is a column of the key of a partitioning index. The default value for the rule depends on the value of the CURRENT RULES special register when the CREATE 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 .

End of references-clause

DATA CAPTURE

Specifies whether the logging of SQL INSERT, UPDATE, and DELETE operations on the table is augmented by additional information. For guidance on intended uses of the expanded log records, see:

- The description of data propagation to IMS in *DataPropagator NonRelational MVS/ESA Administration Guide*
- 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 Appendix C (Volume 2) of *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 columns is not available.

For details about the recording of additional data for logged updates to catalog tables, see “Notes” on page 408.

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.

If the table has a primary index, its description is changed to indicate that it is not a primary index.

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 CONSTRAINT *constraint-name*

Drops the constraint *constraint-name*. The constraint-name must identify an existing check constraint or referential constraint defined on the table. If the constraint-name identifies a referential constraint in which the table is the dependent table, then the privilege set must include the ALTER or REFERENCES privilege on the parent table of that relationship.

DROP CONSTRAINT must not be used on the same ALTER TABLE statement as DROP FOREIGN KEY or DROP CHECK.

DROP CHECK *constraint-name*

Drops the check constraint *constraint-name*. The constraint-name must identify an existing check constraint defined on the table.

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.

Notes

Restrictions for adding columns: When using ALTER TABLE, you cannot add:

- A column to a table that has an edit procedure
- A LOB column unless the table has a ROWID column
- A ROWID column to a table that already has a ROWID column
- An identity column to a table that already has an identity column
- A LOB, ROWID, or identity column to a created temporary table

Because a distinct type is subject to the same restrictions as its source type, all the syntactic rules that apply to LOB and ROWID columns apply to distinct type columns that are sourced on LOBs and row IDs. For example, if a table has 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.

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. 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.

Altering the length of a column: Only the length of VARCHAR columns can be changed. When changing the length of a column, be aware of the following information about indexes, limit keys, check constraints, and invalidation.

- **Restrictions.** The length of a VARCHAR column cannot be changed if any of the following conditions are true:
 - The column is referenced in a referential constraint or view.
 - The column has a field procedure routine
 - The table has an edit or validation routine
 - The table is defined with DATA CAPTURE CHANGES
 - The table is a created temporary table
- **Indexes.** After the ALTER TABLE statement is executed, each index on the table with a key that includes a column whose length was increased remains available. However, SQL operations against such an index are not allowed until the changes from the ALTER TABLE statement are committed.

The maximum number of *distinct* alters that increase the index key length is
 # sixteen or less. If the maximum number of alters is exceeded, SQLCODE -148
 # is returned, and the index must be reorganized or rebuilt. An alter is considered
 | distinct when it occurs in a different unit of work than the previous alter. For
 | example, changing an index column length, committing database changes, and
 | changing the column length of that index column or another index column
 | counts as two distinct alters. Whereas, changing an index column length twice
 | before committing any changes counts as one distinct alter; the second
 | changes replace the first because it was in the same commit scope. Changing
 | the length of two different index columns before committing the changes also
 | counts as one distinct alter.

- *Length of partitioned index keys.* When a table is altered and the length of a column in the index 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 column is increased.

However, if the index was created in a release prior to Version 6 of DB2 for OS/390, when the maximum length of the limit key was 40 bytes instead of 255 bytes, the limit key length is not always changed. Its length changes only if the new column length would require that data be relocated to a different partition. Therefore, in general, DB2 changes the length of the limit key if the column being altered is not the last column in the partitioning index, and the sum of the lengths of the preceding columns in the index and the existing length of the columns being altered is less than 40 bytes. The length of the limit key is increased by the same amount that the length of column is increased.

- *Check constraints.* If a table check constraint refers to the column being altered, the length of the column is also changed in the check constraint.
- *Invalidation.* When a table is altered to change the length of a VARCHAR column, all plans, packages, and dynamic cached statements that reference the table are invalidated.

Invalidation of plans and packages: When a table is altered, all the plans and packages that refer to the table are invalidated if any one of the following conditions is true:

- 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.
- The length attribute of a VARCHAR column is changed.
- The table is a created temporary table.

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.

Views: Adding a column to a table has no effect on existing views.

Order of processing of clauses: When there is more than one clause, they are
 # processed in the following order: VALIDPROC, AUDIT, DATA CAPTURE, DROP
 # clauses, ALTER COLUMN, and ADD clauses.

ALTER TABLE

Running utilities: You cannot execute the ALTER TABLE statement while a utility has control of the table space that contains the table.

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.

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.

Activity to the catalog that is caused by DB2 utilities is not captured. For example, log records from executing a utility on a catalog table, to record the event of executing a utility, or for catalog changes that result from executing the RUNSTATS utility on a user table will not have data capture information.

Examples

Example 1: Column DEPTNAME in table DSN8610.DEPT was created as a VARCHAR(36). Increase its length to 50 bytes. Also, add the column BLDG to the table DSN8610.DEPT. Describe the new column as a character string column that holds SBCS data.

```
ALTER TABLE DSN8610.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 DSN8610.EMP.

```
ALTER TABLE DSN8610.EMP
  VALIDPROC DSN8EAEM;
```

Example 3: Disassociate the current validation procedure from the table DSN8610.EMP. After the statement is executed, the table no longer has a validation procedure.

```
ALTER TABLE DSN8610.EMP
  VALIDPROC NULL;
```

Example 4: Define ADMRDEPT as the foreign key of a self-referencing constraint on DSN8610.DEPT.

```
ALTER TABLE DSN8610.DEPT
  FOREIGN KEY(ADMRDEPT) REFERENCES DSN8610.DEPT ON DELETE CASCADE;
```

Example 5: Add a check constraint to the table DSN8610.EMP which checks that the minimum salary an employee can have is \$10,000.

```
ALTER TABLE DSN8610.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;
```

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

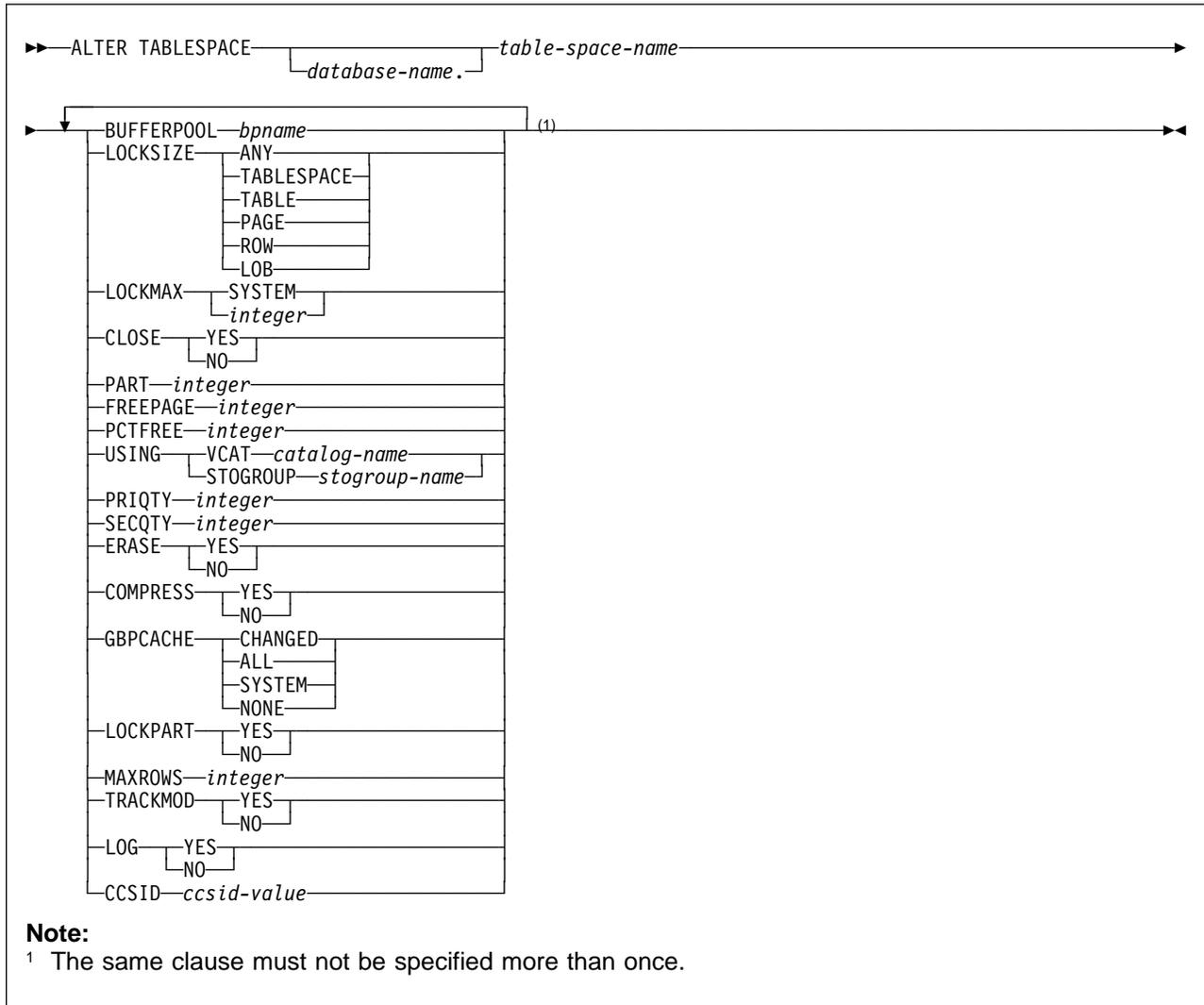
The privilege set that is defined below must include at least one of the following:

- Ownership of the table space
- DBADM authority for its database
- SYSADM or SYSCTRL authority

If 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 authorization ID of 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

database-name.table-space-name

Identifies the table space 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 table space of a work file database, the database must be in the stopped state. If you identify a partitioned table space, you can use the PART clause as explained below.

BUFFERPOOL *bpname*

Identifies the buffer pool to be used for the table space. The *bpname* must identify an activated buffer pool with the same page size as the table space. See "Naming conventions" on page 50 for more details about *bpname*.

The privilege set must include SYSADM or SYSCTRL authority or the USE privilege for the buffer pool.

ALTER TABLESPACE

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 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.

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 or a TEMP 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 nonsegmented, 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.

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)
- In all other cases, LOCKSIZE has no effect on S until S is rebound.

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 TEMP 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.

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 TEMP database.

NO

Eligible for closing after all eligible CLOSE YES data sets are closed.

For a table space in a TEMP database, DB2 uses CLOSE NO regardless of the value specified

PART *integer*

Identifies a partition of the table space. 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 or for a LOB table space. You must use this clause for a partitioned table space if you use any of the following clauses:

FREEPAGE
PCTFREE
USING
PRIQTY

ALTER TABLESPACE

SECQTY
COMPRESS
ERASE
GBPCACHE
TRACKMOD

In this case, the changes specified by these clauses apply only to the identified partition of the table space.

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, or a table space in a work file database or a TEMP 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.

The change to the description of the table space or partition has no effect until it is loaded or 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, or a table space in a work file database or a TEMP database.

This change to the description of the table space or partition has no effect until it is loaded or reorganized.

USING

Specifies whether a data set for the table space or partition is managed by the user or managed by DB2. If the table space is partitioned, USING applies to the data set for the partition identified in the PART clause. 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 $PRIQTY+118 \times SECQTY$ 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 on page 422 to determine how and when changes take effect.

VCAT *catalog-name*

Specifies a user-managed data set with a name that starts with *catalog-name*. You must specify the catalog name in the form of a short 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 table space is applied, the integrated catalog facility catalog must contain an entry for the data set conforming to the DB2 naming conventions set forth in Section 2 (Volume 1) of *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. 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. 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 MVS 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 DB2-managed:

- Omission of the PRIQTY clause is an implicit specification of the current PRIQTY value.
- Omission of the SECQTY clause is an implicit specification of the current SECQTY value.
- Omission of the ERASE clause is an implicit specification of the current ERASE rule.

If you specify USING STOGROUP to convert from user-managed data sets to DB2-managed data sets:

- Omission of the PRIQTY clause is an implicit specification of PRIQTY 12, 24, 48, or 96 for a table space with 4KB, 8KB, 16KB, or 32KB pages, respectively (For LOB table spaces, the respective PRIQTY values are 200, 400, 800, and 1600).
 - Omission of the SECQTY and PRIQTY clauses is an implicit specification of SECQTY 12, 24, 48, or 96 for a table space with 4KB, 8KB, 16KB, or 32KB pages, respectively. (For LOB table spaces, the respective SECQTY values are 200, 400, 800, and 1600).
- If SECQTY is omitted and PRIQTY is specified, SECQTY is either 10% of PRIQTY or 3 times the page size of the table space, whichever is larger. (For LOB table spaces, SECQTY is either 10% of PRIQTY or 50 times the page size of the table space, whichever is larger.)
- 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.

ALTER TABLESPACE

If PRIQTY is specified, 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 pages sizes, if *integer* is less than 200, n is 200.
- For 8KB pages sizes, if *integer* is less than 400, n is 400.
- For 16KB pages sizes, if *integer* is less than 800, n is 800.
- For 32KB 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 USING STOGROUP is specified and PRIQTY is omitted, the value of PRIQTY is the default specified in the description of USING STOGROUP.

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 on page 422 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 is specified, the secondary space allocation is at least n kilobytes, where n is the value of *integer* with the following exceptions:

If *integer* is greater than 4194304, n is 4194304. A value of 0 for *integer* indicates that no data set can be extended.

For LOB table spaces the exceptions are:

- 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.

If USING STOGROUP is specified and SECQTY is omitted, the value of SECQTY is the default specified in the description of USING STOGROUP.

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 *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

#

See Altering storage attributes on page 422 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. See Altering storage attributes on page 422 to determine how and when changes take effect.

COMPRESS

Specifies whether data compression applies to the rows of the table space or partition. Do not specify COMPRESS for a LOB table space.

|

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 Section 2 (Volume 1) of *DB2 Administration Guide* for more information about the dictionary and data compression.

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 or TEMP database, but it is ignored. Do not

#

ALTER TABLESPACE

specify GBPCAHCE for a table space in a work file database or in a TEMP
database in either environment (data sharing or not).

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 DB2s have any interest in the page set, no pages are cached in the group buffer pool.

Hiperpools are not used for table spaces or partitions that are defined with GBPCACHE ALL.

| 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.

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. You cannot alter the GBPCACHE value for the following
table spaces:

- # • DSNDB06.SYSDBASE
- # • DSNDB06.SYSDBAUT
- # • DSNDB06.SYSPKAGE
- # • DSNDB06.SYSPLAN
- # • DSNDB06.SYSUSER (exception: the attribute can be altered if
authorization includes installation SYSADM authority.)

LOCKPART

Indicates whether selective partition locking (SPL) is to be used when locking a partitioned table space.

- YES** If all the conditions that are required for SPL are met, specifies that only the partitions accessed will be locked. If all the conditions that are required for SPL are not met, every partition of the table space is locked. LOCKPART YES is not allowed with LOCKSIZE TABLESPACE.
- NO** Specifies that selective partition locking is not used. The table space is locked with a single lock on the last partition. This has the effect of locking all partitions in the table space.

To alter the LOCKPART option, you must stop the entire table space with the STOP DATABASE command. Use LOCKPART only for partitioned table spaces.

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 may 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 in a work file database, or the following table spaces for catalog tables:

- DSND06.SYSDBASE
- DSND06.SYSDBAUT
- DSND06.SYSGROUP
- DSND06.SYSPLAN
- DSND06.SYSVIEWS

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. For a table space in a TEMP database, DB2 uses TRACKMOD NO regardless of the value specified.

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.

LOG

Specifies whether changes to a LOB column in the table space are to be written to the log. Use LOG only for a LOB table space.

YES

Indicates that changes to a LOB column are to be written to the log. You cannot use YES if the auxiliary table in the table space stores a LOB column that is greater than 1 gigabyte in length.

When you change the value of LOG to YES, the LOB table space is placed in copy pending status.

ALTER TABLESPACE

NO

Indicates that changes to a LOB column are not to be written to the log.

LOG NO has no effect on a commit or rollback operation; the consistency of the database is maintained regardless of whether the LOB value is logged. All committed changes and changes that are rolled back reflect the expected results.

Even when LOG NO is specified, changes to system pages and to the auxiliary index are logged. During the log apply operation of the RECOVER utility, LPL recovery, or GPB recovery, all LOB values that were not logged are marked invalid and cannot be accessed by a SELECT or FETCH statement. Invalid LOB values can be updated or deleted.

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 349 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 or a table space in a TEMP database.

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 Section 2 (Volume 1) of *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.

Altering table spaces for DB2 catalog tables: For details on altering options on catalog tables, see “SQL statements allowed on the catalog” on page 915.

Examples

Example 1: Alter table space DSN8S61E in database DSN8D61A. CLOSE NO means that the data sets of the table space are not to be closed when there are no current users of the table space.

```
ALTER TABLESPACE DSN8D61A.DSN8S61E
CLOSE NO;
```

Example 2: Alter table space DSN8S61D in database DSN8D61A. BP2 is the buffer pool associated with the table space. PAGE is the level at which locking is to take place.

```
ALTER TABLESPACE DSN8D61A.DSN8S61D
BUFFERPOOL BP2
LOCKSIZE PAGE;
```

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a stored procedure.

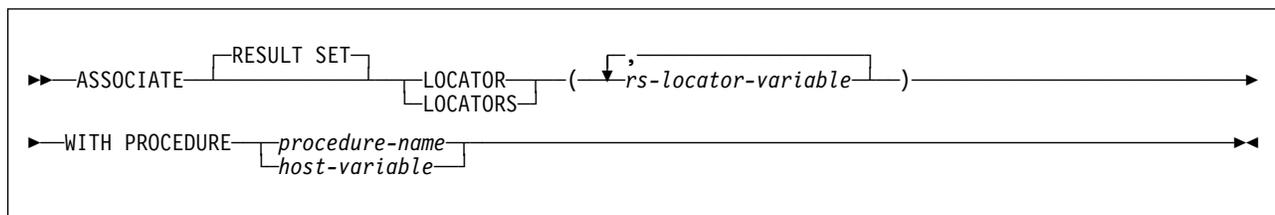
Invocation

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.

One result set locator variable is required for each result set that the stored procedure returns. If the stored procedure returns fewer result sets than the number of result set locator variables specified, the extra variables are assigned a value of 0.

WITH PROCEDURE *procedure-name* or *host-variable*

Identifies the stored procedure that returned result set locators by the specified procedure name or the procedure name contained in the host variable.

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 a long identifier that contains the location name that identifies the DBMS at which the procedure is stored. The second part is a short identifier that contains the schema name of the stored procedure. The last part is a long 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 one implicit qualifier. The implicit qualifier is the location name of the current server. An

implicit schema name is not needed as a qualifier. Successful execution of the ASSOCIATE LOCATOR statement only requires that the unqualified procedure name in the statement is the same as the procedure name in the most recently executed CALL statement that was specified with an unqualified procedure name. (The implicit schema name for the unqualified name in the CALL statement is not considered in the match.) The rules for how the procedure name must be specified are described below.

If a host variable is used:

- 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 application server. Regardless of the application 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

When the ASSOCIATE LOCATORS statement is executed, the procedure name or specification must identify a stored procedure that the requester has already invoked using the CALL statement.

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 name was specified on the CALL statement, you must use a two-part name in the ASSOCIATE LOCATORS statement. However, there is one condition under which the names do not have to match. If the CALL statement was made with a three-part name and the current server is the same as the location in the three-part name, you can omit the location name and specify a two-part name.

Notes

More than one locator can be assigned to a result set. You can issue the same ASSOCIATE LOCATORS statement more than once with different result set locator variables.

If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is less than the number of locators returned by the stored procedure, all variables in the statement are assigned a value, and a warning is issued.

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 variables are assigned a value of 0.

The ASSOCIATE LOCATORS statement assigns result set locator values from the SQLVAR sections of the SQLDA to result set locator variables. For languages other than REXX, the first SQLDATA field is assigned to the first locator variable, the second SQLDATA field to the second locator variable, and so on. For REXX, the first SQLLOCATOR field is assigned to the first locator variables, the second SQLLOCATOR field to the second locator variable, and so on.

ASSOCIATE LOCATORS

If a stored procedure is called more than once with a one-part name at the same location, only the most recent result sets are accessible.

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 a host variable declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

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 687.

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
- BEGIN DECLARE SECTION and END DECLARE SECTION statements must be paired and must not be nested.
- Host variable declaration sections must not contain statements other than host variable declarations or SQL INCLUDE statements that include host variable declarations.

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 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 OS/390.
- The stored procedure package and most packages that run under the stored procedure
 - The authorization that is required for which packages is explained in detail below under "Authorization to execute packages under the stored procedure."

Authorization to execute the stored procedure

The authorization ID 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* (ODBC applications use this form of the CALL statement), the authorization ID 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 and the privilege set that is held by that authorization ID:

Run behavior	The privilege set is the union of the set of privileges held by the SQL authorization ID and each authorization ID 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 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 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.

For a list of the DYNAMICRULES values that specify run, bind, define, or invoke behavior, see Table 2 on page 61.

Authorization to execute packages under the stored procedure

The authorization that is required to run the stored procedure package and any packages that are used under the stored procedure apply to any form of the CALL statement as follows:

- **Stored procedure package**

One of the authorization IDs that are defined below under Set of authorization IDs on page 430 must have at least one of the following on the stored procedure package:

- The EXECUTE privilege on the package
- Ownership of the package
- PACKADM authority for the package's collection
- SYSADM authority

A PKLIST entry is not required for the stored procedure package.

- **Packages other than user-defined function, trigger, and stored procedure packages**

One of the authorization IDs that are defined below under Set of authorization IDs on page 430 must have at least one of the following on any packages other than user-defined function and trigger packages that are used under the stored procedure:

- The EXECUTE privilege on the package
- 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.

- **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 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 of the SQL statement that activates the trigger must have EXECUTE authority on the trigger. For more information about the EXECUTE

CALL

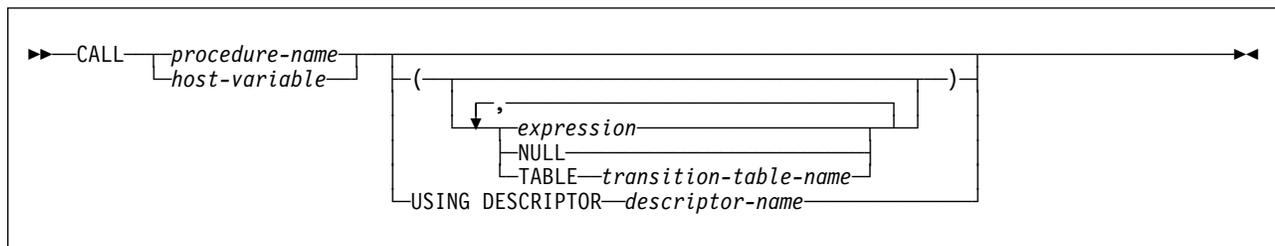
authority for user-defined functions, triggers, and user-defined function packages, see Section 3 of *DB2 Administration Guide*.

PKLIST entries are not required for any user-defined function packages or trigger packages that are used under 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 other than user-defined function and trigger packages as described above:

- The owner (the definer) of the stored procedure
- The owner of the plan that contains the statement that invokes the package if the application is local, the application is distributed and DB2 for OS/390 is both the requester and the server, or the application uses Recoverable Resources Management Services attachment facility (RRSAF) and has no plan.
- The owner of the package that contains the statement that invokes the package if the application is distributed and DB2 for OS/390 is the server but not the requester
- 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*

Syntax



Description

procedure-name or *host-variable*

Identifies the stored procedure to call. The procedure name can be specified as a character string constant or within a host variable.

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 a long identifier that contains the location name that identifies the DBMS at which the procedure is stored. The second part is a short identifier that contains the schema name of the stored procedure. The last part is a long 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 application server. If the server is DB2 for OS/390, 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 literal 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.
- The create timestamp for the stored procedure must be older than the bind timestamp for the package or plan in which the procedure is invoked.

If the list of schemas in the SQL path is exhausted before the procedure name is resolved, an error is returned.

If a host variable is used:

- 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 application server. Regardless of the application 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 the three-part procedure name.

If the server is DB2 for OS/390, 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 application server.

CALL

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.

Parameters (*expression*, **NULL**, **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 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.

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 84.

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.

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 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 131, *expression*
cannot include a column name or column function or a user-defined
function that is sourced on a column function.
- # • INOUT or OUT *expression* can only be a single host 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. For information
about creating a trigger, see “CREATE TRIGGER” on page 615. 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.

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 890.

CALL

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 907 for how to represent *descriptor-name* in C.

Notes

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 application server can be the local DB2. In which case, packages are still required.

All values of all parameters are passed from the application requester to the application 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 6 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.

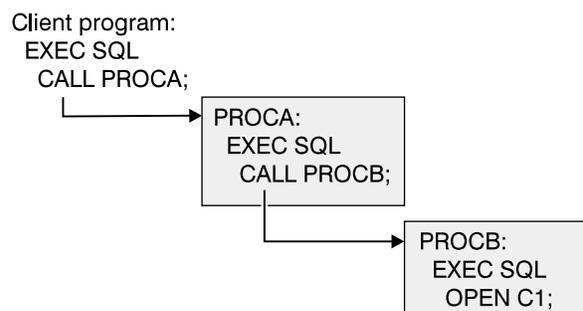


Figure 6. Nested CALL statements

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. A stored procedure can call another stored procedure only if
| they execute in the same type of address space; they must both execute in a
| DB2-established address space or in a WLM-established address space.

Example

| 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);
```

CLOSE

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.

Authorization

See “DECLARE CURSOR” on page 634 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 634. When the CLOSE statement is executed, the cursor must be in the open state.

Notes

Any open cursors of an application process are implicitly closed at the termination of a unit of work. However, explicitly closing cursors as soon as possible can improve performance. CLOSE does not cause a commit or rollback operation.

The cursor could have been allocated. See “ALLOCATE CURSOR” on page 346.

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 DSN8610.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 ON

The COMMENT ON statement adds or replaces comments in the descriptions of various objects in the DB2 catalog at the current server. The objects that can have comments are aliases, columns, distinct types, stored procedures, tables, triggers, user-defined functions, and views.

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, or column, 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

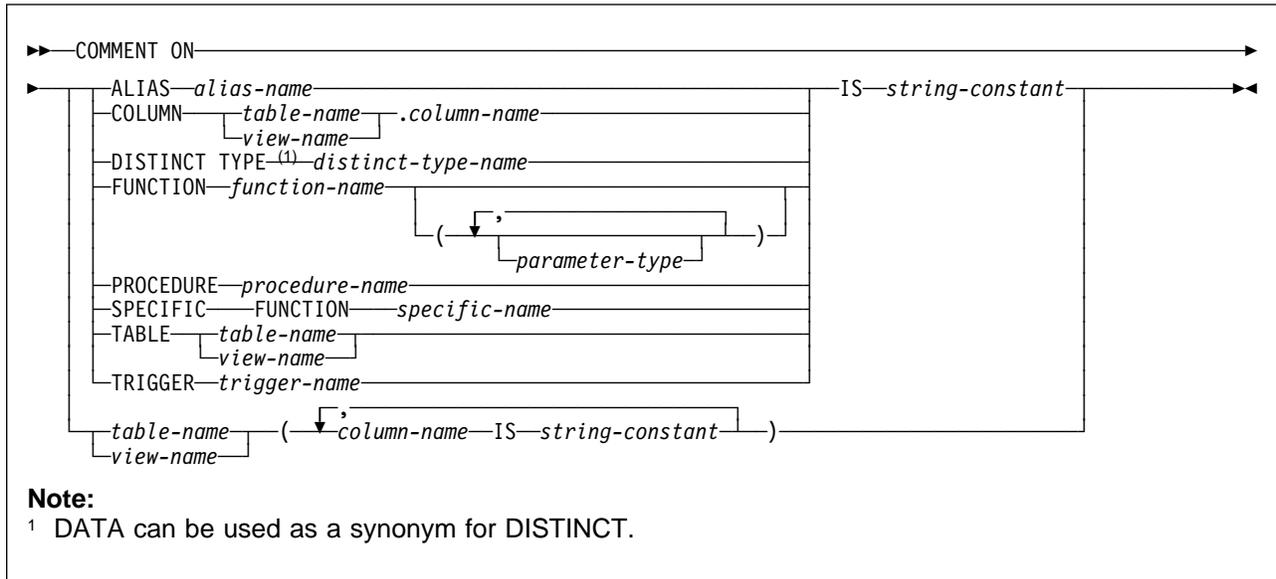
For a comment on a distinct type, stored procedure, trigger, or user-defined function, the privilege set that is defined below must include at least one of the following:

- Ownership of the distinct type, stored procedure, trigger, or user-defined function
- The ALTERIN privilege for the schema or all schemas (for the addition of comments)
- 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 authorization ID of 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 29 on page 342. (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 61.)

Syntax



parameter-type

►► *data-type* [AS LOCATOR ⁽¹⁾] ►►

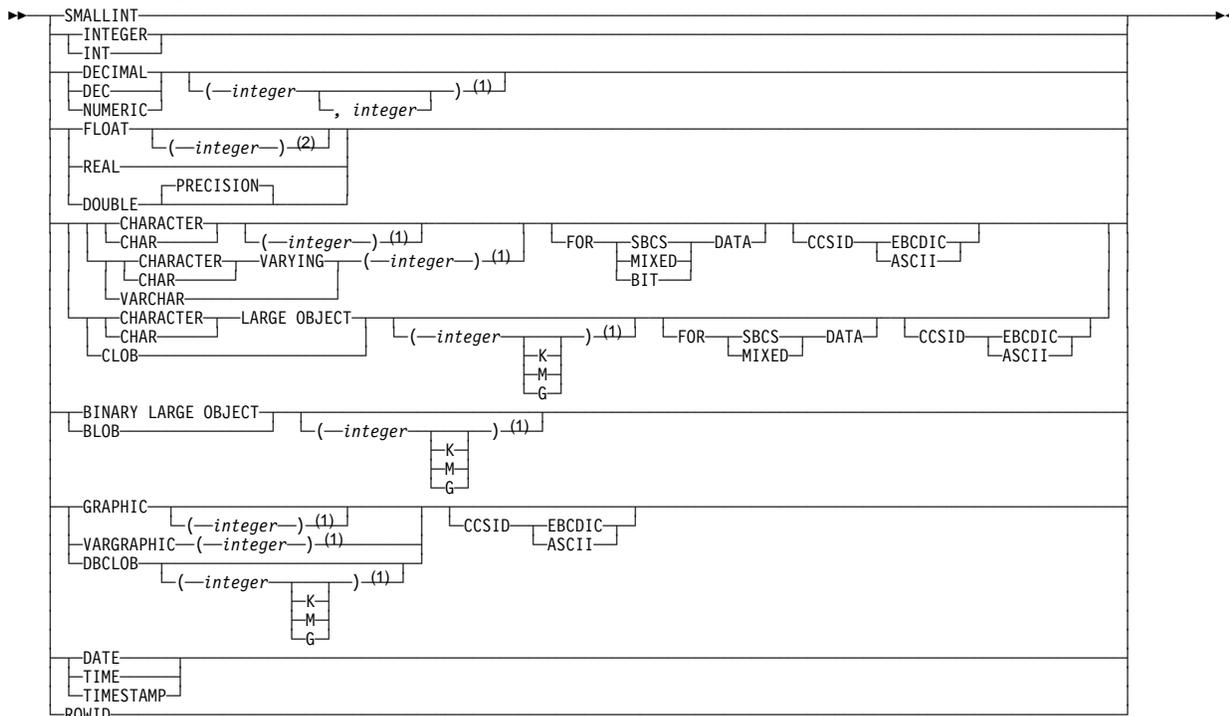
Note:

¹ AS LOCATOR can be specified only for a LOB data type or a distinct type that is based on a LOB data type.

data-type

►► *built-in-data-type* | *distinct-type-name* ►►

built-in-data-type



Notes:

- 1 The values that are specified for length, precision, or scale attributes must match the values that were specified when the function was created. Coding specific values is optional. Empty parentheses, (), can be used instead to indicate that DB2 ignores the attributes when determining whether data types match.
- 2 The value that is specified does not have to match the value that was specified when the function was created because matching is based on data type (REAL or DOUBLE). 1<=integer<= 21 indicates REAL and 22<=integer<=53 indicates DOUBLE. Coding a specific value is optional. Empty parentheses cannot be used.

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.

DISTINCT 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.

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 DISTINCT TYPE statement. The comment is placed in the REMARKS column of the SYSIBM.SYSROUTINES catalog table for the row that describes the function.

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 particular function, and is valid only if there is exactly one function with *function-name*.

FUNCTION *function-name (parameter-type,...)*

Provides the function signature, which uniquely identifies the function.

function-name

Identifies the name of the function.

(parameter-type,...)

Identifies the parameters of the function.

The data types of the parameters must match the data types that were specified on the CREATE FUNCTION statement in the corresponding position. The number of data types and the logical concatenation of the data types are used to identify the specific function.

For data types that have a length, precision, or scale attribute, you can specify a value or use a set of empty parentheses:

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match.

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 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. For example:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. For a complete list of the default lengths of data types, see “CREATE TABLE” on page 570.

For data types with a subtype or encoding scheme attribute, specifying the FOR DATA clause or 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.

SPECIFIC FUNCTION *specific-name*

Identifies the particular function using the specific name either specified or defaulted to when the function was created.

PROCEDURE *procedure-name*

Identifies the stored procedure to which the comment applies. *procedure-name* must identify a stored procedure that has been defined with the CREATE PROCEDURE statement at the current server. The comment is placed in the REMARKS column of the SYSIBM.SYSROUTINES catalog table for the row that describes the stored procedure.

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.

IS *string-constant*

Introduces the comment that you want to make. *string-constant* can be any SQL character string constant of up to 254 characters.

Examples

Example 1: Enter a comment on table DSN8610.EMP.

```
COMMENT ON TABLE DSN8610.EMP
  IS 'REFLECTS 1ST QTR 81 REORG';
```

Example 2: Enter a comment on view DSN8610.VDEPT.

```
COMMENT ON TABLE DSN8610.VDEPT
  IS 'VIEW OF TABLE DSN8610.DEPT';
```

Example 3: Enter a comment on the DEPTNO column of table DSN8610.DEPT.

```
COMMENT ON COLUMN DSN8610.DEPT.DEPTNO
  IS 'DEPARTMENT ID - UNIQUE';
```

Example 4: Enter comments on the two columns in table DSN8610.DEPT.

```
COMMENT ON DSN8610.DEPT
  (MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
   ADMRDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT');
```

Example 5: Assume that you are SMITH and that you created the distinct type DOCUMENT in schema SMITH. Enter comments on DOCUMENT.

```
COMMENT ON DISTINCT 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 stored procedure OSMOSIS is in schema BIOLOGY. Enter comments on 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';
```

COMMIT

COMMIT

The COMMIT statement ends a unit of recovery and commits the relational database changes that were made in that unit of recovery. If relational databases are the only recoverable resources used by the application process, COMMIT also ends the unit of work.

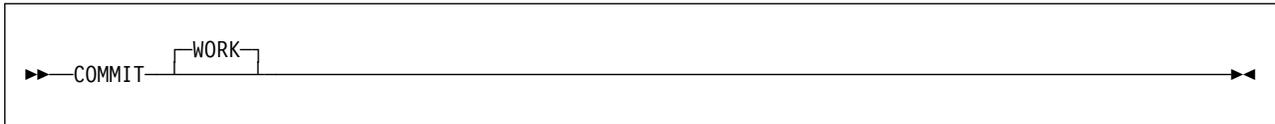
Invocation

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 unit of recovery in which the statement is executed is ended and a new unit of recovery is effectively started for the process. All changes made by ALTER, COMMENT ON, CREATE, DELETE, DROP, EXPLAIN, GRANT, INSERT, LABEL ON, RENAME, REVOKE, and UPDATE statements executed during the unit of recovery are committed, and all savepoints that were set within the unit of recovery are released. SQL connections are ended when any of the following 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, INSERT, UPDATE, and DELETE 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 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:

- 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 Section 5 (Volume 2) of *DB2 Administration 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.
```

Notes

The SQL 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.

In all DB2 environments, the normal termination of a process is an implicit commit operation.

Example

Commit all DB2 database changes made since the unit of recovery was started.

```
COMMIT WORK;
```

CONNECT

The CONNECT statement connects the application process to a designated server. This server is then the *current server* for the process. “When an application process has a current server” on page 447 describes what happens when the process has a current server.

CONNECT (Type 1) and CONNECT (Type 2) differences

There are two types of CONNECT statements with the same syntax but different semantics, as summarized below. Both types of the CONNECT statement are used for DRDA access, however the level of function available for each type is different. For a description of an individual type of CONNECT, see:

“CONNECT (Type 1)” on page 449

“CONNECT (Type 2)” on page 454

The following table summarizes the differences between CONNECT (Type 1) and CONNECT (Type 2) rules:

Table 31 (Page 1 of 2). CONNECT (Type 1) and CONNECT (Type 2) differences

Type 1 rules	Type 2 rules
CONNECT statements can be executed only when the application process is in the connectable state. Only one CONNECT statement can be executed within the same unit of work.	More than one CONNECT statement can be executed within the same unit of work. There are no rules about the connectable state.
If a CONNECT statement fails because the application process is not in the connectable state, the SQL connection status of the application process is unchanged.	If a CONNECT statement fails, the current SQL connection is unchanged and any subsequent SQL statements are executed by that server, unless the failure prevents the execution of SQL statements by that server.
If a CONNECT statement fails for any other reason, the application process is placed in the unconnected state.	
CONNECT ends any existing connections of the application process. Accordingly, CONNECT also closes any open cursors of the application process. (The only cursors that can possibly be open when CONNECT is successfully executed are those defined with the WITH HOLD option.)	CONNECT does not end connections and does not close cursors.

Table 31 (Page 2 of 2). CONNECT (Type 1) and CONNECT (Type 2) differences

Type 1 rules	Type 2 rules
A CONNECT to the current application server is executed like any other CONNECT (Type 1) statement.	<p>If the SQLRULES(STD) bind option is in effect, a CONNECT to an existing SQL connection of the application process is an error. Thus, a CONNECT to the current application server is an error. For example, an error occurs if the first CONNECT is a CONNECT TO x where x is the local DB2.</p> <p>If the SQLRULES(DB2) bind option is in effect, a CONNECT to an existing SQL connection is not an error. Thus, if x is an existing SQL connection of the application process, CONNECT TO x makes x its current connection. If x is already the current connection, CONNECT TO x has no effect on the state of any connections.</p>

Determining the CONNECT rules that apply: The following table explains how to determine the CONNECT rules that apply:

Table 32. Determining the CONNECT rules that apply

If the precompiler option...	is...	then the rules for...
CONNECT(1)	specified	CONNECT (Type 1) apply
CONNECT(2)	specified	CONNECT (Type 2) apply
CONNECT	omitted	CONNECT (Type 2) implicitly apply.

The CONNECT rules that apply to an application process are determined by the first CONNECT statement that is executed (successfully or unsuccessfully) by that application process:

- If it is a CONNECT (Type 1), then CONNECT (Type 1) rules apply and CONNECT (Type 2) statements are invalid.
- If it is a CONNECT (Type 2), then CONNECT (Type 2) rules apply and CONNECT (Type 1) statements are invalid.

Programs containing CONNECT statements that are precompiled with different CONNECT precompiler options cannot execute as part of the same application process. An error will occur when an attempt is made to execute the invalid CONNECT statement.

When an application process has a current server

The *current server* is the DBMS to which an application is actively connected. The following rules apply when an application process has a current server:

- Static SQL statements executed by the application process are taken from a package which was bound at that server. However, this does not apply to SQL statements such as CONNECT and RELEASE which are not represented in packages. Furthermore, if the current server is the local DB2 subsystem, SQL statements can also be taken from a DBRM that has been bound with the application plan. This is the case if the CURRENT PACKAGESET special

register is blank and the name of the application program executing the SQL statement is the same as the name of a DBRM.

- The package from which SQL statements are taken is determined by the name of the application program executing the SQL statement, the package list of the application plan, and CURRENT PACKAGESET.

The last part of the package name is the same as the name of the application program, unless a member name is specified during the bind process along with the DBRMLIB DD statement. The qualifier of the package name (the collection ID) can be determined by the package list or by the CURRENT PACKAGESET special register. For more information, see “SET CURRENT PACKAGESET” on page 817.

- Dynamic and static SQL statements that refer to objects at the server are executed at the server. Statements that refer to objects at yet another DB2 (which is possible only if the server is a DB2 subsystem) are executed at that DB2 rather than at the server.

Establishing a different server

The initial server of an application process is the local DB2 subsystem. A different server can be established by the explicit or implicit execution of a CONNECT statement.

The CURRENTSERVER bind option can affect which CONNECT rule is in effect. When an application process executes an SQL statement other than COMMIT, CONNECT TO, CONNECT RESET, SET CONNECTION, or ROLLBACK, a CONNECT (Type 1) statement is implicitly executed if both of the following apply:

- The CURRENTSERVER bind option was specified when the application plan was bound or rebound and the identified server is not the local DB2.
- An implicit or explicit CONNECT statement has not been executed by the application process.

For example, if CURRENTSERVER x was specified and the first SQL statement executed by the application process is an OPEN statement, a CONNECT TO x (Type 1) is executed before the OPEN statement is executed. If the implicit CONNECT fails, the application process is in the unconnected state. Regardless of whether the implied CONNECT is successful, the application process cannot execute a CONNECT (Type 2) statement because CONNECT (Type 1) rules are in effect.

In new distributed applications, use CONNECT (Type 2) and do not use the CURRENTSERVER bind option.

CONNECT (Type 1)

The CONNECT (Type 1) statement connects the application process to a designated server. This server is then the current server for the process. The CONNECT (Type 1) statement is used for DRDA access using the restricted level of function available in DB2 Version 2 Release 3. Differences between the two types of statements are described in “CONNECT (Type 1) and CONNECT (Type 2) differences” on page 446.

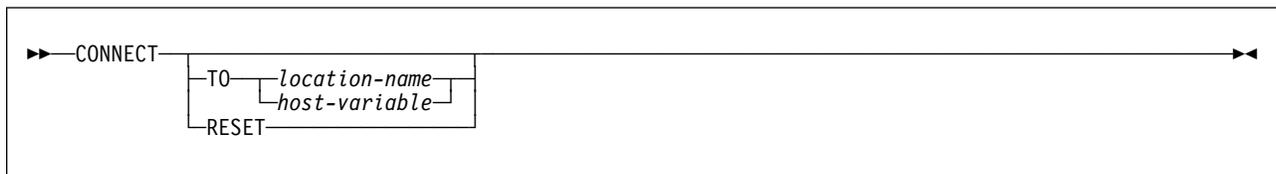
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The primary authorization ID of the process must be authorized to connect to the identified server. That server performs the authorization check and determines the specific authorization required. See Section 3 (Volume 1) of *DB2 Administration Guide* for further information.

Syntax



Description

TO *location-name* or *host-variable*

Identifies the server 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 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 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.

When the CONNECT statement is executed:

- The location name must identify a server known to the local DB2 subsystem. Hence, it must either 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 be in a *connectable state*. (Connection states are explained in Connection states on page 451.)

CONNECT (Type 1)

If execution of the CONNECT statement is successful:

- The application process is connected to the identified server.
- The existing connections of the application process are ended. (The existing connections include the previous SQL connection, if any, and all DB2 private connections, if any.) When a connection is ended, all resources acquired by the application process through the connection and all resources used to create and maintain the connection are deallocated. Thus, all cursors are closed, all prepared statements are destroyed, and so on.
- The location name is placed in the CURRENT SERVER special register.
- Information about the server is placed in the SQLERRP field of the SQLCA. If the application server is an IBM relational database product, the information has the form *pppvrrm*, where:
 - *ppp* is:
 - ARI for DB2 Server for VSE & VM
 - DSN for DB2 for MVS
 - QSQ for OS/400®
 - SQL for all other DB2 products
 - *vv* is a two-digit version identifier such as '06'.
 - *rr* is a two-digit release identifier such as '01'.
 - *m* is a one-digit modification level such as '0'.

#

For example, if the server is Version 6 of DB2 for OS/390 with the latest maintenance, the value of SQLERRP is 'DSN06011'.

If execution of the CONNECT statement is unsuccessful, the SQLERRP field of the SQLCA is set to the name of the DB2 application 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.

CONNECT RESET

CONNECT RESET is equivalent to CONNECT TO x where x is the location name of the local DB2 subsystem.

CONNECT with no operand

This form of the CONNECT statement returns information about the current server. The information is returned in the SQLERRP field of the SQLCA as described above. This form of CONNECT:

- Does not require the application process to be in the connectable state
- Does not change the connection state
- Does not close cursors
- Returns blanks if the application process is in the unconnected state

Notes

Connection states: In the following description of the connection states, CONNECT means CONNECT TO or CONNECT RESET, not the form of CONNECT with no operand. At any time, an application process is in one of four states:

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

The following diagram shows the state transitions:

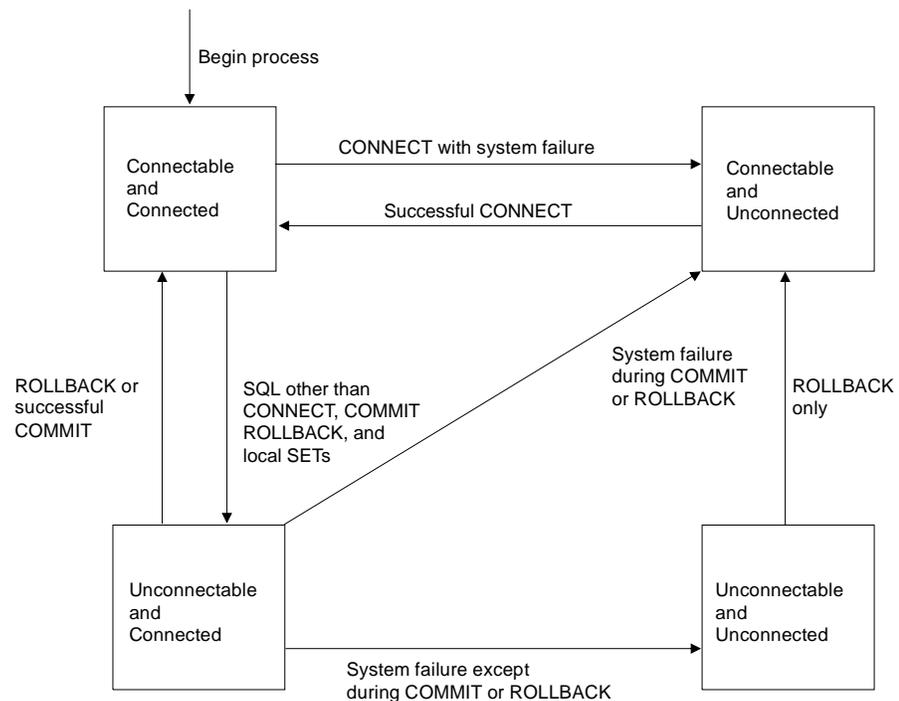


Figure 7. Connect state transitions

In the **connectable and connected state**, an application process is connected to a server and can execute CONNECT statements. This is the initial state. The process also enters this state when:

- It executes a rollback operation or successful commit from the unconnectable and connected state.
- It executes a successful CONNECT from the connectable and unconnected state.

In the **unconnectable and connected state**, an application process is connected to a server but cannot execute a CONNECT statement (SQLCODE -752). The process enters this state from the connectable and connected state when it executes any SQL statement other than CONNECT, COMMIT, ROLLBACK, or local SET (SET CURRENT PACKAGESET or SET *host-variable* = CURRENT PACKAGESET or CURRENT SERVER). A process cannot enter this state from the connectable and unconnected nor the unconnectable and unconnected states.

CONNECT (Type 1)

In the **unconnectable and unconnected state**, an application process is not connected to a server and cannot execute a CONNECT statement. The process enters this state from the unconnectable and connected state when the execution of an SQL statement other than COMMIT or ROLLBACK is unsuccessful because of a system failure that results in a rollback and deallocation of the conversation. The only SQL statement that can be successfully executed in this state is ROLLBACK. Any attempt to execute other SQL statements will result in an error (SQLCODE -918).

In the **connectable and unconnected state**, an application process is not connected to a server. The process enters this state when:

- The execution of CONNECT is unsuccessful for any reason other than the application process not being in the connectable state.
- A system failure occurs during the execution of a COMMIT or ROLLBACK statement from the unconnectable and connected state.
- A ROLLBACK statement is executed from the unconnectable and unconnected state.

The only SQL statements that can be successfully executed in this state are CONNECT, COMMIT, ROLLBACK, and local SET statements. Any attempt to execute other SQL statements will result in an error (SQLCODE -900). SET *host-variable* = CURRENT SERVER will set the host variable to blanks.

Additional rules: It is not an error to execute consecutive CONNECT statements because CONNECT itself does not remove the application process from the connectable state. It is an error to execute any SQL statement other than CONNECT, COMMIT, ROLLBACK, or local SET, and then execute CONNECT. To avoid the error, execute a commit or rollback operation before executing the CONNECT.

A CONNECT to the current server is treated like any other CONNECT. Such a CONNECT can cause the closing of cursors and the redundant deallocation and allocation of a conversation.

It may be the case that the SQL CONNECT statement returns, and indicates a successful execution when no physical connection yet exists. DB2 will delay the physical connection process, when possible, to economize on the number of messages sent. Therefore, errors in CONNECT statement processing may be reported following the next executable SQL statement, not immediately following the CONNECT statement.

When CONNECT is used to connect back to the local DB2, the CURRENT SQLID special register is not reinitialized.

SET CONNECTION and RELEASE do not change the state of the application process from connectable to unconnectable.

The SQLRULES bind option has no effect on CONNECT (Type 1) statements.

Examples

Example 1: Connect the application to a DBMS whose location identifier is in the character-string variable LOCNAME.

```
EXEC SQL CONNECT TO :LOCNAME;
```

Example 2: Use the CONNECT statement to obtain information about the current server. The information is then stored in the SQLERRP field of the SQLCA.

```
EXEC SQL CONNECT;
```

Example 3: An application has connected to a DB2 server that is not the local DBMS. During the connection, the application has opened a cursor and fetched rows from the cursor's result table. To connect to the local DBMS, the application executes the following statements:

```
EXEC SQL COMMIT WORK;  
EXEC SQL CONNECT RESET;
```

The commit operation is required because the OPEN statement for the cursor has caused the application to enter the unconnectable and connected state. If the cursor had been declared with WITH HOLD and had not been closed with a CLOSE statement, it would still be open after the execution of the COMMIT, but would be closed with the execution of the CONNECT.

CONNECT (Type 2)

The CONNECT (Type 2) statement connects the application process to a designated server. This server is then the current server for the process. Differences between the two types of statements are described in “CONNECT (Type 1) and CONNECT (Type 2) differences” on page 446. Refer to “Connection management for DRDA access and DB2 private protocol” on page 34 for more information about connection states.

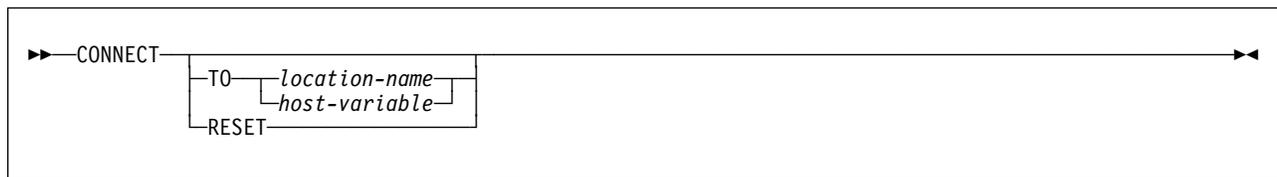
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The primary authorization ID of the process must be authorized to connect to the identified server. The authorization check is performed by the application server when the statement is executed, and the specific authorization required is determined by that server. See Section 3 (Volume 1) of *DB2 Administration Guide* for further information.

Syntax



Description

TO *location-name* or *host-variable*

Identifies the application server 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 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 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 not identify a DB2 private connection of the application process. If the SQLRULES(STD) bind option is in effect, S must not identify an existing SQL connection of the application process.

S must identify an application server known to the local DB2 subsystem. Hence, S must be the location name of the local DB2 subsystem or it must appear in the LOCATION column of the SYSIBM.LOCATIONS table.

If the CONNECT statement is successful:

- S becomes the current connection of the application process in one of the following ways:
 - If S is not an existing SQL connection of the application process, an SQL connection to application server S is created and placed in the current and held states. The previously current SQL connection, if any, is placed in the dormant state.
 - If S is a dormant SQL connection of the application process and the SQLRULES(DB2) option is in effect, S is placed in the current state. The previously current SQL connection, if any, is placed in the dormant state.
 - If S is the current SQL connection of the application process and the SQLRULES(DB2) option is in effect, the states of S and all other connections of the application process are unchanged.
- S is placed in the CURRENT SERVER special register.
- Information about application server S is placed in the SQLERRP field of the SQLCA. If the application server is an IBM relational database product, the information has the form *pppvrrm*, where:
 - *ppp* is:
 - ARI for DB2 Server for VSE & VM
 - DSN for DB2 for OS/390
 - QSQ for OS/400
 - SQL for all other DB2 products
 - *vv* is a two-digit version identifier such as '06'.
 - *rr* is a two-digit release identifier such as '01'.
 - *m* is a one-digit modification level such as '0'.

#

For example, if the server is Version 6 of DB2 for OS/390 with the latest maintenance, the value of SQLERRP is 'DSN06011'.

If the CONNECT statement is unsuccessful, the connection state of the application process and the states of its SQL connections are unchanged.

CONNECT 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.

CONNECT with no operand

This form of the CONNECT statement returns information about the current server and has no effect on connection states. The information is returned in

CONNECT (Type 2)

the SQLERRP field of the SQLCA as described above. SQLERRP is set to blanks if the application process is in the unconnected state.

Notes

When CONNECT is used to connect back to the local DB2, the CURRENT SQLID special register is not reinitialized.

Example

Execute SQL statements at TOROLAB1 and TOROLAB2. The first CONNECT statement creates the TOROLAB1 connection. The second CONNECT statement creates the TOROLAB2 connection and places the TOROLAB1 connection in the dormant state.

```
EXEC SQL CONNECT TO TOROLAB1;
```

```
(execute statements referencing objects at TOROLAB1)
```

```
EXEC SQL CONNECT TO TOROLAB2;
```

```
(execute statements referencing objects at TOROLAB2)
```

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

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that held by the authorization ID of the owner of the plan or package. If 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.

If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. If 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, or
- The qualifier must be the same as one of the authorization IDs of the process and the privileges that are held by that authorization ID must include the CREATEALIAS privilege. This is an exception to the rule that the privilege set is the privileges that are held by the SQL authorization ID of the process.

Syntax

```

▶▶ CREATE ALIAS alias-name FOR table-name
                                view-name
◀◀

```

Description

alias-name

Names the alias. The name must not identify a table, view, alias, or synonym that exists at the current server.

If qualified, the name can be a two-part or three-part name. If a three-part name is used, the first part must match the value of the field DB2 LOCATION NAME on installation panel DSNTIPR at the current server. (If the current server is not the local DB2, this name is not necessarily the name in the CURRENT SERVER special register.) Whether the name is two-part or three-part, the authorization ID that qualifies the name is the owner of the alias.

CREATE ALIAS

If the alias name is unqualified and the statement is embedded in an application program, the owner of the alias is the authorization ID that serves as the implicit qualifier for unqualified object names. This is the authorization ID in the QUALIFIER operand when the plan or package was created or last rebound. If QUALIFIER was not used, the owner of the alias is the owner of the package or plan.

If the alias name is unqualified and the statement is dynamically prepared, the SQL authorization ID is the owner of the alias.

The owner has the privilege to drop the alias.

FOR *table-name* or *view-name*

Identifies the table or view for which the alias is defined. If a table is identified,
it must not be an auxiliary table or a declared temporary table. The table or
view need not exist at the time the alias is defined. If it does exist, it can be at
the current server or at another server. The name must not be the same as the
alias name and must not identify an alias that exists at the current server.

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.

A warning occurs if an alias is defined for a table or view that is local to the current server but does not exist.

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

Do not use this statement if the value of special register CURRENT RULES is 'STD' when the statement is executed. When the register's value 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 chooses the names and characteristics of these objects. For more information about the names and the characteristics, see *Creating a table with LOB columns* on page 591.

This statement can be embedded in an application program or issued interactively if the value of special register CURRENT RULES is 'DB2' 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.

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

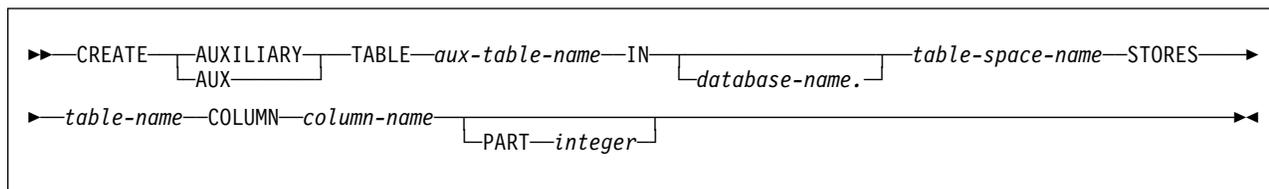
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package. If the specified table name includes a qualifier that is not the same as this authorization ID, the privilege set must include SYSADM or SYSCTRL authority, DBADM authority for the database, or DBCTRL authority for the database.

If 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 specified table name includes a qualifier that is not the same as this authorization ID, the following rules apply:

1. If the privilege set includes SYSADM or SYSCTRL authority, DBADM authority for the database, or DBCTRL authority for the database, any qualifier is valid.
2. If the privilege set does not include any of the authorities listed in item 1 above, the 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 all²⁸ privileges needed to create the table.

²⁸ Exception: The CREATETAB privilege is checked on the SQL authorization ID of the process.

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 must not identify a table, view, alias, or synonym that exists at the current server.

If 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.) Whether the name is two-part or three-part, the authorization ID that qualifies the name is the table's owner.

If the table name is unqualified and the statement is embedded in a program, the owner of the table is the authorization ID that serves as the implicit qualifier for unqualified object names. This is the authorization ID in the QUALIFIER operand when the plan or package was created or last rebound. If QUALIFIER was not used, the owner of the table is the owner of the package or plan.

If the table name is unqualified and the statement is dynamically prepared, the SQL authorization ID is the owner of the table.

IN *database-name.table-space-name* **or** **IN** *table-space-name*

Identifies the table space in which the auxiliary table is created. The name must identify an empty LOB table space that currently exists at the current server.

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 database DSNDB04.

STORES *table-name* **COLUMN** *column-name*

Identifies the base table and the column of that table that is to be stored in the auxiliary table. If the base table is nonpartitioned, an auxiliary table must not 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 or EBCDIC depending on the value of the CCSID clause when the base table was created.

The auxiliary table can store a BLOB, CLOB, or DBCLOB value that is greater than 1 gigabyte in length only if the LOB table space for the auxiliary table was defined with LOG NO.

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

Determining the number of auxiliary tables to create: The number of auxiliary tables to create depends on the number of LOB columns in the base table and whether the base table is partitioned. If the base table is nonpartitioned, you need one LOB table space and one auxiliary table for each LOB column in the base table. If the base table is partitioned, you need one LOB table space and one auxiliary table for each partition for each LOB column. For example if the base table has four partitions and two LOB columns, you need to create a total of eight auxiliary tables in eight different LOB table spaces.

Example

Assume that a column named EMP_PHOTO with a data type of BLOB(110K) has been added to sample employee table DSN8610.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 DSN8D61A.PHOTOLTS.

```
CREATE AUX TABLE EMP_PHOTO_ATAB
  IN DSN8D61A.PHOTOLTS
  STORES DSN8610.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

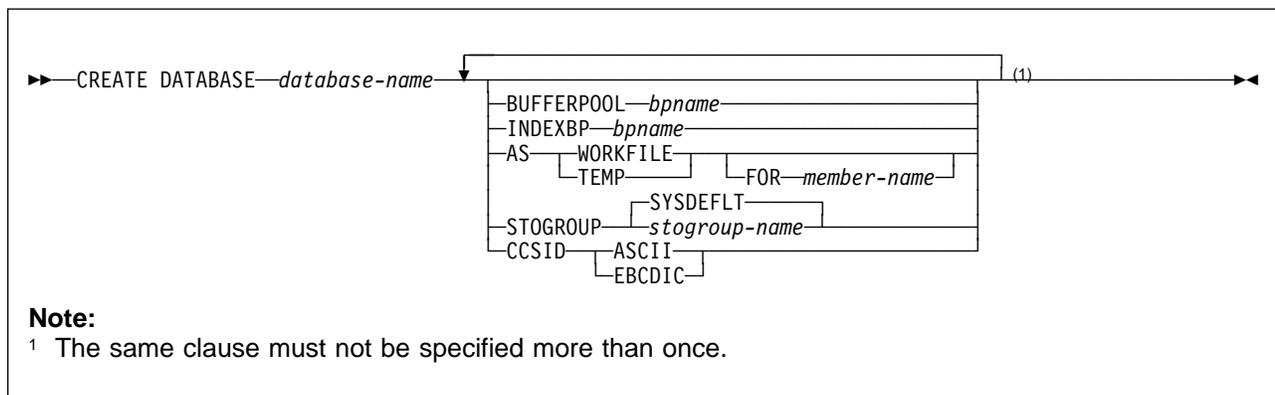
The privilege set that is defined below must include at least one of the following:

- 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 authorization ID of 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.

See “Notes” on page 464 for the authorization effect of a successful CREATE DATABASE statement.

Syntax



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. 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 50 for more details about *bpname*.

If you omit the BUFFERPOOL clause, the buffer pool specified for user data on installation panel DSNTIP1 is used. The default value for the user data field on that panel is BP0.

INDEXBP *bpname*

Specifies the default buffer pool name to be used for the indexes created within the database. The name must identify a 4KB buffer pool. See “Naming conventions” on page 50 for more details about *bpname*. If the database is a work file database, INDEXBP cannot be specified.

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 or **AS TEMP**

Indicates that this is a work file database or a database for declared temporary tables (a TEMP database).

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 member.

AS TEMP

Specifies the database is for declared temporary tables only. AS TEMP must be specified to create a database that will be used for declared temporary tables; otherwise, the database will not be used for declared temporary tables. Only one TEMP database can be created for each DB2 subsystem or data sharing member. A TEMP database cannot be shared between DB2 subsystems or data sharing members.

PUBLIC implicitly receives the CREATETAB privilege (without GRANT authority) to define a declared temporary table in the TEMP database. This implicit privilege is not recorded in the DB2 catalog and cannot be revoked.

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.

The CCSID clause is not supported for a work file database or a TEMP database. A TEMP database can contain a mixture of encoding schemes. If you specify AS WORKFILE or AS TEMP, do not use the CCSID clause.

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.

CREATE DATABASE

EBCDIC Specifies that the data must be encoded using the EBCDIC CCSIDs of the server.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed or graphic 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 or AS TEMP 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 DSN8D61P. Specify DSN8G610 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 DSN8D61P  
| STOGROUP DSN8G610  
| 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 DISTINCT TYPE

The CREATE DISTINCT TYPE statement defines a distinct type, which is a data type that a user defines. A distinct type must be sourced 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 for the schema or all schemas
- 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 authorization ID of 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 distinct type 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 CREATEIN privilege on the schema.

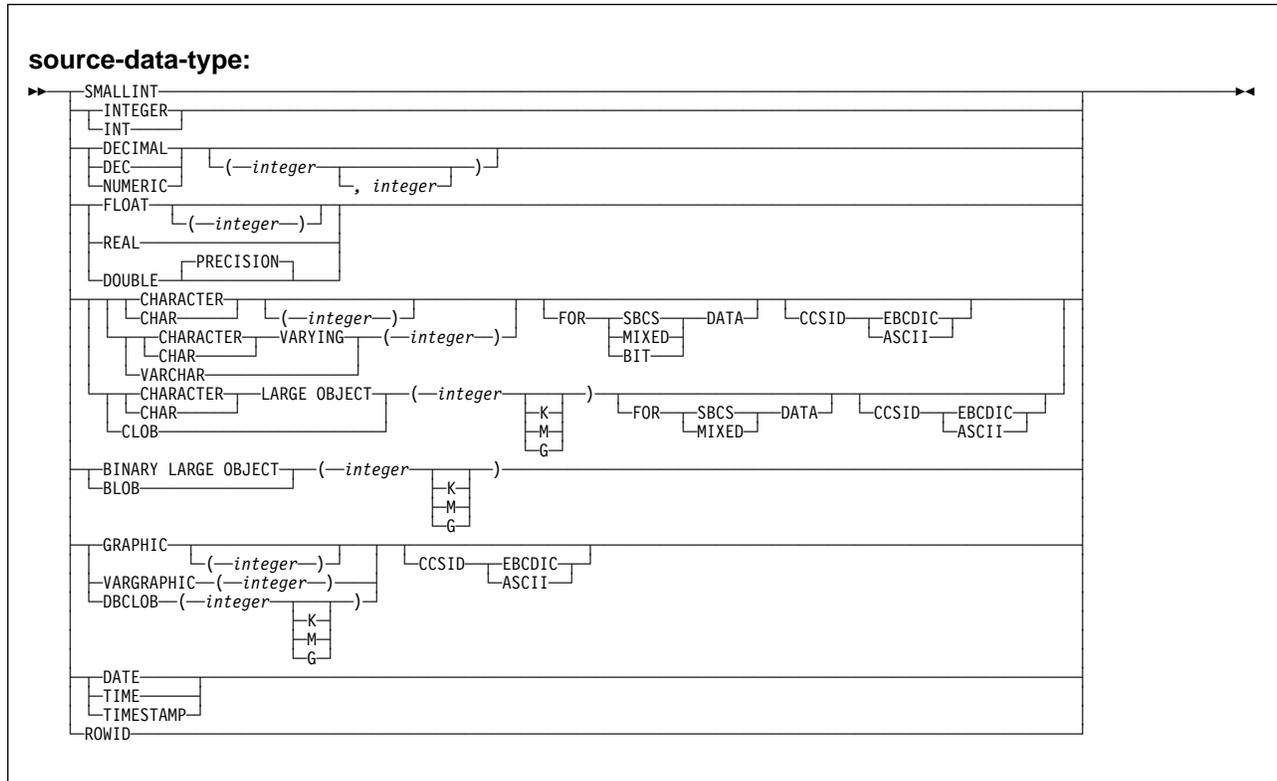
Syntax

```
►► CREATE DISTINCT TYPE distinct-type-name AS source-data-type WITH COMPARISONS (1) ◀◀
```

Note:

- ¹ Do not specify WITH COMPARISONS for *source-data-types* that are BLOBs, CLOBs, or DBCLOBs. The WITH COMPARISONS clause is required for all other *source-data-types*.

CREATE DISTINCT TYPE



Description

distinct-type-name

Names the distinct type. The name is implicitly or explicitly qualified by a schema name. The name, together with the implicit or explicit schema name, must not identify a distinct type that exists at the current server.

- The unqualified form of *distinct-type-name* is a long SQL identifier.

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 unqualified name is implicitly qualified with a schema name according to the following rules:

If the CREATE DISTINCT TYPE statement is embedded in a program, the schema name is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not specified, the schema name is the owner of the plan or package.

If the CREATE DISTINCT TYPE statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

- The qualified form of *distinct-type-name* is a short SQL identifier (the schema name) followed by a period and a long SQL identifier.

A schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

The owner of the distinct type is determined by how the CREATE DISTINCT TYPE 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.

Although the information is not recorded in the catalog, the owner is given the USAGE privilege on the distinct type. The owner is also given the EXECUTE privilege with the GRANT option on each of the generated cast functions.

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. You can use any of the built-in data types that are allowed for the CREATE TABLE statement except LONG VARCHAR or LONG VARGRAPHIC. Use VARCHAR or VARGRAPHIC with an explicit length instead.

If you do not specify a specific value for the data types that have length, precision, or scale attributes (CHAR, GRAPHIC, DECIMAL, NUMERIC, FLOAT), the defaults are as follows:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

For more information on built-in data types, see “built-in-data-type” on page built-in-data-type on page 575.

If the distinct type is sourced on a character 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. 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.

If the distinct type is sourced on a string data type, the CCSID clause indicates whether the encoding scheme of the data is ASCII or EBCDIC. If you do not

CREATE DISTINCT TYPE

specify CCSID ASCII or CCSID EBCDIC, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

WITH COMPARISONS

Specifies that system-generated comparison operators are to be created for comparing two instances of the distinct type. Do not specify WITH COMPARISONS if the source data type is BLOB, CLOB, or DBCLOB; otherwise, a warning occurs and the comparison operators are not generated. You must specify WITH COMPARISONS for all other source data types.

DB2 implicitly creates comparison functions for the following comparison operators for use with the distinct type:

BETWEEN	IS NULL	<	¬<
NOT BETWEEN	IS NOT NULL	>	¬>
IN	=	<=	>=
NOT IN	¬=	<>	

The name of the function is the same as the comparison operator. You cannot invoke the comparison functions using function notation syntax, for example, '<' (C1,C2). Instead, use the syntax C1 < C2.

You must not specify the comparison operations for distinct types that are sourced on a CLOB, BLOB, or DBCLOB, or that have a length greater than 255 bytes. (Distinct types that are sourced on a VARCHAR or VARGRAPHIC can have a length greater than 255 bytes).

Notes

Source data types with DBCS or mixed data: If you specify a GRAPHIC source type and the ASCII encoding scheme, or a string source type with a mixed data subtype (FOR MIXED DATA), the value of field FOR MIXED DATA on installation panel DSNTIPF must be YES; otherwise, an error occurs. In addition, to create a distinct type on a GRAPHIC data type, the corresponding CCSID must be defined for the implicitly or explicitly specified encoding scheme.

Generated cast functions: The successful execution of the CREATE DISTINCT 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
- 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 DISTINCT 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 33 on page 470 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 DISTINCT TYPE CLAIRE.T_SHOESIZE AS VARCHAR(2) WITH COMPARISONS
```

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 DISTINCT TYPE CLAIRE.T_SHOESIZE AS CHAR(2) WITH COMPARISONS
```

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.

CREATE DISTINCT TYPE

For each built-in data type that can be the source data type for a distinct type, Table 33 on page 470 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 33 (Page 1 of 2). CAST functions on distinct types

Source type name	Function name	Parameter-type	Return-type
CHAR CHARACTER	<i>distinct</i>	CHAR (n)	<i>distinct</i>
	CHAR	<i>distinct</i>	CHAR (n)
	<i>distinct</i>	VARCHAR (n)	<i>distinct</i>
VARCHAR CHARACTER VARYING CHAR VARYING	<i>distinct</i>	VARCHAR (n)	<i>distinct</i>
	VARCHAR	<i>distinct</i>	VARCHAR (n)
CLOB	<i>distinct</i>	CLOB (n)	<i>distinct</i>
	CLOB	<i>distinct</i>	CLOB (n)
BLOB	<i>distinct</i>	BLOB (n)	<i>distinct</i>
	BLOB	<i>distinct</i>	BLOB (n)
GRAPHIC	<i>distinct</i>	GRAPHIC (n)	<i>distinct</i>
	GRAPHIC	<i>distinct</i>	GRAPHIC (n)
	<i>distinct</i>	VARGRAPHIC (n)	<i>distinct</i>
VARGRAPHIC	<i>distinct</i>	VARGRAPHIC (n)	<i>distinct</i>
	VARGRAPHIC	<i>distinct</i>	VARGRAPHIC (n)
DBCLOB	<i>distinct</i>	DBCLOB (n)	<i>distinct</i>
	DBCLOB	<i>distinct</i>	DBCLOB (n)
SMALLINT	<i>distinct</i>	SMALLINT	<i>distinct</i>
	<i>distinct</i>	INTEGER	<i>distinct</i>
	SMALLINT	<i>distinct</i>	SMALLINT
INTEGER	<i>distinct</i>	INTEGER	<i>distinct</i>
	INTEGER	<i>distinct</i>	INTEGER
DECIMAL	<i>distinct</i>	DECIMAL (p,s)	<i>distinct</i>
	DECIMAL	<i>distinct</i>	DECIMAL (p,s)
NUMERIC	<i>distinct</i>	DECIMAL (p,s)	<i>distinct</i>
	DECIMAL	<i>distinct</i>	DECIMAL (p,s)
REAL	<i>distinct</i>	REAL	<i>distinct</i>
	<i>distinct</i>	DOUBLE	<i>distinct</i>
	REAL	<i>distinct</i>	REAL
FLOAT(n) where n<=24	<i>distinct</i>	REAL	<i>distinct</i>
	<i>distinct</i>	DOUBLE	<i>distinct</i>
	REAL	<i>distinct</i>	REAL
FLOAT(n) where n>24	<i>distinct</i>	DOUBLE	<i>distinct</i>
	DOUBLE	<i>distinct</i>	DOUBLE
FLOAT	<i>distinct</i>	DOUBLE	<i>distinct</i>
	DOUBLE	<i>distinct</i>	DOUBLE

Table 33 (Page 2 of 2). CAST functions on distinct types

Source type name	Function name	Parameter-type	Return-type
DOUBLE	<i>distinct</i>	DOUBLE	<i>distinct</i>
	DOUBLE	<i>distinct</i>	DOUBLE
DOUBLE PRECISION	<i>distinct</i>	DOUBLE	<i>distinct</i>
	DOUBLE	<i>distinct</i>	DOUBLE
DATE	<i>distinct</i>	DATE	<i>distinct</i>
	DATE	<i>distinct</i>	DATE
TIME	<i>distinct</i>	TIME	<i>distinct</i>
	TIME	<i>distinct</i>	TIME
TIMESTAMP	<i>distinct</i>	TIMESTAMP	<i>distinct</i>
	TIMESTAMP	<i>distinct</i>	TIMESTAMP
ROWID	<i>distinct</i>	ROWID	<i>distinct</i>
	ROWID	<i>distinct</i>	ROWID

Notes: In the table, *distinct* represents *distinct-type-name*.

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 508.

Examples

Example 1: Create a distinct type named SHOESIZE that is sourced on an INTEGER data type.

```
CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS;
```

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 sourced on a DOUBLE data type.

```
CREATE DISTINCT TYPE MILES AS DOUBLE WITH COMPARISONS;
```

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 FUNCTION

The CREATE FUNCTION statement registers a user-defined function with an application server. You can register three 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 an application server along with various attributes of the function. See “CREATE FUNCTION (external scalar)” on page 473.

- External Table

The function is written in a programming language and returns a complete table. The external executable is registered with an application server along with various attributes of the function. See “CREATE FUNCTION (external table)” on page 492.

- Sourced

The function is implemented by invoking another function (either built-in, external, or sourced) that is already registered with an application server. See “CREATE FUNCTION (sourced)” on page 508.

CREATE FUNCTION (external scalar)

This CREATE FUNCTION statement registers a user-defined external scalar function with an application 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 for the schema or all schemas
- 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 authorization ID of 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 function 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:

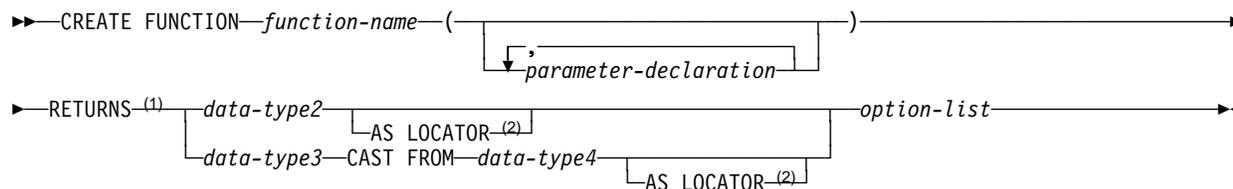
- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

Additional privileges are required if the function uses a table as a parameter, refers to a distinct type, or is to run in a MVS workload manager (WLM) 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.

CREATE FUNCTION (external scalar)

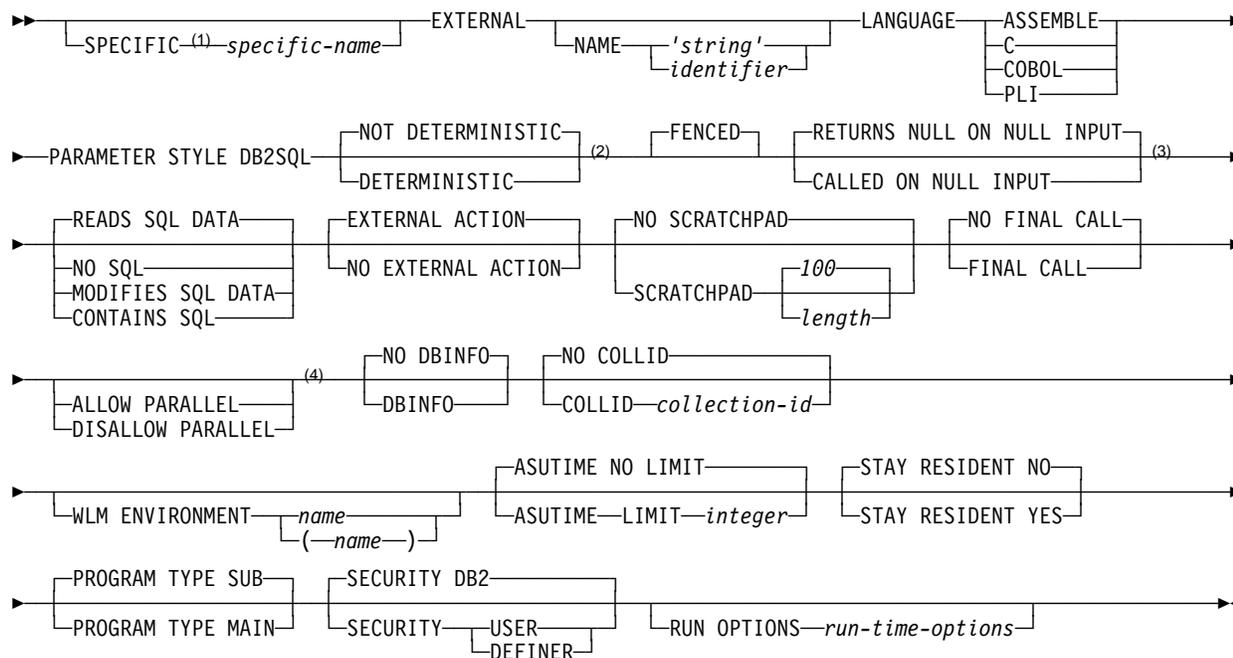
Syntax



Notes:

- 1 This clause and the clauses that follow in the *option-list* can be specified in any order.
- 2 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

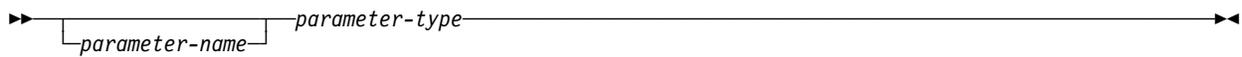
option-list:



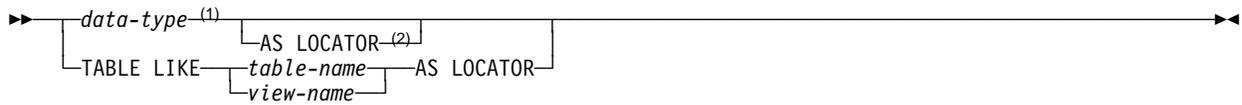
Notes:

- 1 This clause and the other clauses in the *option-list* can be specified in any order.
- 2 Synonyms for this clause include VARIANT for NOT DETERMINISTIC, and NOT VARIANT for DETERMINISTIC.
- 3 Synonyms for this clause include NOT NULL CALL for RETURNS NULL ON NULL INPUT, and NULL CALL for CALLED ON NULL INPUT.
- 4 If NOT DETERMINISTIC, EXTERNAL ACTION, SCRATCHPAD, or FINAL CALL is specified, DISALLOW PARALLEL is the default.

parameter-declaration:



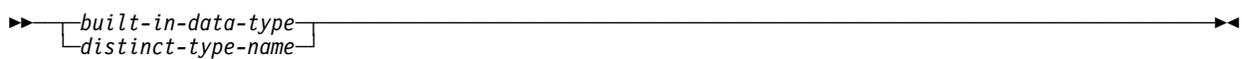
parameter-type:



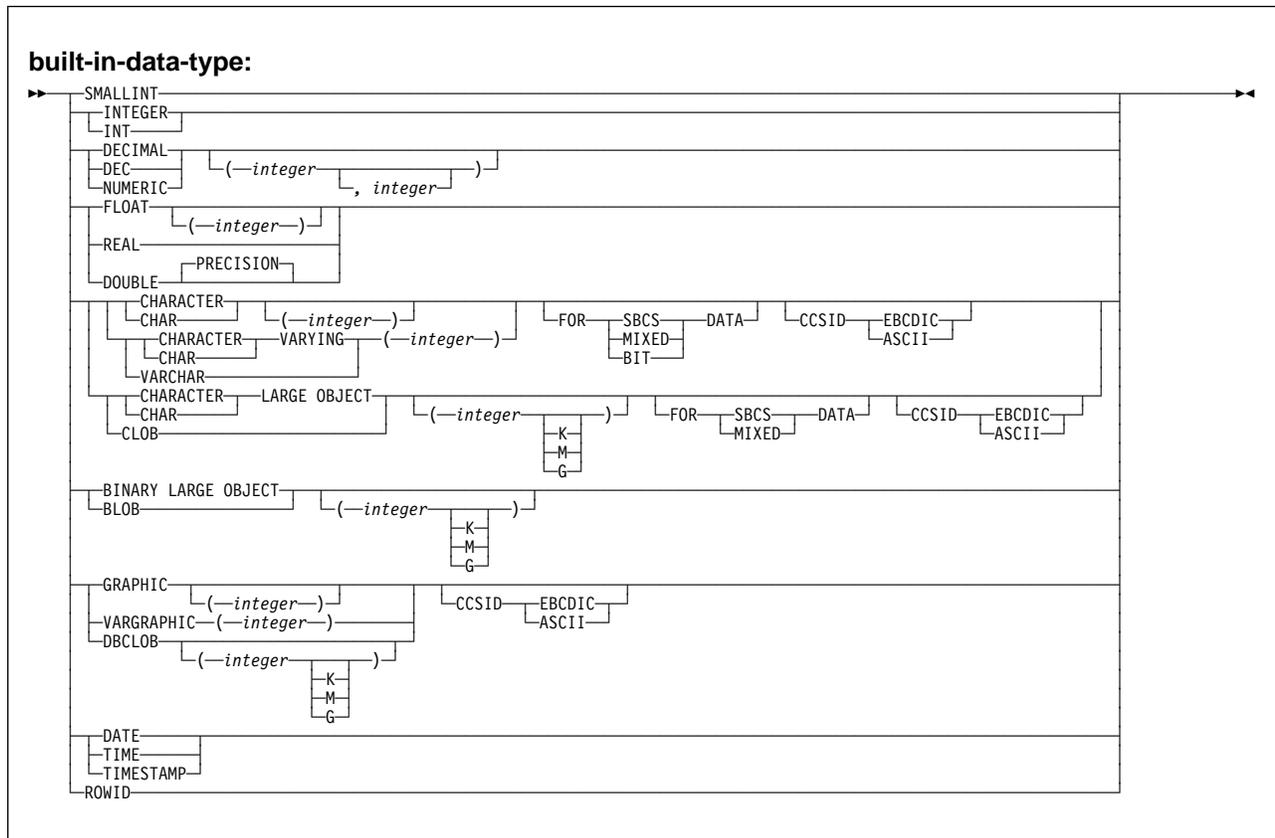
Notes:

- ¹ A LOB data type or distinct type based on a LOB data type must be no greater than 1M unless a locator is passed.
- ² AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

data-type:



CREATE FUNCTION (external scalar)



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 parameter²⁹ (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 a long SQL identifier.

The name must not be any of the following system-reserved keywords even if you specify them as delimited identifiers:

²⁹ If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

CREATE FUNCTION (external scalar)

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 unqualified function name 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 plan or package was created or last rebound. If QUALIFIER was not specified, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.
- The qualified form of *function-name* is a short SQL identifier (the schema name) followed by a period and a long SQL identifier.

The schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

The owner of the function is determined by how the CREATE FUNCTION 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 function.

(parameter-declaration,...)

Identifies the number of input parameters of the function, and specifies the data type of each parameter. All 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. 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 a long 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.

CREATE FUNCTION (external scalar)

built-in-data-type

The data type of the input parameter is a built-in data type. You can use the same built-in data types as for the CREATE TABLE statement except LONG VARCHAR or LONG VARGRAPHIC. Use VARCHAR or VARGRAPHIC with an explicit length instead.

If you do not specify a specific value for the data types that have length, precision, or scale attributes (CHAR, GRAPHIC, DECIMAL, NUMERIC, FLOAT), the defaults are as follows:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

For information on the data types, including the subtype of character data types (the FOR *subtype* DATA clause), see “built-in-data-type” on page built-in-data-type on page 575.

For parameters with a string data type, the CCSID clause indicates whether the encoding scheme of the parameter value is ASCII or EBCDIC. If you do not specify CCSID ASCII or CCSID EBCDIC, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

Although 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. 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 *name* specified after TABLE LIKE must identify a table or view that exists at the current server. The name must not identify a declared temporary table. 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 SQLID 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.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

RETURNS

Identifies the output of the function. Consider this clause in conjunction with the optional CAST FROM clause.

data-type2

Specifies the data type of the output.

The same considerations that apply to the data type of input parameter, as described under *data-type* on page 477, apply to the data type of the output of the function.

CREATE FUNCTION (external scalar)

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 or distinct type. (For information on casting data types, see “Casting between data types” on page 83.) The encoding scheme of the parameters, if they are string data types, must be the same.

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 a long SQL identifier. The qualified form is a short SQL identifier (the schema name) followed by a period and a long 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 ON, 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.

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 MVS load module that contains the user-written code that implements the logic of the function. The name can be a string constant that is no longer than 8 characters or a short identifier, The name must conform to the naming conventions for MVS load modules.

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++.

COBOL

The function is written in COBOL, including the object-oriented language extensions.

PLI

The function is written in PL/I.

PARAMETER STYLE DB2SQL

Specifies the linkage convention that the function program uses to receive input parameters from and pass return values to the invoking SQL statement.

DB2SQL indicates that parameters for indicator variables are associated with each input and return value to allow for null values. The parameters that are passed between the invoking SQL statement and the function include:

- The first *n* parameters are the input parameters that are specified for the function
- A parameter for the result of the function

CREATE FUNCTION (external scalar)

- 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

Zero to three additional parameters might also be passed:

- The scratchpad, if SCRATCHPAD is specified
- The call type, if NO FINAL CALL is specified
- The DBINFO structure, if DBINFO is specified

NOT DETERMINISTIC or **DETERMINISTIC**

Specifies whether the function returns the same results for identical input arguments.

NOT DETERMINISTIC

The function might not return the same result for identical input arguments. The function depends on some state values that affect the results. DB2 uses this information when processing a SELECT, UPDATE, DELETE, or INSERT statement to disable merging of views that refer to the 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 for identical input arguments. DB2 can use this information to optimize view processing for SELECT, UPDATE, DELETE, or INSERT statements. An example of a deterministic function is a function that calculates the square root of the input.

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 INPUT

The function is not called if any of the input arguments is null. The result is the null value.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments is null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

NO SQL, MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL

Indicates whether the function can execute any SQL statements and, if so, what type. See Table 56 on page 877 for a detailed list of the SQL statements that can be executed under each data access indication.

NO SQL

The function cannot execute any SQL statements.

MODIFIES SQL DATA

The function can execute any SQL statement except those statements that are not supported in any function. Do not specify MODIFIES SQL DATA when ALLOW PARALLEL is in effect.

READS SQL DATA

The function cannot execute SQL statements that modify data. SQL statements that are not supported in any function return a different error.

READS SQL DATA is the default.

CONTAINS SQL

The function cannot execute any SQL statements that read or modify data. SQL statements that are not supported in any function return a different error.

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

The function can take an action that changes the state of an object that DB2 does not manage.

Some functions with external actions can receive 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 in SELECT, UPDATE, DELETE or INSERT statements that refer to function.
- 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.

CREATE FUNCTION (external scalar)

NO EXTERNAL ACTION

The function does not take any action that changes the state of an object that DB2 does not manage. DB2 can use this information to optimize the processing of views for SELECT, UPDATE, DELETE or INSERT statements.

DB2 does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

NO SCRATCHPAD or SCRATCHPAD

Specifies whether DB2 provides 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

A scratchpad is not allocated and passed to the function. NO SCRATCHPAD is the default.

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'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.

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

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

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:

Normal call

SQL arguments are passed and the function is expected to return a result.

First call

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.

Final call

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

CREATE FUNCTION (external scalar)

function. Specify the `DISALLOW PARALLEL` clause for functions that have inappropriate actions when executed in parallel.

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

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 `NOT DETERMINISTIC`, `EXTERNAL ACTION`, `SCRATCHPAD`, and `FINAL CALL` for considerations when specifying `ALLOW PARALLEL`.

DISALLOW PARALLEL

Specifies that DB2 does not consider parallelism for the function.

The default is `DISALLOW PARALLEL`, if you specify one or more of the following clauses:

- `NOT DETERMINISTIC`
- `EXTERNAL ACTION`
- `FINAL CALL`
- `MODIFIES SQL DATA`
- `SCRATCHPAD`

Otherwise, `ALLOW PARALLEL` is the default.

NO DBINFO or DBINFO

Specifies whether specific information that DB2 knows is passed to the function when it is invoked.

NO DBINFO

No additional information is passed. `NO DBINFO` is the default.

DBINFO

An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run-time authorization ID, the schema name, the name of a table or column

that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

NO COLLID or **COLLID** *collection-id*

Identifies the package collection that is 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, the package collection is the value of the CURRENT PACKAGESET special register.

NO COLLID is the default.

COLLID *collection-id*

The name of the package collection that is used when the function is executed.

WLM ENVIRONMENT

Identifies the MVS workload manager (WLM) application environment in which the function is to run. The *name* of the WLM environment is a long 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 MVS 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 on page 490.

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.

CREATE FUNCTION (external scalar)

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 *OS/390 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 2GB. 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 remains 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 *run-time-options*

Specifies the Language Environment run-time options to be used for the function. You must specify *run-time-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any run-time options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment run-time options, see *OS/390 Language Environment for OS/390 & VM Programming Reference*.

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 81). 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

For portability of functions across platforms that are not DB2 for OS/390, 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 the parameters with a 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), and NUMERIC(4,2).

CREATE FUNCTION (external scalar)

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15)) ...  
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...
```

```
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...  
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Overriding a built-in function: Giving 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 MVS address space in which a function 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)
```

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 DB2SQL
  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 DB2SQL
  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 DB2SQL
  NO EXTERNAL ACTION
  SCRATCHPAD
  NO FINAL CALL;

```

CREATE FUNCTION (external table)

This CREATE FUNCTION statement registers a user-defined external table function with an application server.

A table function can be used in the FROM clause of a SELECT. It returns a table to the SELECT one row at a time.

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 for the schema or all schemas
- 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 authorization ID of the owner of the plan or package.

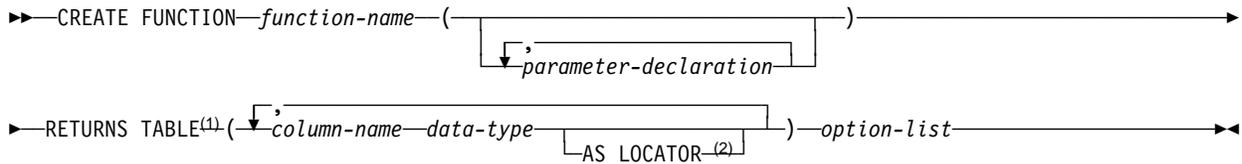
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. The specified function name can include a schema name (a qualifier). However, if the schema name is not the same as the SQL authorization ID, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the 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 MVS workload manager (WLM) 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.

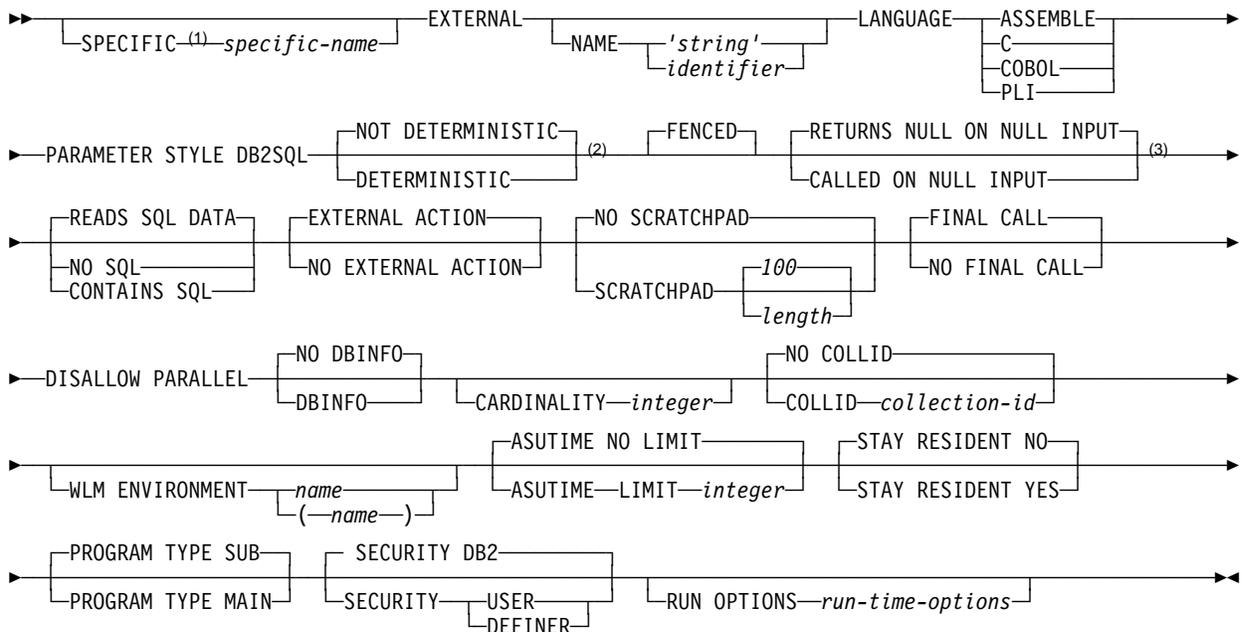
Syntax



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.

option-list:



Notes:

- 1 This clause and the other clauses in the *option-list* can be specified in any order.
- 2 Synonyms include VARIANT for NOT DETERMINISTIC, and NOT VARIANT for DETERMINISTIC.
- 3 Synonyms include NOT NULL CALL for RETURNS NULL ON NULL INPUT, and NULL CALL for CALLED ON NULL INPUT.

parameter-declaration:



CREATE FUNCTION (external table)

parameter-type:



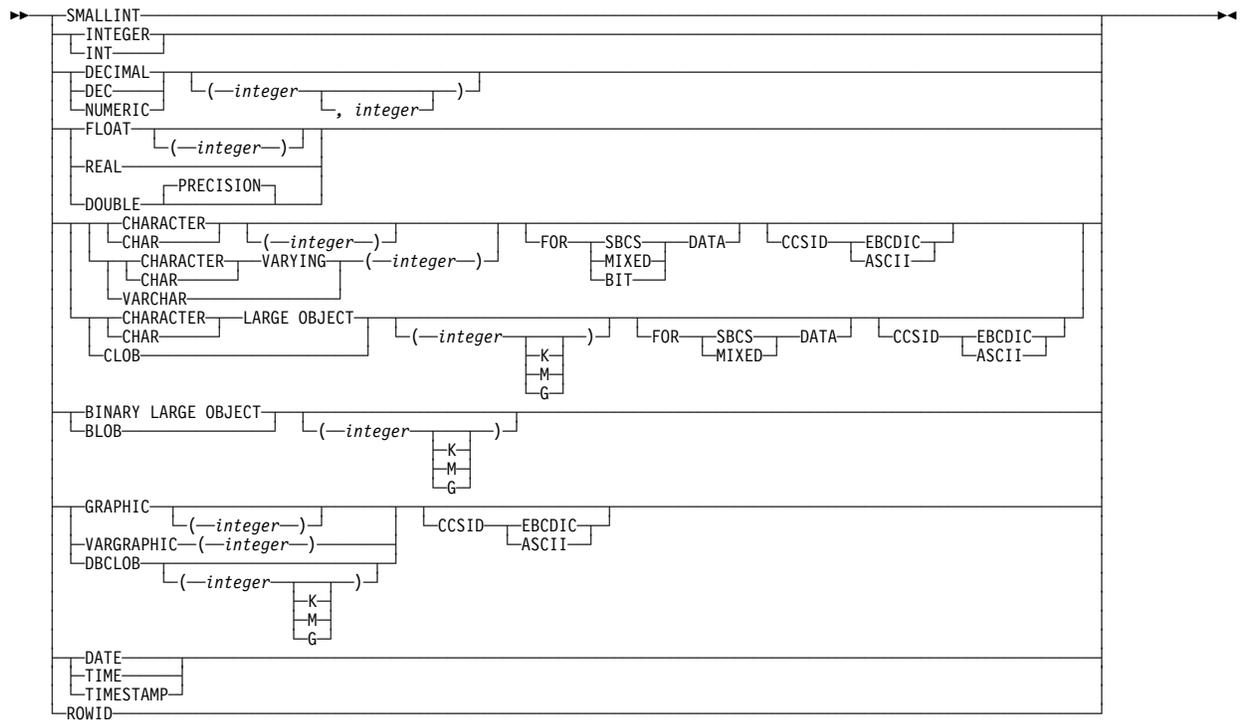
Notes:

- 1 A LOB data type or distinct type based on a LOB data type must be no greater than 1M unless a locator is passed.
- 2 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-data-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 parameter³⁰ (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 a long 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 unqualified function name 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 plan or package was created or last rebound. If QUALIFIER was not specified, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.
- The qualified form of *function-name* is a short SQL identifier (the schema name) followed by a period and a long SQL identifier.

The schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

The owner of the function is determined by how the CREATE FUNCTION 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 function.

³⁰ If the function has more than 30 parameters, only the first 30 parameters are used to determine whether the function is unique.

CREATE FUNCTION (external table)

(parameter-declaration,...)

Identifies the number of input parameters of the function, and specifies the data type of each parameter. All 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. 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 a long 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-data-type

The data type of the input parameter is a built-in data type. You can use the same built-in data types as for the CREATE TABLE statement except LONG VARCHAR or LONG VARGRAPHIC. Use VARCHAR or VARGRAPHIC with an explicit length instead.

If you do not specify a specific value for the data types that have length, precision, or scale attributes (CHAR, GRAPHIC, DECIMAL, NUMERIC, FLOAT), the defaults are as follows:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

For information on the data types, including the subtype of character data types (the FOR *subtype* DATA clause), see “built-in-data-type” on page built-in-data-type on page 575.

For parameters with a string data type, the CCSID clause indicates whether the encoding scheme of the parameter value is ASCII or EBCDIC. If you do not specify CCSID ASCII or CCSID EBCDIC, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

Although 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. 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 *name* specified after TABLE LIKE must identify a table or view that exists at the current server. The name must not identify a declared temporary table. 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 SQLID special register.

CREATE FUNCTION (external table)

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, encoding scheme, and CCSID of these columns must match exactly. The description of the table or view at the time the CREATE FUNCTION statement was executed is used.

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 a long identifier and must be unique within the RETURNS TABLE clause for the function.

data-type

Specifies the data type of the column. The data type can be any built-in data type, except LONG VARCHAR or LONG VARGRAPHIC. The data type can also be any distinct type.

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 a long SQL identifier. The qualified form is a short SQL identifier (the schema name) followed by a period and a long 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 ON, 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.

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 MVS load module that contains the user-written code that implements the logic of the function. The name can be a string constant that is no longer than 8 characters or a short identifier. The name must conform to the naming conventions for MVS load modules.

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++.

COBOL

The function is written in COBOL, including the object-oriented language extensions.

PLI

The function is written in PL/I.

PARAMETER STYLE DB2SQL

Specifies the linkage convention that the function program uses to receive input parameters from and pass return values to the invoking SQL statement.

DB2SQL indicates that parameters for indicator variables are associated with each input and return value to allow for null values. The parameters that are passed between the invoking SQL statement and the function include:

- The first *n* parameters are the input parameters that are specified for the function
- A parameter for the result of the function

CREATE FUNCTION (external table)

- 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

Zero to three additional parameters might also be passed:

- The scratchpad, if SCRATCHPAD is specified
- The call type, if NO FINAL CALL is specified
- 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 for identical input arguments.

NOT DETERMINISTIC

The function might not return the same result for identical input arguments. The function depends on some state values that affect the results. DB2 uses this information when processing a SELECT, UPDATE, DELETE, or INSERT statement to disable merging of views that refer to the 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 for identical input arguments. DB2 can use this information to optimize view processing for SELECT, UPDATE, DELETE, or INSERT statements. An example of a deterministic function is a function that calculates the square root of the input.

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 NULL INPUT is the default.

CALLED ON NULL INPUT

The function is called regardless of whether any of the input arguments is null, making the function responsible for testing for null argument values. The function can return an empty table, depending on its logic.

NO SQL, READS SQL DATA, or CONTAINS SQL

Indicates whether the function can execute any SQL statements and, if so, what type. See Table 56 on page 877 for a detailed list of the SQL statements that can be executed under each data access indication.

NO SQL

The function cannot execute any SQL statements.

READS SQL DATA

The function cannot execute any SQL statements that modify data. SQL statements that are not supported in any function return a different error. READS SQL DATA is the default.

CONTAINS SQL

The stored procedure cannot execute any SQL statements that read or modify data. SQL statements that are not supported in any function return a different error.

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.

Some functions with external actions can receive 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 in `SELECT`, `UPDATE`, `DELETE` or `INSERT` statements that refer to function.
- 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.

CREATE FUNCTION (external table)

NO EXTERNAL ACTION

The function does not take any action that changes the state of an object that DB2 does not manage. DB2 can use this information to optimize the processing of views for SELECT, UPDATE, DELETE or INSERT statements.

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

A scratchpad is not allocated and passed to the function. NO SCRATCHPAD is the default.

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'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.

FINAL CALL or NO FINAL CALL

Specifies whether a *first call* and a *final call* are made to the function.

FINAL CALL

A first call and final call are made to the function in addition to one or more *open*, *fetch*, or *close calls*. FINAL CALL is the default.

NO FINAL CALL

A first call and final call are not made to the function.

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 may 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.

CREATE FUNCTION (external table)

- *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.

DISALLOW PARALLEL

Specifies that DB2 does not consider parallelism for the function.

NO DBINFO or **DBINFO**

Specifies whether specific information that is known by DB2 is passed to the function when it is invoked.

NO DBINFO

No additional information is passed. NO DBINFO is the default.

DBINFO

An additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run-time authorization ID, the schema name, the name of a table or column that the function might be inserting into or updating, and identification of the database server that invoked the function. For details about the argument and its structure, see *DB2 Application Programming and SQL Guide*.

CARDINALITY *integer*

Specifies an estimate of the expected number of rows that the function returns. The number is used for optimization purposes. The value of *integer* must range from 0 to 2147483647.

If you do not specify CARDINALITY, DB2 assumes a finite value. The finite value is the same value that DB2 assumes for tables for which the RUNSTATS utility has not gathered statistics.

If a function has an infinite cardinality—the function never returns the “end-of-table” condition and always returns a row, then a query that requires the “end-of-table” to work correctly will need to be interrupted. Thus, avoid using such functions in queries that involve GROUP BY and ORDER BY.

NO COLLID or **COLLID** *collection-id*

Identifies the package collection that is 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, the package collection is the value of the CURRENT PACKAGESET special register.

NO COLLID is the default.

COLLID *collection-id*

The name of the package collection that is used when the external is executed.

WLM ENVIRONMENT

Identifies the MVS workload manager (WLM) application environment in which the function is to run. The *name* of the WLM environment is a long 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 MVS 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 on page 490.

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 *OS/390 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 2GB. 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 remains 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.

CREATE FUNCTION (external table)

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 *run-time-options*

Specifies the Language Environment run-time options to be used for the function. You must specify *run-time-options* as a character string that is no longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty string, DB2 does not pass any run-time options to Language Environment, and Language Environment uses its installation defaults.

For a description of the Language Environment run-time options, see *OS/390 Language Environment for OS/390 & VM Programming Reference*.

Notes

See “Notes” on page 489 for information about:

- Choosing data types for parameters
- Specifying the encoding scheme for parameters
- Determining the uniqueness of functions in a schema
- Running external functions in WLM environments

Examples

The following 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 will always return 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 DB2SQL
    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 column function with an application 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 for the schema or all schemas
- 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 authorization ID of the owner of the plan or package.

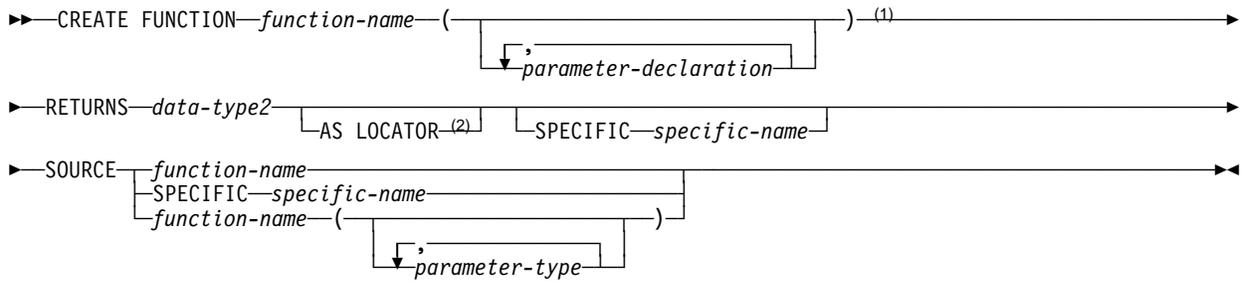
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. The specified function name can include a schema name (a qualifier). However, if the schema name is not the same as the SQL authorization ID, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the 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



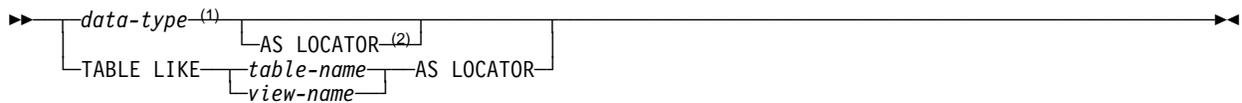
Notes:

- 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:



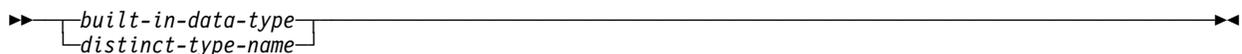
parameter-type:



Notes:

- 1 A LOB data type or distinct type based on a LOB data type must be no greater than 1M unless a locator is passed.
- 2 AS LOCATOR can be specified only for a LOB data type or a distinct type based on a LOB data type.

data-type:



- The unqualified form of *function-name* is a long 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 unqualified function name 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 plan or package was created or last rebound. If QUALIFIER was not specified, the schema name is the owner of the plan or package.
 - If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.
- The qualified form of *function-name* is a short SQL identifier (the schema name) followed by a period and a long SQL identifier.

The schema name must not begin with 'SYS' unless the schema name is 'SYSADM'.

The owner of the function is determined by how the CREATE FUNCTION 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 function.

(parameter-declaration,...)

Specifies the number of input parameters of the function and the data type of each parameter. All 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. 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 a long SQL identifier, and each name in the parameter list must not be the same as any other name.

CREATE FUNCTION (sourced)

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-data-type

The data type of the input parameter is a built-in data type. You can use the same built-in data types as for the CREATE TABLE statement except LONG VARCHAR or LONG VARGRAPHIC. Use VARCHAR or VARGRAPHIC with an explicit length instead.

If you do not specify a specific value for the data types that have length, precision, or scale attributes (CHAR, GRAPHIC, DECIMAL, NUMERIC, FLOAT), the defaults are as follows:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

For information on the data types, including the subtype of character data types (the FOR *subtype* DATA clause), see “built-in-data-type” on page built-in-data-type on page 575.

For parameters with a string data type, the CCSID clause indicates whether the encoding scheme of the parameter value is ASCII or EBCDIC. If you do not specify CCSID ASCII or CCSID EBCDIC, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

The data type of the input parameter is a distinct type. Any length, precision, scale, subtype, or encoding scheme attributes for the parameter are those of the source type of the distinct type.

Although 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.

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 83.) 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 length of the data type is implied. For example:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

For more information on default lengths of data types, see “built-in-data-type” on page built-in-data-type on page 575.

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. 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 *name* specified after TABLE LIKE must identify a table or view that exists at the current server. The name must not identify a declared temporary table. 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

CREATE FUNCTION (sourced)

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 SQLID 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.

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.

You can specify any built-in data type or distinct type that can be cast to from the data type of the source function's result. To specify a LONG VARCHAR or LONG VARGRAPHIC, use VARCHAR or VARGRAPHIC with an explicit length instead. (For information on casting data types, see "Casting between data types" on page 83.) For additional rules that apply to the data type that you can specify, see Rules for creating sourced functions on page 518.

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 a long SQL identifier. The qualified form is a short SQL identifier (the schema name) followed by a period and a long 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 ON, 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.

SOURCE

Specifies that the function that you are registering is a sourced function. A *sourced function* is implemented by another function (the *source function*). The source function can be any previously created user-defined function except an external table function. The source function can also be any built-in function except the following (if a particular syntax is shown, only the indicated form cannot be specified):

- COUNT(*)
- COUNT_BIG(*)
- CHAR(*datetime-expression*, *second-argument*) where *second-argument* is ISO, USA, EUR, JIS, or LOCAL
- COALESCE
- NULLIF
- RAISE-ERROR
- STRIP with any optional parameters specified
- VALUE

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.

EXECUTE authority is required on the source function.

To specify a built-in function as the source function, use the last syntax variation in the following list, *function-name (parameter-type, ...)*.

function-name

Identifies the function to be used as the source function by its function name. A schema name implicitly or explicitly qualifies the name. Only one function with this name must exist in the schema.

If you specify an unqualified *function-name*, DB2 uses the SQL path of the authorization ID (the value of the CURRENT PATH special register) to locate the function. The first schema that has only one function with this name on which the user has EXECUTE authority is selected. DB2 returns an error if it cannot find a function or encounters a schema that has more than one function with this name.

If you specify a qualified *function-name*, DB2 returns an error if there is no function with this name in the named schema or more than one function with this name exists in the schema.

SPECIFIC *specific-name*

Identifies the function to be used as the source function by its specific name. A schema name implicitly or explicitly qualifies the name.

CREATE FUNCTION (sourced)

If you specify an unqualified *specific-name*, DB2 uses the SQL path of the authorization ID (the value of the CURRENT PATH special register) to locate the schema. DB2 searches the SQL path and 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.

function-name (parameter-type,...)

Identifies the function to be used as the source function by its function signature. You must use this form of the syntax if the source function is a built-in function. You cannot use this form of the syntax if the source 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). Instead, identify the function with its function name, if unique, or with its specific name.

DB2 does not use function resolution to select the source function because the function signature uniquely identifies the function.

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 the named schema.

parameter-type,...

Provides a list of data types, separated by commas, that must match the data types of the parameters of the source function. DB2 uses the number of data types and the logical concatenation of the data types to identify the source function.

For data types that have length, precision, or scale attributes, you can either specify a value for the attribute or use a set of empty parentheses (with the noted exceptions):

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match. For example, DEC() will match a parameter whose data type is DEC(7,2).

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 for the source function. For example, DECIMAL(7,4) does not match a parameter whose data type is DECIMAL(7,2). Coding specific values for length, precision, scale, subtype, and encoding scheme attributes ensures that the source function you intend to use is used.

The specific value for FLOAT(*n*) does not have 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. For example:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. For a complete list of the default lengths of data types, see “CREATE TABLE” on page 570.

For either empty parentheses or specific values, the synonyms for data types are considered a match. For example, DEC and NUMERIC will match.

If you omit the FOR DATA or CCSID clause for data types with a subtype or encoding scheme attribute, DB2 ignores the attribute when determining whether the data types match.

If no function with the specified signature exists in the explicitly or implicitly specified schema, an error occurs.

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.

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 81). 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

For portability of functions across platforms that are not DB2 for OS/390, 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 the parameters with a 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. 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), and NUMERIC(4,2).

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; otherwise, an error occurs when the CREATE FUNCTION statement is executed.
- 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; otherwise, an error occurs. (For information on the promotion of data types, see “Casting between data types” on page 83.)

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; otherwise, an error occurs.

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.

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 GLOBAL TEMPORARY TABLE

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

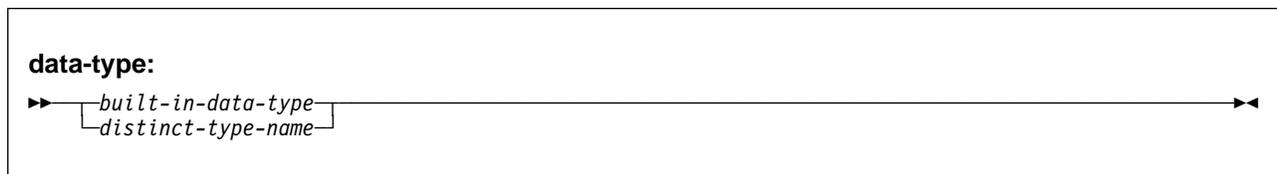
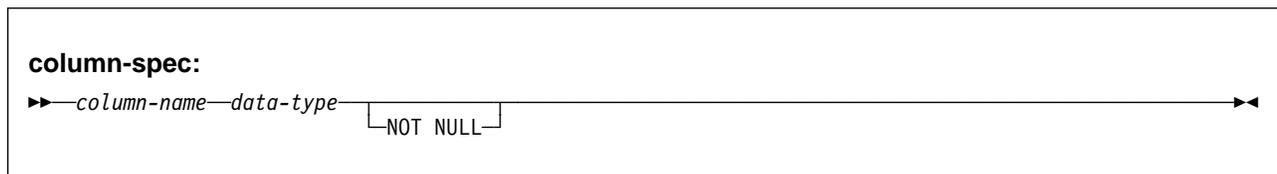
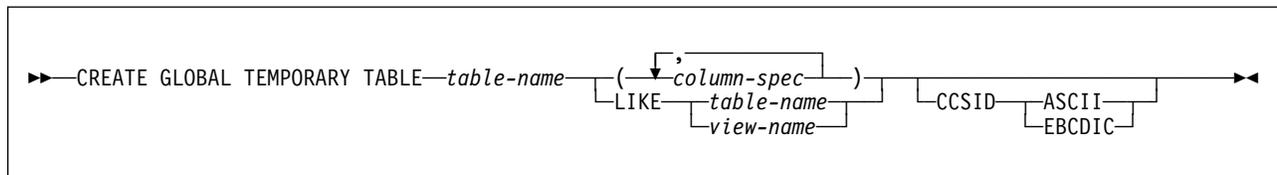
The privilege set that is defined below must include at least one of the following:

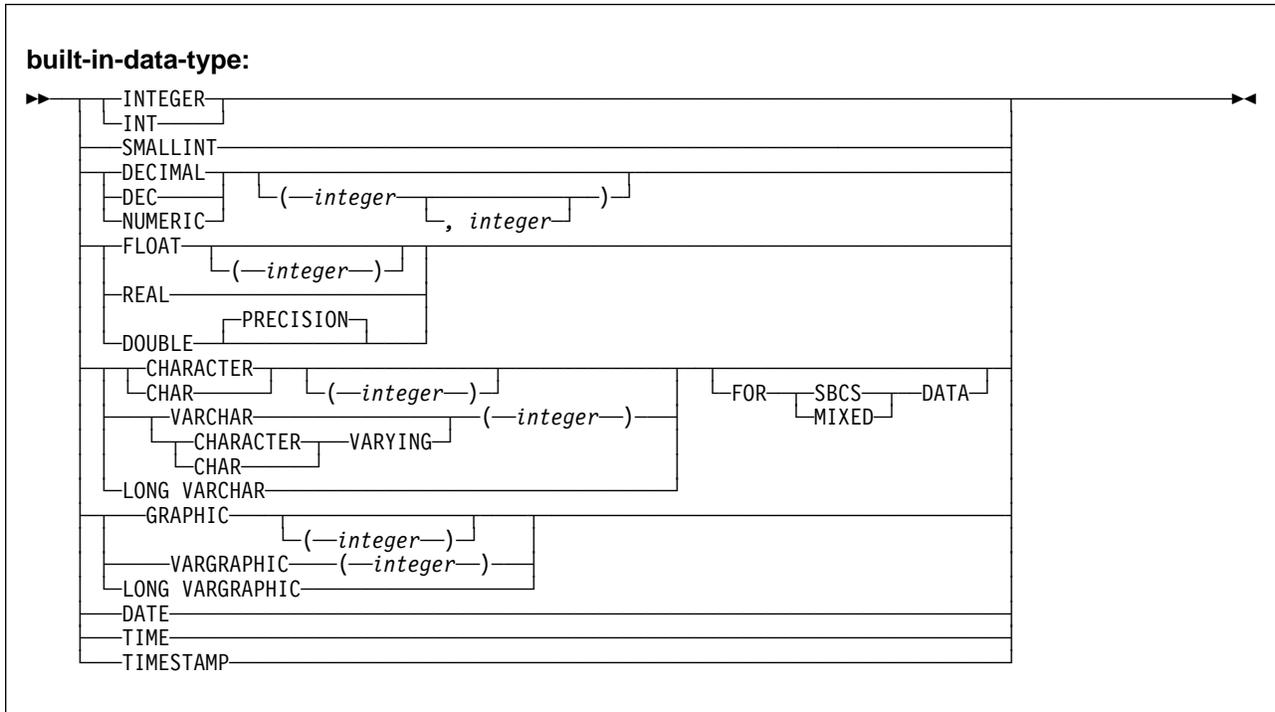
- The CREATETMTAB system privilege
- The CREATETAB database privilege for any database
- DBADM, DBCTRL, or DBMAINT authority for any database
- SYSADM or SYSCTRL authority

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 on page 570 for details.

Syntax





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 application server.

The qualification rules for *table-name* are the same as for *table-name* in the CREATE TABLE statement. (See “table-name” on page 574.)

The owner acquires ALL PRIVILEGES on the table WITH GRANT OPTION and the authority to drop the table.

column-spec

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-data-type

Any built-in data type that can be specified for the CREATE TABLE statement with the exception that you cannot define a temporary table with a LOB or ROWID column.

CREATE GLOBAL TEMPORARY TABLE

For more information on and the rules that apply to the data types, including the subtype of character data types (the *FOR subtype DATA* clause, see “built-in-data-type” on page built-in-data-type on page 575.

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, then 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.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed or graphic data is used.

The CCSID clause applies to all temporary tables. A temporary table that is created with the LIKE clause does not inherit the encoding scheme of the copied table.

The default for the CCSID clause is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

Notes

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, INSERT, or DELETE 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 application 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.
 - 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 *subselect* 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.

|
|

CREATE GLOBAL TEMPORARY TABLE

- 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 OF 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	DROP TABLE
COMMENT ON	CREATE TABLE LIKE	INSERT
CREATE ALIAS	CREATE VIEW	LABEL ON
CREATE FUNCTION	DESCRIBE TABLE	SELECT INTO

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 or nonpartitioning 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

Additional privileges might be required, as explained in the description of the BUFFERPOOL and USING STOGROUP clauses.

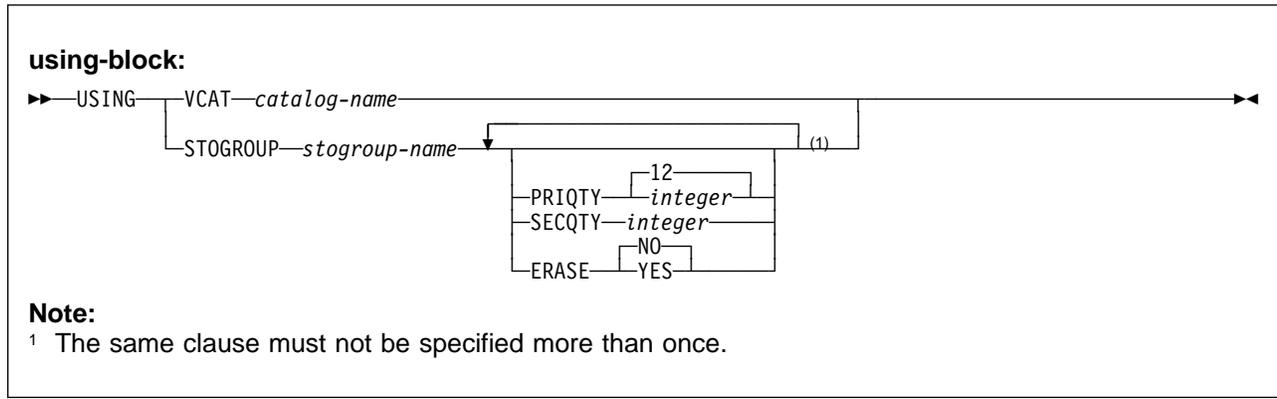
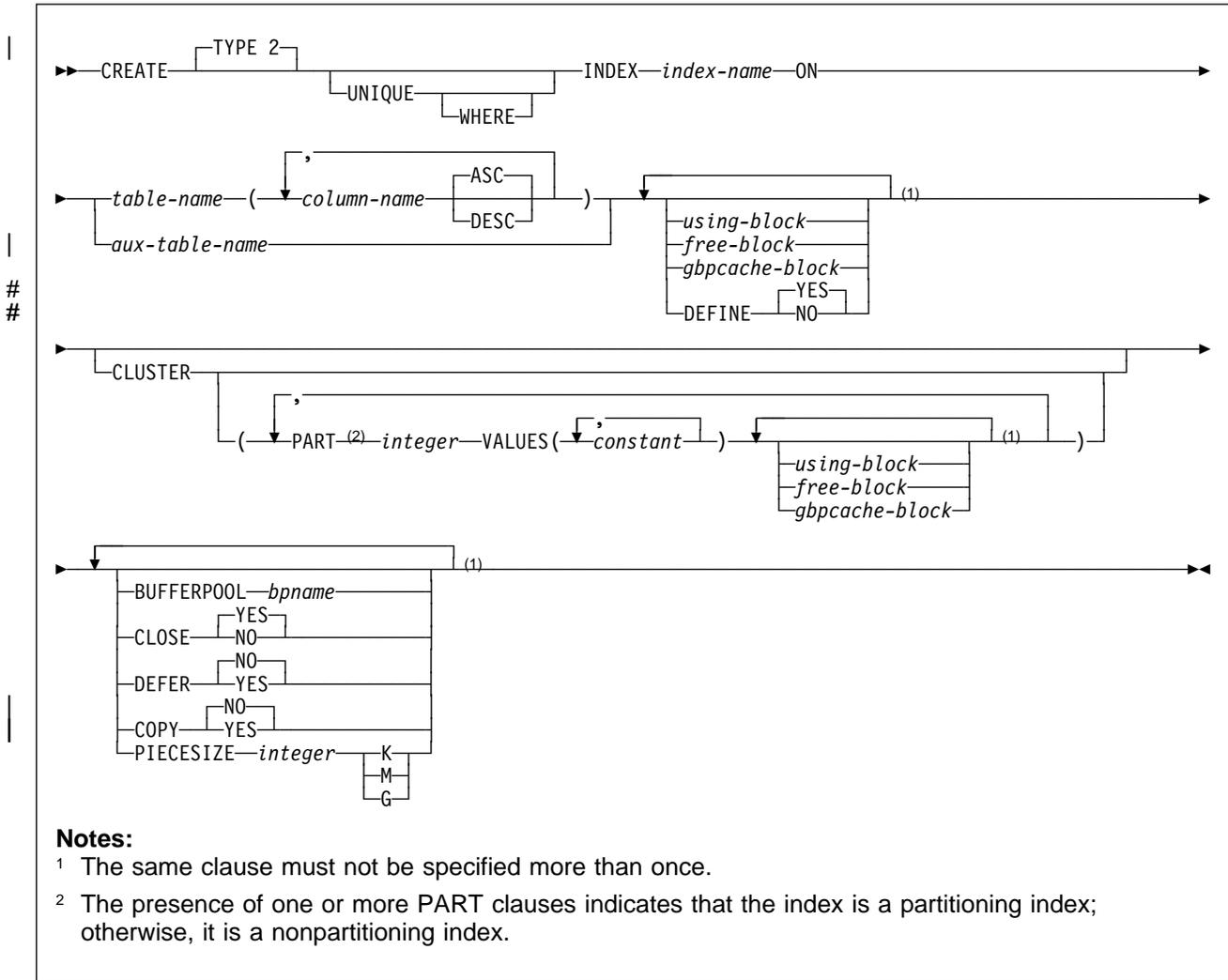
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package. If the specified index name includes a qualifier that is not the same as this authorization ID, the privilege set must include SYSADM or SYSCTRL authority, or DBADM or DBCTRL authority for the database.

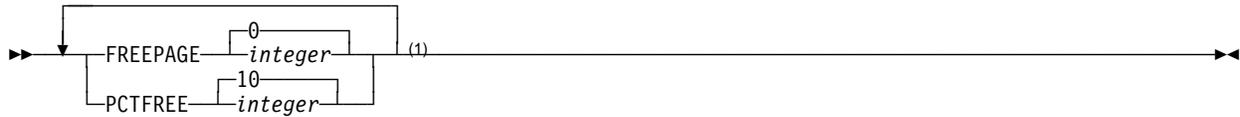
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. However, if the specified index name includes a qualifier that is not the same as this authorization ID, the following rules apply:

- If the privilege set includes SYSADM or SYSCTRL authority, or DBADM or DBCTRL authority for the database, any qualifier is valid.
- If the privilege set includes none of these authorities, the 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.

CREATE INDEX

Syntax



free-block:**Note:**

¹ The same clause must not be specified more than once.

gbpcache-block:

Description

TYPE 2

Specifies a type 2 index. The TYPE 2 clause is not required. A type 2 index is always created.

UNIQUE

Prevents the table from containing two or more rows with the same value of the index key. If any column of the key can contain null values, the meaning of “the same value” is determined by the use or omission of the option WHERE NOT NULL:

- If WHERE NOT NULL is omitted, any two null values are taken to be equal. For example, if the key is a single column, that column can contain no more than one null value.
- If WHERE NOT NULL is used, any two null values are taken to be unequal. If the key is a single column, that column can contain any number of null values, though its other values must be unique.

Unless DEFER YES is specified, the uniqueness 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. Refer to Section 2 (Volume 1) of *DB2 Administration Guide* for more information about using the REBUILD INDEX utility when duplicate keys exist for an index defined with UNIQUE and DEFER YES.

A table requires a unique index if you use the UNIQUE or PRIMARY KEY clause in the CREATE TABLE statement. DB2 implicitly creates those unique indexes if the CREATE TABLE statement is processed by the schema processor; 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,

CREATE INDEX

K is its primary key or any of the keys defined in the CREATE TABLE statement as UNIQUE and n is the number of such keys. After the definition of a table is complete, its status can return to incomplete only by the dropping of its primary index; in that case K is the primary key of the table and n is one.)

- 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 TABLE statement.

A table also requires a unique index if there is a ROWID column that is defined
as GENERATED BY DEFAULT.

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, then DB2 derives a unique name.

The qualification rules for an index name depend on the type of table as
follows:

- *Index on a base table or auxiliary table.* If the index name is unqualified and the statement is embedded in an application program, the owner of the index is the authorization ID that serves as the implicit qualifier for unqualified object names. This is the authorization ID in the QUALIFIER operand when the plan or package was created or last rebound. If QUALIFIER was not used, the owner of the index is the owner of the package or plan.

If the index name is unqualified and the statement is dynamically prepared, the SQL authorization ID is the owner of the index.

- *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 declared temporary table, or an auxiliary table.

table-name

Identifies the base 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 created temporary table.

(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, the same column more than once, or a LOB column (or a column with a distinct type that is based on a LOB data type). Do not qualify *column-name*.

The sum of the length attributes of the columns must not be greater than $255 - n$, where n is the number of columns that can contain null values.

ASC Puts the index entries in ascending order by the column. ASC is the default.

DESC Puts the index entries in descending order by the column.

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.

using-block

The components of the USING clause are discussed below, first for nonpartitioning indexes and then for partitioning indexes.

Using Clause for Nonpartitioning Indexes

For nonpartitioning 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 table's database. 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*. Because *catalog-name* is a short identifier, an alias must be used if the catalog name is longer than eight characters.

Conventions for index data set names are given in Section 2 (Volume 1) of *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.

STOGROUP *stogroup-name*

Specifies that DB2 will define and manage the data sets for the index. Each data set will be defined on a volume listed in the identified storage group. The values specified (or the defaults) for PRIQTY and SECQTY determine the primary and secondary allocations for the data set. If $PRIQTY + 118 \times SECQTY$ 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 MVS 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. The primary space allocation is at least *n* kilobytes, where *n* is:

12	If <i>integer</i> is less than 12 or PRIQTY is omitted
<i>integer</i>	If <i>integer</i> is between 12 and 4194304
4194304	If <i>integer</i> is greater than 4194304

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 are used by DB2 for purposes other than storing index entries.

SECQTY *integer*

Specifies the minimum secondary space allocation for a DB2-managed data set. 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

If *integer* is 0, no data set for the index can be extended. If you specify PRIQTY and do not specify SECQTY, the default for SECQTY is either 10% of PRIQTY or 3 times the index page size (4KB), whichever is larger.

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 *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 *DFSMS/MVS: Access Method Services for the Integrated Catalog* 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 Partitioning Indexes:

If the index is partitioning, there is a PART clause for each partition. Within a PART 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 nonpartitioning index applies to the entire index.

When a USING block is absent from a PART clause, the USING clause parameters for the partition depend on whether a USING clause is specified before the PART clauses.

- If the USING clause is specified, it applies to every PART 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 the catalog name in the form of a short 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 *n*th data set of the index, conforming to the DB2 naming convention for data sets set forth in Section 2 (Volume 1) of *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.

#

Do not specify VCAT for an index on a declared temporary table.

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 PART clause, or a default USING clause. DB2 assumes one and only one data set for each partition.

_____ End of using-block _____

_____ free-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 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 partitioning, 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 PART clause for that partition. Do not use more than one *free-block* in any PART clause.
- The values given in a *free-block* that is not in any PART clause.
- The default values FREEPAGE 0 and PCTFREE 10.

_____ End of free-block _____

_____ gbpcache-block _____

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

When there is inter-DB2 R/W interest on the index 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 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 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.

Hiperpools are not used for indexes or partitions that are defined with GBPCACHE ALL.

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 the index is partitioning, 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 PART clause for that partition. Do not use more than one *gbpcache-block* in any PART clause.
2. The value given in a *gbpcache-block* that is not in any PART clause.
3. The default value is CHANGED.

End of *gbpcache-block*

DEFINE

Specifies when the underlying data sets for the index are physically created.

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). DEFINE NO is ignored for user-managed data sets (USING VCAT is specified). DB2 uses the SPACE column in catalog table SYSINDEXPART to record the status of the data sets (undefined or allocated). 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 for an index on a declared temporary table.

CLUSTER

Specifies that the index is the cluster index of the table. Do not use CLUSTER if the index is for an auxiliary table, or if CLUSTER was used in the definition of an existing index on the table. If you do not use CLUSTER, the index is not a cluster index unless it is the first index defined on the table in a nonpartitioned table space. In this case, the first index implicitly serves as the cluster index until CLUSTER is used in the definition of another index on the table.

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 only affects data that is inserted with the INSERT statement; data is always loaded and reorganized in cluster order.

PART *integer*

A PART 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 PART 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.

VALUES(*constant*,...)

You must use at least one constant after VALUES in each PART clause. You can use as many as there are columns in the key. The concatenation of all the constants is the highest value of the key in the corresponding partition of the index.

The use of the constants to define key values is subject to these rules:

- The first constant corresponds to the first column of the key, the second constant to the second column, and so on. Each constant must have the same data type as its corresponding column.
- If a key includes a ROWID column (or a column with a distinct type that is sourced 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 constant that is specified for the corresponding ROWID column are considered.
- The precision and scale of a decimal constant must not be greater than the precision and scale of its corresponding 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'.
- Using fewer constants than there are columns in the key has the same effect as using the highest possible values for all omitted columns.
- 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 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 table spaces created with the LARGE or DSSIZE options, the constants you specify after VALUES are enforced. The value specified by the constants 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.

When you give a tablespace a DSSIZE value, you also give the same size to all the indexes that point to that tablespace.

BUFFERPOOL *bpname*

Identifies the buffer pool to be used for the index. The *bpname* must identify an activated 4KB 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 buffer pool for indexes in the database.

See “Naming conventions” on page 50 for more details about *bpname*. See Chapter 2 of *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 DSMAX is reached and there are no CLOSE YES page sets to close, CLOSE NO page sets 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 empty and DEFER YES is specified, the index is neither built nor placed in a rebuild pending status. Refer to Section 2 (Volume 1) of *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 piece (data set) for a nonpartitioning index. The subsequent keyword K, M, or G, indicates the units of the value specified in *integer*.

K Indicates that the *integer* value is to be multiplied by 1 024 to specify the maximum piece 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 piece 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 piece size in bytes. The integer must be a power of two between 1 and 64.

Table 34 shows the valid values for piece size, which depend on the size of the table space.

Table 34. Valid values of *PIECESIZE* clause

K units	M units	G units	Size attribute of table space
254 K	-	-	-
512 K	-	-	-
1024 K	1 M	-	-
2048 K	2 M	-	-
4096 K	4 M	-	-
8192 K	8 M	-	-
16384 K	16 M	-	-
32768 K	32 M	-	-
65536 K	64 M	-	-
131072 K	128 M	-	-
262144 K	256 M	-	-
524288 K	512 M	-	-
1048576 K	1024 M	1 G	-
2097152 K	2048 M	2 G	-
4194304 K	4096 M	4 G	LARGE, DSSIZE 4 G (or greater)
8388608 K	8192 M	8 G	DSSIZE 8 G (or greater)
16777216 K	16384 M	16 G	DSSIZE 16 G (or greater)
33554432 K	32768 M	32 G	DSSIZE 32 G (or greater)
67108864 K	65536 M	64 G	DSSIZE 64 G

As only a specification of the maximum amount of data that a piece can hold and not the actual allocation of storage, *PIECESIZE* has no effect on primary and secondary space allocation.

The default for piece size is 2 G (2 GB) for indexes that are backed by table spaces that were created without the *LARGE* or *DSSIZE* option, and 4 G (4 GB) for indexes that are backed by table spaces that were created with the *LARGE* or *DSSIZE* option.

Later if you change the *PIECESIZE* clause with the *ALTER INDEX* statement, be aware of the effect on the index, which 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

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.

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 and ASCII 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 nonpartitioning index by the number of data sets that you want. For example, to ensure that you have 5 data sets for the nonpartitioning index, and your nonpartitioning index is 10 MB (and not likely to grow much), specify PIECESIZE 2 M. If your nonpartitioning index is likely to grow, choose a larger value. Remember that 32 pieces is the limit if the underlying table space is not defined as LARGE (or as DSSIZE 4G or greater) and that the limit is 254 if the table space is defined as LARGE (or as DSSIZE 4G or greater).

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 only be dropped by dropping the associated table space. Nonpartitioning 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 on page 682.

Creating indexes on DB2 catalog tables: For details on creating indexes on catalog tables, see "SQL statements allowed on the catalog" on page 915.

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.

Creating indexes in a data sharing environment: DB2 will not invalidate any package or plan referred to in a table on which the index is created. However, the VALID column in SYSIBM.SYSPACKAGE or SYSIBM.SYSPLAN is marked as "A" unless the package or plan is already invalidated and marked VALID = "N". See "SYSIBM.SYSPACKAGE table" on page 963 for a description of SYSIBM.SYSPACKAGE and "SYSIBM.SYSPLAN table" on page 976 for a description of SYSIBM.SYSPLAN.

Examples

Example 1: Create a unique index, named DSN8610.XDEPT1, on table DSN8610.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 DSN8G610. Each data set (piece) 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— $512\text{KB} + (8 \times 64\text{KB}) = 1024\text{KB}$.

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 DSN8610.XDEPT1
  ON DSN8610.DEPT
  (DEPTNO ASC)
  USING STOGROUP DSN8G610
  PRIQTY 512
  SECQTY 64
  ERASE NO
  BUFFERPOOL BP1
  CLOSE YES
  PIECESIZE 1 M;
```

For the above example, the underlying data sets for the index will be created
immediately, which is the default (DEFINE YES). Assuming that table
DSN8610.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.

Example 2: Create a cluster index, named XEMP2, on table EMP in database DSN8610. Put the entries in ascending order by column EMPNO. Let DB2 define the data sets for each partition using storage group DSN8G610. 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

```
CREATE INDEX DSN8610.XEMP2
ON DSN8610.EMP
  (EMPNO ASC)
USING STOGROUP DSN8G610
  PRIQTY 36
  ERASE NO
  CLUSTER
  (PART 1 VALUES('H99'),
   PART 2 VALUES('P99'),
   PART 3 VALUES('Z99'),
   PART 4 VALUES('999'))
BUFFERPOOL BP1
CLOSE YES
COPY YES;
```

Example 3: Create a nonpartitioning index, named DSN8610.XDEPT1, on table DSN8610.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 (piece) is to hold 1 gigabyte of data at most before the next data set is used.

```
CREATE UNIQUE INDEX DSN8610.XDEPT1
ON DSN8610.DEPT
  (DEPTNO ASC)
USING VCAT DSNCAT
PIECESIZE 1048576 K;
```

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 DSN8D61A.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 DSN8610.XPHOTO
ON DSN8610.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 (external)

The CREATE PROCEDURE statement defines a stored 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 specified implicitly or explicitly.

Authorization

The privilege set that is defined below must include at least one of the following:

- The CREATEIN privilege for the schema or all schemas
- 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 authorization ID of the owner of the plan or package.

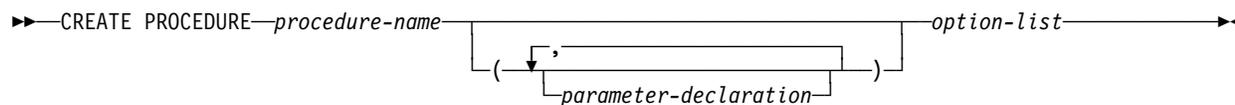
If the statement is dynamically prepared, the privilege set is the privileges that are held by the SQL authorization ID of the process. The specified procedure name can include a schema name (a qualifier). However, if the schema name is not the same as the SQL authorization ID, one of the following conditions must be met:

- The privilege set includes SYSADM or SYSCTRL authority.
- The SQL authorization ID of the process has the CREATEIN privilege on the schema.

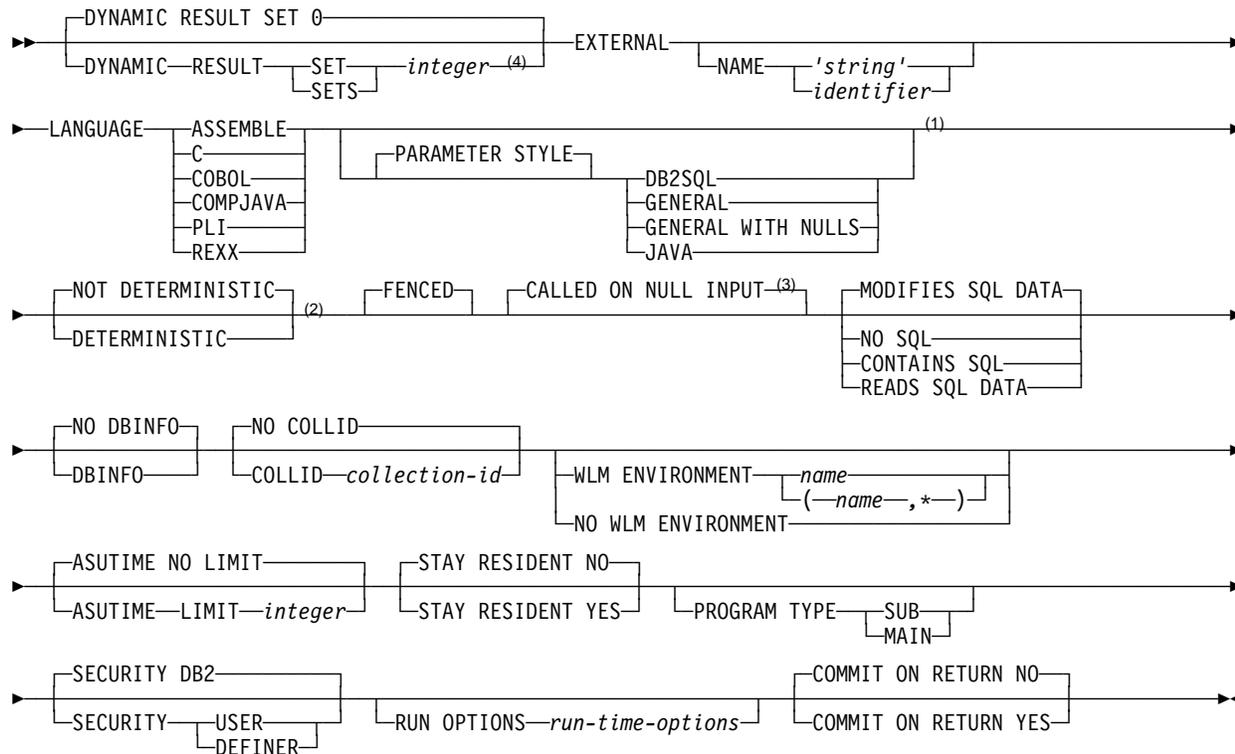
The authorization ID that is used to create the stored procedure must have authority to create programs that are to be run either in the DB2-established stored procedure address space or the specified workload manager (WLM) environment. In addition, if the stored procedure uses a distinct type as a parameter, this authorization ID must have the USAGE privilege on each distinct type that is a parameter.

CREATE PROCEDURE (external)

Syntax



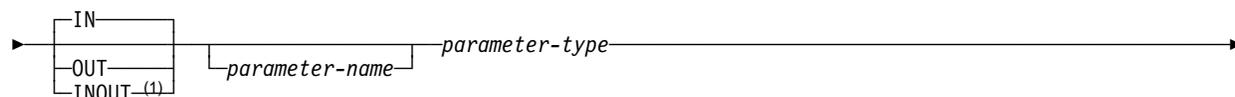
option-list:



Notes:

- 1 Synonyms include STANDARD CALL for DB2SQL, SIMPLE CALL for GENERAL, and SIMPLE CALL WITH NULLS for GENERAL WITH NULLS. The default value and values that can be specified depend on the value of LANGUAGE, as explained in detail in the description of the clause.
- 2 Synonyms include VARIANT for NOT DETERMINISTIC, and NOT VARIANT for DETERMINISTIC.
- 3 NULL CALL is a synonym for CALLED ON NULL INPUT.
- 4 Synonyms include RESULT SET for DYNAMIC RESULT SET and RESULT SETS for DYNAMIC RESULT SETS.

parameter-declaration:



Note:

- 1 For a REXX stored procedure, only one parameter can have type OUT or INOUT. That parameter must be declared last.

parameter-type:



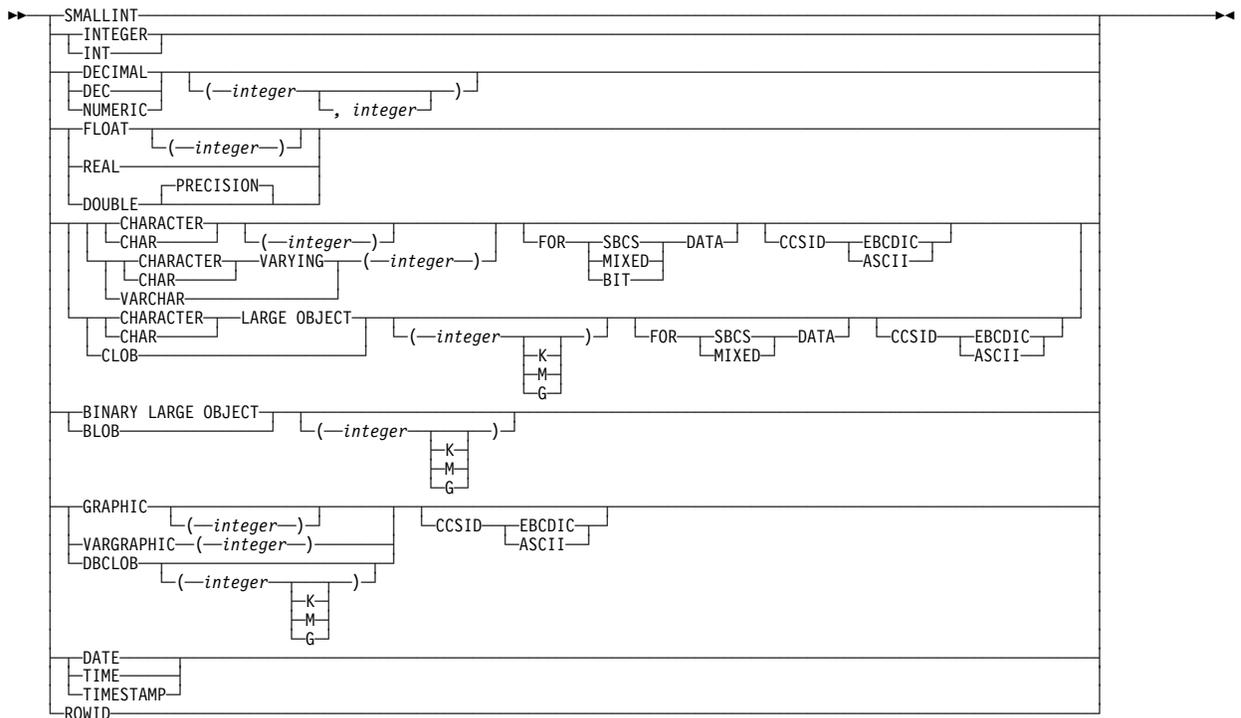
Notes:

- 1 A LOB data type or distinct type based on a LOB data type must be no greater than 1M unless a locator is passed.
- 2 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-data-type:



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 is implicitly or explicitly qualified by a schema. The name, including the implicit or explicit qualifier, must not identify an existing stored procedure at the current server.

- The unqualified form of *procedure-name* is a long SQL identifier. The unqualified name 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 plan or package was created or last rebound. If QUALIFIER was not specified, the schema name is the owner of the plan or package.

If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

- The qualified form of *procedure-name* is a short SQL identifier (the schema name) followed by a period and a long SQL identifier.

The schema name must not begin with 'SYS' unless the schema name is 'SYSPROC' or 'SYSADM'.

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.

(parameter-declaration,...)

Specifies the number of parameters of the stored procedure and the data type of each parameter. A parameter for a stored procedure can be used only for input, only for output, or for both input and output. Although not required, you can give each parameter a name.

IN Identifies the parameter as an input parameter to the stored procedure. The parameter does not contain a value when the stored procedure returns control to the calling SQL application.

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. *parameter-name* is a long identifier.

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-data-type

The data type of the parameter is a built-in data type. You can use the same built-in data types as for the CREATE TABLE statement except LONG VARCHAR or LONG VARGRAPHIC. Use VARCHAR or VARGRAPHIC with an explicit length instead.

If you do not specify a specific value for the data types that have length, precision, or scale attributes (CHAR, GRAPHIC, DECIMAL, NUMERIC, FLOAT), the defaults are as follows:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

For more information on the data types, including the subtype of character data types (the FOR *subtype* DATA clause), see “built-in-data-type” on page built-in-data-type on page 575.

For parameters with a string data type, the CCSID clause indicates whether the encoding scheme of the parameter value is ASCII or EBCDIC. If you do not specify CCSID ASCII or CCSID EBCDIC, the encoding scheme is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

distinct-type-name

Specifies that the data type of the parameter is a distinct type. Any length, precision, scale, or subtype attributes for the parameter are those of the source type of the distinct type.

Although an input parameter with a character data type has an implicitly or explicitly specified subtype (BIT, SBCS, or MIXED), the value that is actually passed in the input argument on the CALL statement can have any subtype. Therefore, conversion of the input data to the subtype of the parameter might occur when the procedure is called. 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.

CREATE PROCEDURE (external)

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. The columns have the same data type, length, precision, scale, subtype, encoding scheme, and CCSID as the identified table or view, as they are described in catalog tables SYSCOLUMNS and SYSTABLESPACE.

The *name* specified after TABLE LIKE must identify a table or view that exists at the current server. The name must not identify a declared temporary table. 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 SQLID 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, encoding scheme, and CCSID of these columns must match exactly. The description of the table or view at the time the CREATE PROCEDURE statement was executed is used.

For more information about using table locators, see *DB2 Application Programming and SQL Guide*.

FENCED

Specifies that the stored procedure runs in an external address space to prevent user programs from corrupting DB2 storage.

FENCED is the default.

DYNAMIC RESULT SET *integer* or **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.

EXTERNAL

Specifies the program that runs when the procedure name is specified in a CALL statement.

NAME '*string*' or *identifier*

Identifies the user-written code that implements the stored procedure.

If LANGUAGE is COMPJAVA, the name is a string constant in the format *class-name.method-name* or *package-name.class-name.method-name* that is no longer than 254 bytes. For other values of LANGUAGE, the name can be a string constant that is no longer than 8 characters or a short identifier, and must conform to the naming conventions for MVS 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 *procedure-name*' is implicit. In this case, *procedure-name* must not be longer than 8 characters.

The program does not need to exist when the CREATE PROCEDURE statement is executed. However, it must exist and be accessible by the current server when a CALL statement to the stored procedure is issued.

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 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.

COMPJAVA

The stored procedure is written in Java and is a compiled program.

PLI

The stored procedure is written in PL/I.

CREATE PROCEDURE (external)

```
#          REXX
#          The stored procedure is written in REXX. Do not specify LANGUAGE
#          REXX when PARAMETER STYLE DB2SQL or NO WLM ENVIRONMENT
#          is in effect. When REXX is specified, the procedure must use PARAMETER
#          STYLE GENERAL or GENERAL WITH NULLS.

|          PARAMETER STYLE
|          Identifies the linkage convention used to pass parameters to 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.

|          DB2SQL
|          In addition to the parameters on the CALL statement, the following
|          arguments are also passed to the stored procedure:
|
|          • A null indicator for each parameter on the CALL statement
|          • The SQLSTATE to be returned to DB2
|          • The qualified name of the stored procedure
|          • The specific name of the stored procedure
|          • The SQL diagnostic string to be returned to DB2
#
#          An additional parameter might also be passed:
#
#          • The DB2INFO structure, if DBINFO is specified
#
#          DB2SQL is the default unless LANGUAGE is COMPJAVA. Do not specify
#          DB2SQL when LANGUAGE REXX is in effect.

|          GENERAL
|          Only the parameters on the CALL statement are passed to the stored
|          procedure. The parameters cannot be null.

|          GENERAL WITH NULLS
|          In addition to the parameters on the CALL statement, another argument is
|          also passed to the stored procedure. The additional argument contains a
|          vector of null indicators for each of the parameters on the CALL statement
|          that enables the stored procedure to accept or return null parameter values.

#          JAVA
#          The stored procedure uses a convention for passing parameters that
#          conforms to the Java and SQLJ specifications. INOUT and OUT
#          parameters are passed as single-entry arrays. The DBINFO structure is not
#          passed.
#
#          JAVA can be specified only if LANGUAGE is COMPJAVA, and JAVA is the
#          only valid value for PARAMETER STYLE when LANGUAGE is
#          COMPJAVA. If LANGUAGE is COMPJAVA and PARAMETER STYLE is
#          not specified, PARAMETER STYLE defaults to JAVA.

#          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
#          DB2SQL.
```

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the stored procedure returns the same result from successive calls with identical input arguments.

NOT DETERMINISTIC

The stored procedure might not return the same result from successive calls with identical input arguments. NOT DETERMINISTIC is the default.

DETERMINISTIC

The stored procedure returns the same result from successive calls with identical input arguments.

DB2 does not verify that the stored procedure code is consistent with the specification of DETERMINISTIC or NOT DETERMINISTIC.

NO SQL, MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL DATA

Indicates whether the stored procedure can execute any SQL statements and, if so, what type. See Table 56 on page 877 for a detailed list of the SQL statements that can be executed under each data access indication.

NO SQL

The stored procedure cannot execute any SQL statements.

MODIFIES SQL DATA

The stored procedure can execute any SQL statement except those statements that are not supported in any stored procedure.

MODIFIES SQL DATA is the default.

READS SQL DATA

The stored procedure cannot execute SQL statements that modify data. SQL statements that are not supported in any stored procedure return a different error.

CONTAINS SQL

The stored procedure cannot execute any SQL statements that read or modify data. SQL statements that are not supported in any stored procedure return a different error.

NO DBINFO or DBINFO

Specifies whether specific information known by DB2 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 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 DB2SQL is specified.

CREATE PROCEDURE (external)

NO COLLID or **COLLID** *collection-id*

Identifies the package collection that is 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 calling program does not use a package, the package collection is set to the value of special register CURRENT PACKAGESET.

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 MVS workload manager (WLM) 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 a long 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 MVS 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 on page 553.

NO WLM ENVIRONMENT

Indicates that the stored procedure is to run in the DB2-established stored procedure address space.

Do not specify NO WLM ENVIRONMENT if you implicitly or explicitly define the stored procedure with any of the following clauses or parameters:

- The PROGRAM TYPE SUB clause
- The SECURITY USER or SECURITY DEFINER clause
- The LANGUAGE REXX or LANGUAGE COMPJAVA clause
- Parameters with a LOB data type or a distinct type based on a LOB data type

To define a stored procedure that is to run in the DB2-established stored procedure address space, you must have appropriate authority for the address space. For an example of a RACF command that provides this authorization, see Running stored procedures on page 553.

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.

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 *OS/390 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 GB. 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 remains resident in memory when the stored procedure ends.

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.

Specify PROGRAM TYPE SUB for stored procedures with a LANGUAGE
value of REXX. Do not specify PROGRAM TYPE SUB when NO WLM
ENVIRONMENT is in effect.

MAIN

The stored procedure runs as a main routine.

The default for PROGRAM TYPE depends on the value of special register CURRENT RULES. The default is:

- 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

CREATE PROCEDURE (external)

| security product protects, the access is performed using the authorization
| ID associated with the stored procedure address space. DB2 is the default.
SECURITY DB2 is the only valid choice when NO WLM ENVIRONMENT is
specified.

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. Do not specify
SECURITY USER when NO WLM ENVIRONMENT is in effect.

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. Do not specify SECURITY
DEFINER when NO WLM ENVIRONMENT is in effect.

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. If you do not specify RUN OPTIONS or pass an empty string, DB2 does
| not pass any run-time options to Language Environment, and Language
| Environment uses its installation defaults.

If you specify the RUN OPTIONS parameter for a stored procedure with a
LANGUAGE value of COMPJAVA, DB2 ignores the run-time options.

| For a description of the Language Environment run-time options, see *OS/390
| Language Environment for OS/390 & VM 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.

CALLED ON NULL INPUT

| Specifies that the stored procedure will be called even if any of the input
| arguments is null, making the procedure responsible for testing for null

argument values. The result is the null value. CALLED ON NULL INPUT is the default.

Notes

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 81). 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

For portability of functions across platforms that are not DB2 for OS/390, 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 the parameters with a string data type (both input and output parameters) must be the same—either all ASCII or all EBCDIC.

Running stored procedures: You can use the WLM ENVIRONMENT clause to identify the MVS 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.

If you use NO WLM ENVIRONMENT, the stored procedure will run in the DB2-established stored procedure address space, where there is no ability to isolate one group of programs from another.

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)
```

Similarly, the following RACF command authorizes the same user to define stored procedures that run in the DB2 stored procedure address space named DB2ASPAS.

```
PERMIT DB2A.WLMENV.DB2ASPAS CLASS(DSNR) ID(DB2USER1) ACCESS(READ)
```

CREATE PROCEDURE (external)

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.

Examples

Example 1: Create the definition for a stored procedure that is written in COBOL. The procedure accepts an assembly part number and returns the number of parts that make up the assembly, the total part cost, and a result set. The result set lists the part numbers, quantity, and unit cost of each part. Assume that the input parameter cannot contain a null value and that the procedure is to run in a WLM environment called PARTSA.

```
CREATE PROCEDURE SYSPROC.MYPROC(IN INT, OUT INT, OUT DECIMAL(7,2))
    LANGUAGE COBOL
    EXTERNAL NAME MYMODULE
    PARAMETER STYLE GENERAL
    WLM ENVIRONMENT PARTSA
    DYNAMIC RESULT SETS 1;
```

Example 2: Create the definition for the stored procedure described in Example 1, except use the linkage convention that passes more information than the parameter specified on the CALL statement. Specify Language Environment run-time options HEAP, BELOW, ALL31, and STACK.

```
CREATE PROCEDURE SYSPROC.MYPROC(IN INT, OUT INT, OUT DECIMAL(7,2))
    LANGUAGE COBOL
    EXTERNAL NAME MYMODULE
    PARAMETER STYLE DB2SQL
    WLM ENVIRONMENT PARTSA
    DYNAMIC RESULT SETS 1
    RUN OPTIONS 'HEAP(,,ANY),BELOW(4K,,),ALL31(ON),STACK(,,ANY,)';
```

CREATE PROCEDURE (SQL)

The CREATE PROCEDURE statement registers an SQL procedure with an
application server and specifies the source statements for an SQL procedure.

Invocation

This statement can only be dynamically prepared, and the DYNAMICRULES run
behavior must be specified implicitly or explicitly.

This statement is intended to be processed using one of the following methods:

- # • Using IBM DB2 Stored Procedure Builder
- # • Using JCL
- # • Using the DB2 for OS/390 SQL procedure processor (DSNTPSMP)

For more information on preparing SQL procedures for execution, see Section 7 of
DB2 Application Programming and SQL Guide. Issuing the CREATE
PROCEDURE statement from another context will result in an incomplete
procedure definition even though the statement processing returns without error.

Authorization

The privilege set that is defined below must include at least one of the following:

- # • The CREATEIN privilege for the schema or all schemas
- # • SYSADM or SYSCTRL authority

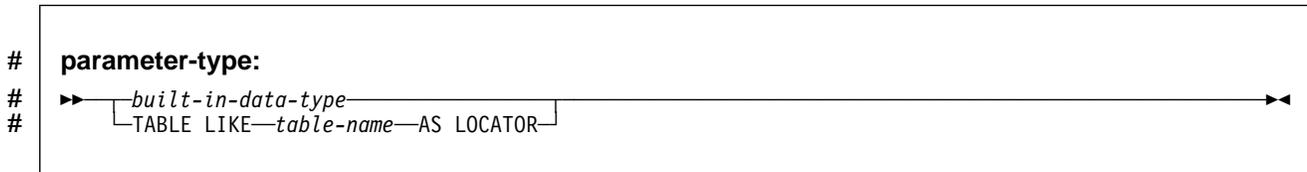
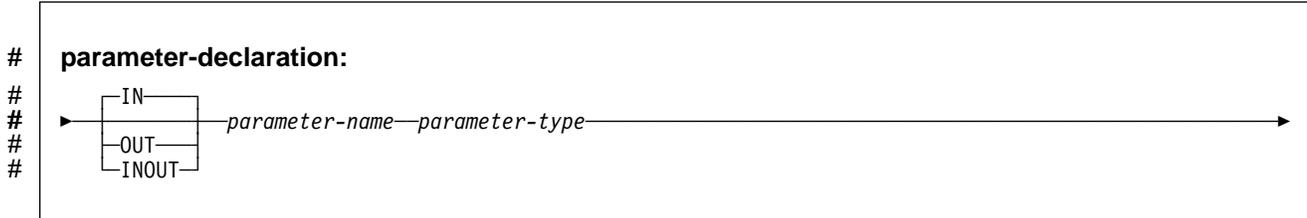
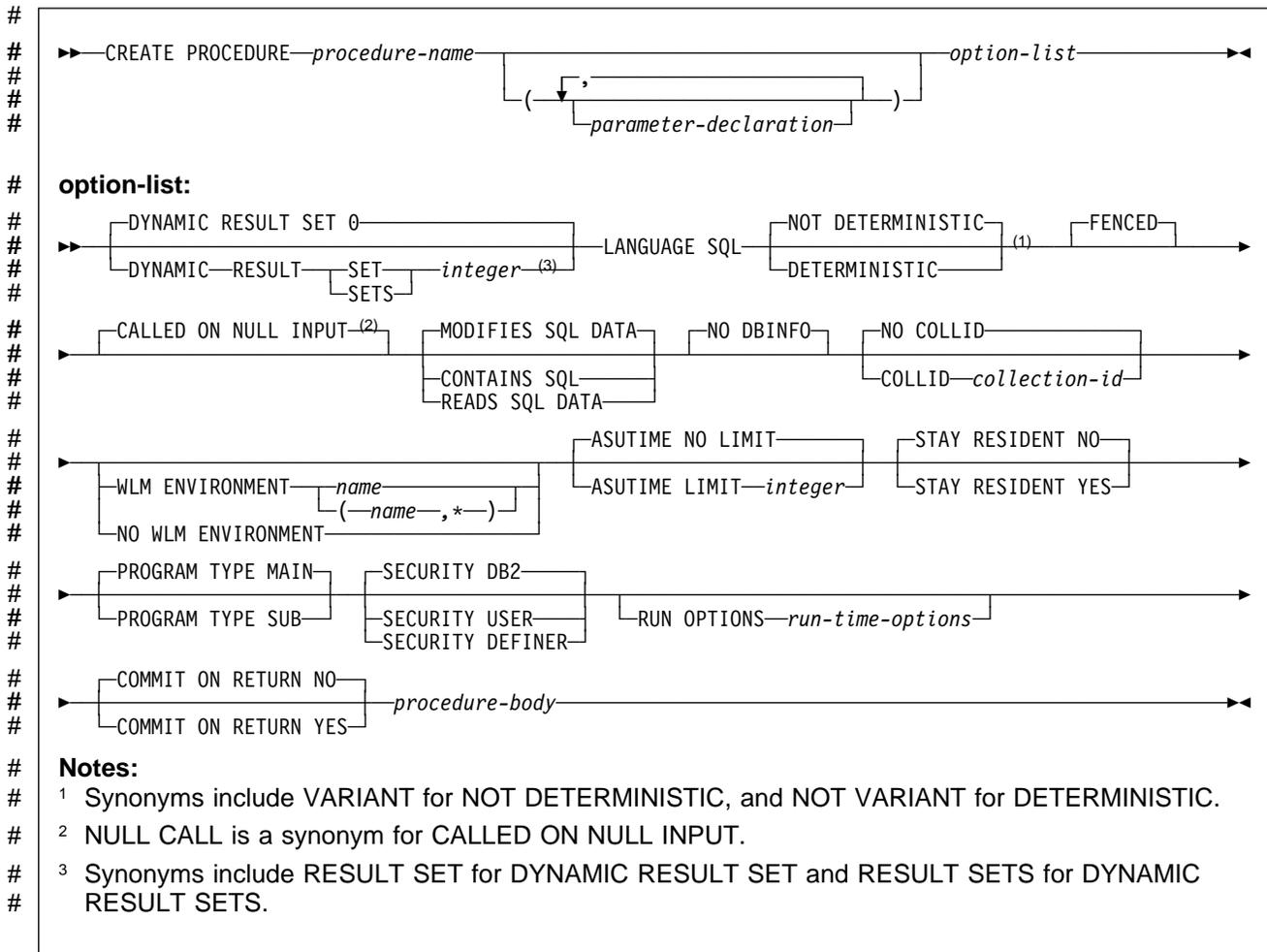
The authorization ID that matches the schema name implicitly has the CREATEIN
privilege on the schema.

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

The authorization ID that is used to create the stored procedure must have
authority to create programs that are to be run either in the DB2-established stored
procedure address space or the specified workload manager (WLM) environment.

CREATE PROCEDURE (SQL)

Syntax



CREATE PROCEDURE (SQL)

```
#           The owner is implicitly given the EXECUTE privilege with the GRANT option for
#           the procedure.

#           (parameter-declaration,...)
#           Specifies the number of parameters of the stored procedure and the data type
#           of each parameter. A parameter for a stored procedure can be used only for
#           input, only for output, or for both input and output. You must give each
#           parameter a name.

#           IN Identifies the parameter as an input parameter to the stored procedure.
#           The parameter does not contain a value when the stored procedure returns
#           control to the calling SQL application.

#           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. parameter-name is a long identifier. A parameter
#           name cannot be an SQL reserved word. For a list of SQL reserved words,
#           see Appendix E, "SQL reserved words" on page 1027.

#           data-type
#           Specifies the data type of the parameter.

#           built-in-data-type
#           The data type of the parameter is a built-in data type. You can use the
#           same built-in data types as for the CREATE TABLE statement except
#           LONG VARCHAR or LONG VARGRAPHIC. Use VARCHAR or
#           VARGRAPHIC with an explicit length instead.

#           For more information on the data types, including the subtype of
#           character data types (the FOR subtype DATA clause), see
#           "built-in-data-type" on page built-in-data-type on page 575.

#           If you do not specify a specific value for the data types that have
#           length, precision, or scale attributes (CHAR, GRAPHIC, DECIMAL,
#           NUMERIC, FLOAT), the defaults are as follows:

#           CHAR           CHAR(1)
#           GRAPHIC        GRAPHIC(1)
#           DECIMAL       DECIMAL(5,0)
#           FLOAT         DOUBLE (length of 8)

#           For parameters with a string data type, the CCSID clause indicates
#           whether the encoding scheme of the parameter value is ASCII or
#           EBCDIC. If you do not specify CCSID ASCII or CCSID EBCDIC, 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 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.

For more information about the TABLE LIKE clause, see TABLE LIKE

on page 546. For more information about using table locators, see *DB2*

Application Programming and SQL Guide.

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. 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 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 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.

FENCED

Specifies that the stored procedure runs in an external address space to

prevent user programs from corrupting DB2 storage.

FENCED is the default.

DYNAMIC RESULT SET *integer* or 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.

LANGUAGE

Specifies the application programming language in which the stored procedure

is written.

SQL

The stored procedure is written in DB2 SQL procedure language.

NOT DETERMINISTIC or DETERMINISTIC

Specifies whether the stored procedure returns the same result from successive

calls with identical input arguments.

NOT DETERMINISTIC

The stored procedure might not return the same result from successive

calls with identical input arguments. NOT DETERMINISTIC is the default.

CREATE PROCEDURE (SQL)

DETERMINISTIC
The stored procedure returns the same result from successive calls with
identical input arguments.

DB2 does not verify that the stored procedure code is consistent with the
specification of DETERMINISTIC or NOT DETERMINISTIC.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL DATA
Indicates whether the stored procedure can execute any SQL statements and,
if so, what type. See Table 56 on page 877 for a detailed list of the SQL
statements that can be executed under each data access indication.

MODIFIES SQL DATA
The stored procedure can execute any SQL statement except those
statements that are not supported in any stored procedure.

MODIFIES SQL DATA is the default.

READS SQL DATA
The stored procedure cannot execute SQL statements that modify data.
SQL statements that are not supported in any stored procedure return a
different error.

CONTAINS SQL
The stored procedure cannot execute any SQL statements that read or
modify data. SQL statements that are not supported in any stored
procedure return a different error.

NO DBINFO
Specifies whether specific information known by DB2 is passed to the stored
procedure when it is invoked.

NO DBINFO
Additional information is not passed. Only NO DBINFO is allowed for SQL
procedures.

NO COLLID or COLLID *collection-id*
Identifies the package collection that is 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 calling program does not
use a package, the package collection is set to the value of special register
CURRENT PACKAGESET.

NO COLLID is the default.

COLLID *collection-id*
The package collection for the stored procedure is the one specified.

WLM ENVIRONMENT
Identifies the MVS workload manager (WLM) 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 a long 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 MVS 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 on page 553 .

#           NO WLM ENVIRONMENT
#           Indicates that the stored procedure is to run in the DB2-established stored
#           procedure address space.

#           Do not specify NO WLM ENVIRONMENT if you implicitly or explicitly define the
#           stored procedure with SECURITY USER, SECURITY DEFINER, or PROGRAM
#           TYPE SUB or if there are any LOB parameters.

#           To define a stored procedure that is to run in the DB2-established stored
#           procedure address space, you must have appropriate authority for the address
#           space. For an example of a RACF command that provides this authorization,
#           see Running stored procedures on page 553 .

#           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.

#           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 OS/390 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 GB.
#           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 remains resident in
#           memory when the stored procedure ends.

#           NO
#           The load module is deleted from memory after the stored procedure ends.
#           NO is the default.

```

CREATE PROCEDURE (SQL)

```
#           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.

#           MAIN
#           The stored procedure runs as a main routine. MAIN is the default for SQL
#           procedures.

#           SUB
#           The stored procedure runs as a subroutine.

#           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.

#           RUN OPTIONS run-time-options
#           Specifies the Language Environment run-time options to be used for the stored
#           procedure. You must specify run-time-options as a character string that is no
#           longer than 254 bytes. If you do not specify RUN OPTIONS or pass an empty
#           string, DB2 does not pass any run-time options to Language Environment, and
#           Language Environment uses its installation defaults.

#           For a description of the Language Environment run-time options, see OS/390
#           Language Environment for OS/390 & VM 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.
#
#           CALLED ON NULL INPUT
#           Specifies that the stored procedure will be called even if any of the input
#           arguments is null, making the procedure responsible for testing for null
#           argument values. The result is the null value. CALLED ON NULL INPUT is the
#           default.
#
#           procedure-body
#           Specifies the source code for an SQL procedure. See “Chapter 7. SQL
#           procedure statements” on page 847 for information on how to write a
#           procedure body.

```

Notes

```

#           The following restrictions apply to the use of parameters in SQL procedures:
#
#           • If IN is specified for a parameter in an SQL procedure, the parameter cannot
#           be modified within the SQL procedure body.
#
#           • If OUT is specified for a parameter in an SQL procedure, the parameter can be
#           used only as the target of an assignment in the SQL procedure body. The
#           parameter cannot be checked or used to set other variables. If the parameter is
#           not set, DB2 returns the null value to the caller.
#
#           See “Notes” on page 553 for information about:
#
#           • Choosing data types for parameters
#           • Specifying the encoding scheme for parameters
#           • Environments for running stored procedures
#           • Accessing result sets from nested stored procedures

```

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:

```

CREATE PROCEDURE (SQL)

```
#           • The procedure runs in a WLM environment called PARTSA.
#           • The same input always produces the same output.
#           • SQL work is committed on return to the caller.
#           • The Language Environment run-time options to be used when the SQL
#             procedure executes are 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'.
#
# CREATE PROCEDURE UPDATE_SALARY_1
#   (IN EMPLOYEE_NUMBER CHAR(10),
#   IN RATE DECIMAL(6,2))
#   LANGUAGE SQL
#   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 "Chapter 7. SQL procedure
# statements" on page 847.
```

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 authorization ID of 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

```

▶▶ CREATE STOGROUP stogroup-name VOLUMES ( volume-id (1) ) VCAT catalog-name

```

Note:

¹ The same *volume-id* must not be specified more than once.

Description

stogroup-name

Names the storage group. The name must not identify a storage group that exists at the current server.

VOLUMES(*volume-id*,...) or **VOLUMES**(' *',...)

Defines the volumes of the storage group. Each *volume-id* is a volume serial number of a storage volume. It can have a maximum of six characters and is specified as an identifier or a string constant.

Asterisks are recognized only by Storage Management Subsystem (SMS).

Contact your site's storage administrator to determine if the SMS Guaranteed Space attribute applies. If SMS Guaranteed Space does not apply for SMS-managed data sets, it is recommended that the VOLUMES clause be specified with one asterisk, VOLUMES(' *'). If SMS Guaranteed Space does apply, contact your site storage manager and refer to *DFSMS/MVS: Access Method Services for the Integrated Catalog* and *DFSMS/MVS: Storage Administration Reference for DFSMSdfp* for information on how to specify the VOLUMES clause.

CREATE STOGROUP

VCAT *catalog-name*

Identifies the integrated catalog facility catalog for the storage group. You must specify the catalog name in the form of a short identifier. Thus, you must specify an alias 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 Section 2 (Volume 1) of *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.

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 MVS 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. Thus, there is no point in creating a storage group with more than 133 volumes.

MVS imposes a limit on the number of volumes that can be allocated per data set: 59 at this writing. 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 volume IDs: When processing the VOLUMES clause, DB2 does not check the existence of the volumes or determine the types of devices that they identify. Later, whenever the storage group is used to allocate data sets, the list of volumes is passed in the specified order to Data Facilities (DFSMSDfp), which does

#

the actual work. See Section 2 (Volume 1) of *DB2 Administration Guide* for more information about creating DB2 storage groups.

Example

Create storage group, DSN8G610, of volumes ABC005 and DEF008. DSNCAT is the integrated catalog facility catalog name.

```
CREATE STOGROUP DSN8G610
  VOLUMES (ABC005,DEF008)
  VCAT DSNCAT;
```

CREATE SYNONYM

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.

Authorization

None required.

Syntax

```
▶▶ CREATE SYNONYM synonym FOR authorization-name . table-name | view-name ▶▶
```

Description

synonym

Names the synonym. The name must not identify a synonym, table, view, or alias owned by authorization ID *x*. If the statement is embedded in an application program, *x* is the owner of the plan or package. If the statement is dynamically prepared, *x* is the value of CURRENT SQLID. In either case, *x* becomes the owner of the synonym.

FOR *authorization-name.table-name* or *authorization-name.view-name*

Identifies the object to which the synonym applies. The name must consist of two parts and must identify a table, view, or alias that exists at the current server. If a table is identified, it must not be an auxiliary table or a declared temporary table. If an alias is identified, it must be an alias for a table or view at the current server and the synonym is defined for that table or view.

#

Notes

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 824 for details on changing the value of the CURRENT SQLID special register.

The authorization ID recorded as the owner of a synonym is the only authorization ID for which the synonym is defined and the only authorization ID that can be used to drop it.

If an alias is used to denote the table or view, the name of that table or view, not the alias, is recorded in the catalog as the definition of the synonym. That severs the connection between the synonym and alias, and even if the alias is dropped and redefined, the synonym is still in effect and names the original table or view.

Example

Define DEPT as a synonym for the table DSN8610.DEPT.

```
CREATE SYNONYM DEPT  
FOR DSN8610.DEPT;
```

This example does not work if the current SQL authorization ID is DSN8610.

CREATE TABLE

The CREATE TABLE statement defines a table at the current server. 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 implicitly or explicitly specified by the IN clause
- DBADM, DBCTRL, or DBMAINT authority for the database
- SYSADM or SYSCTRL authority

Additional privileges might be required when:

- 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.

See the description of the appropriate clauses for details about these privileges.

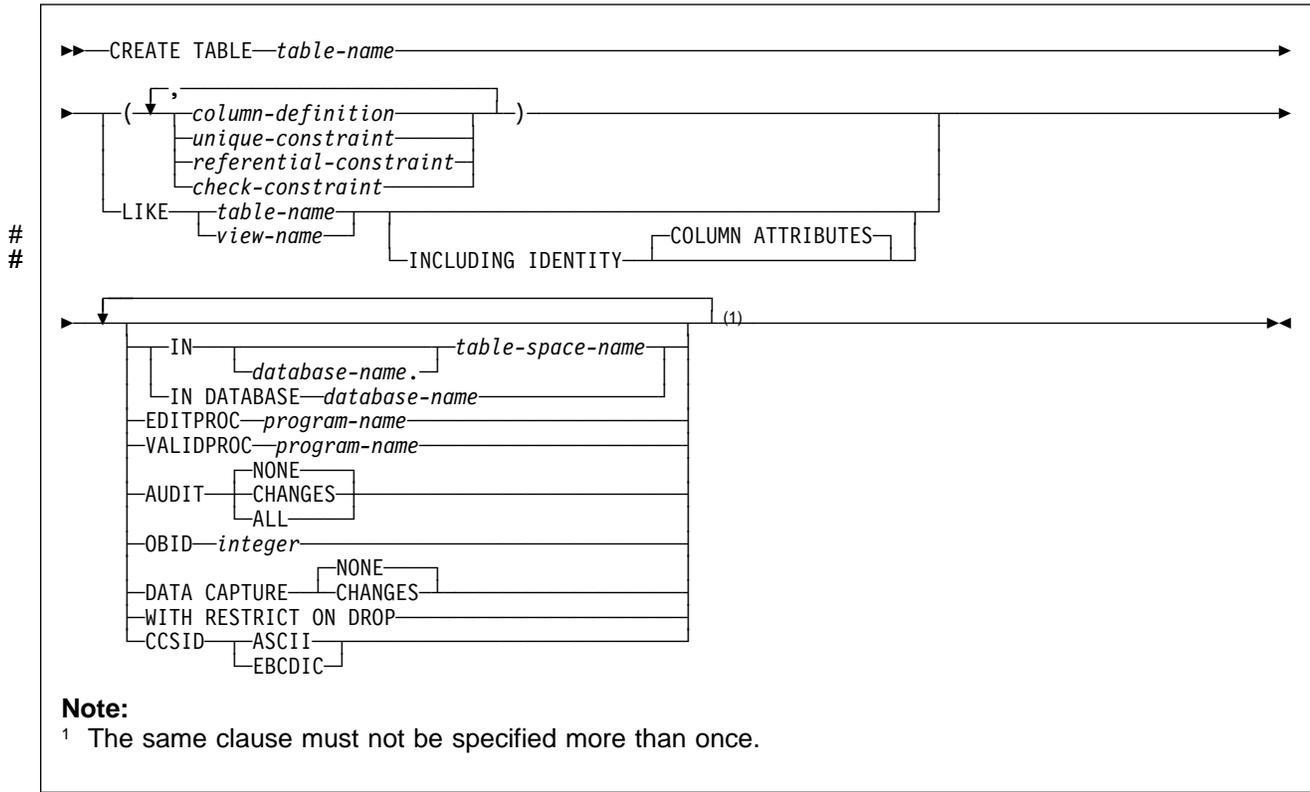
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package. If the specified table name includes a qualifier that is not the same as this authorization ID, the privilege set must include SYSADM or SYSCTRL authority, DBADM authority for the database, or DBCTRL authority for the database.

If 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 specified table name includes a qualifier that is not the same as this authorization ID, the following rules apply:

1. If the privilege set includes SYSADM or SYSCTRL authority, DBADM authority for the database, or DBCTRL authority for the database, any qualifier is valid.
2. If the privilege set does not include any of the authorities listed in item 1 above, the 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 all³² privileges needed to create the table.

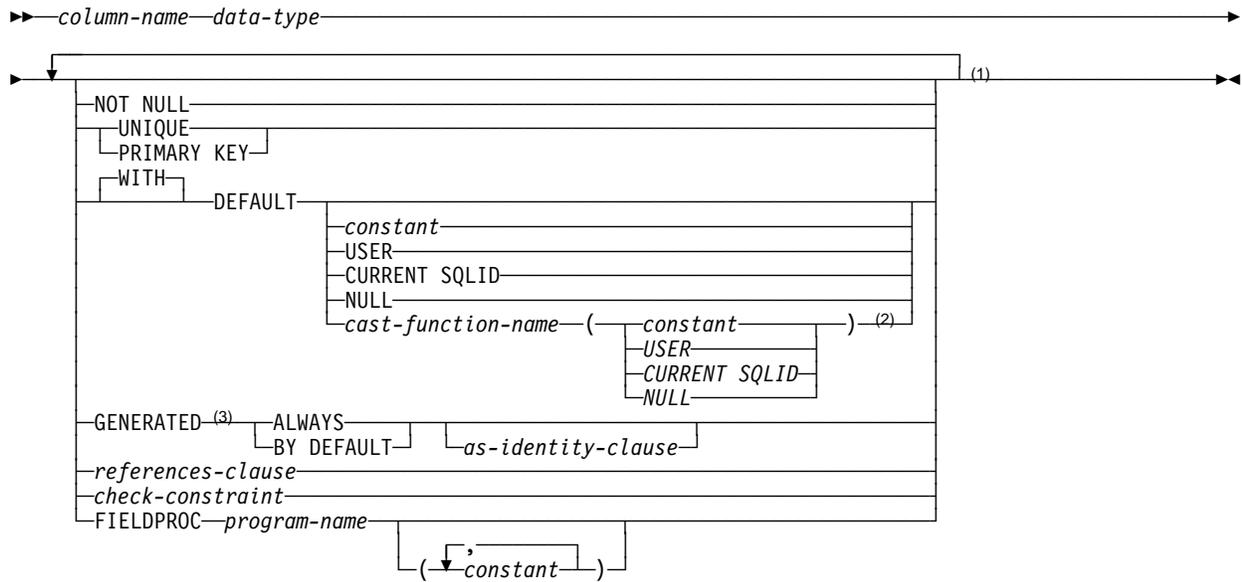
³² Exception: The CREATAB privilege is checked on the SQL authorization ID of the process.

Syntax



CREATE TABLE

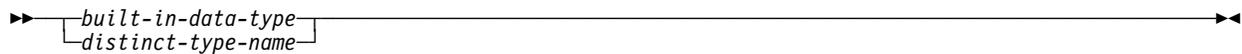
column-definition:



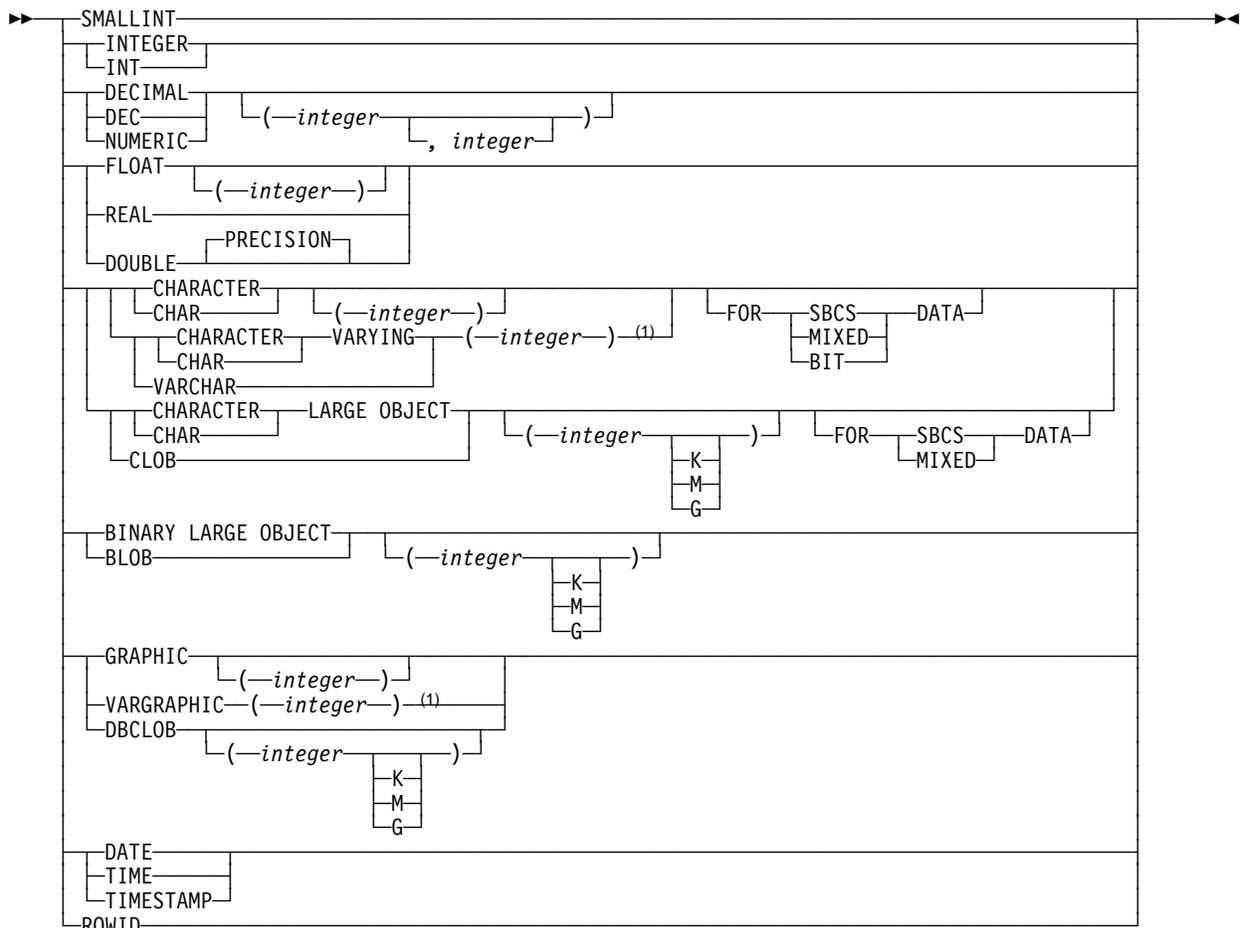
Notes:

- 1 The same clause must not be specified more than once.
- 2 This form of the DEFAULT value can only be used with columns that are defined as a distinct type.
- 3 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), or the column is to be an identity column.

data-type:



built-in-data-type:

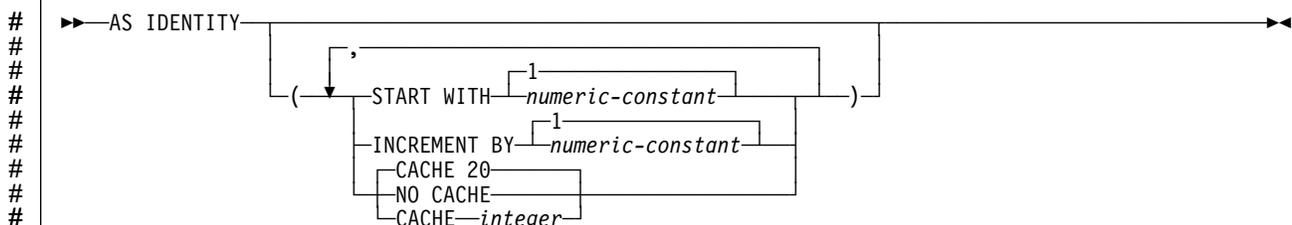


Note:

¹ Although the syntax of LONG VARCHAR and LONG VARGRAHPIC is supported, the alternative syntax of VARCHAR(*integer*) and VARGRAPHIC(*integer*), is preferred. VARCHAR(*integer*) and VARGRAPHIC(*integer*) are recommended 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.

To determine the maximum length of a column defined as LONG VARCHAR or LONG VARGRAPHIC, see Length of a LONG column on page 593.

as-identity-clause:

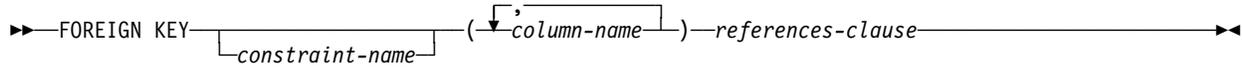


CREATE TABLE

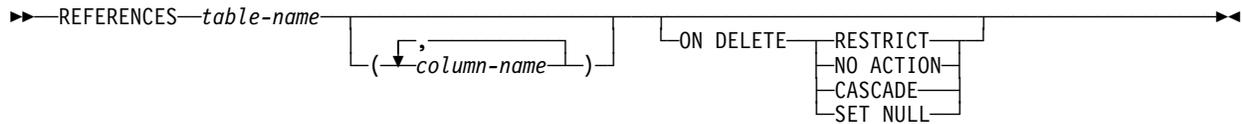
unique-constraint:



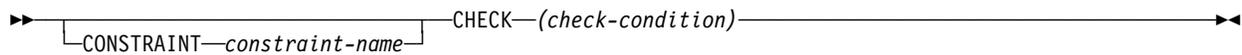
referential-constraint:



references-clause:



check-constraint:



Description

table-name

Names the table. The name must not identify a table, view, alias, or synonym that exists at the current server.

If 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.) Whether the name is two-part or three-part, the authorization ID that qualifies the name is the table's owner.

If the table name is unqualified and the statement is embedded in a program, the owner of the table is the authorization ID that serves as the implicit qualifier for unqualified object names. This is the authorization ID in the QUALIFIER operand when the plan or package was created or last rebound. If QUALIFIER was not used, the owner of the table is the owner of the package or plan.

If the table name is unqualified and the statement is dynamically prepared, the SQL authorization ID is the owner of the table.

The owner has all table privileges on the table (SELECT, UPDATE, and so on), and the authority to drop the table. All the owner's table privileges are grantable.

column-definition

Defines the attributes of a column.

column-name

Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one 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.

built-in-data-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. If you define the table with a LOB column (CLOB, DBCLOB, or BLOB), you must also define a ROWID column. For more information, see Creating a table with LOB columns on page 591.

INTEGER or **INT**

For a large integer.

SMALLINT

For a small integer.

FLOAT(*integer*)

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 also specify:

REAL	For single precision floating-point
DOUBLE	For double precision floating-point
DOUBLE PRECISION	For double precision floating-point
FLOAT	For double precision floating-point

DECIMAL(*integer,integer*) or **DEC**(*integer,integer*)

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 also specify:

DECIMAL (<i>integer</i>)	For DECIMAL(<i>integer,0</i>)
DECIMAL	For DECIMAL(5,0)

The word NUMERIC can be used in place of DECIMAL. For example, NUMERIC(8) is equivalent to DECIMAL(8). Unlike DECIMAL, NUMERIC has no allowable abbreviation.

³³ Columns with distinct types based on LOB or row ID types count as LOB or ROWID columns.

CHARACTER(*integer*) or CHAR(*integer*)

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 8 bytes. See Table 36 on page 592 to determine the maximum record size. An *integer* greater than 255 defines a long string column.

FOR *subtype* DATA

Specifies a subtype for a character string column, which is a column with a data type of CHAR, VARCHAR, LONG VARCHAR, or CLOB. Do not use the FOR 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.

BIT

Column holds BIT data. Do not specify BIT for a CLOB column.

If you do not specify the FOR clause, the column is defined with a default subtype. 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.

CLOB(*integer* [K|M|G]), CHAR LARGE OBJECT(*integer* [K|M|G]), or CHARACTER LARGE OBJECT(*integer* [K|M|G])

For a character large object (CLOB) string of maximum length *integer*, which can range from 1 to 2 147 483 647. A CLOB column has a varying-length and is a long string column regardless of its length.

If the length specification is omitted, a length of 1M bytes is assumed.

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.

BLOB (*integer* [K|M|G]), **BINARY LARGE OBJECT**(*integer* [K|M|G])

For a binary large object (BLOB) string of maximum length *integer*, which can range from 1 to 2 147 483 647. A BLOB column has a varying-length and is a long string column regardless of its length.

If the length specification is omitted, a length of 1M bytes is assumed.

The meaning of *integer* K|M|G is the same as for CLOB.

GRAPHIC(*integer*)

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. An integer longer than 127 defines a long string column.

DBCLOB(*integer* [K|M|G])

For a double-byte character large object (DBCLOB) string of maximum length *integer*. A DBCLOB column has a varying-length and is a long string column regardless of length.

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 and the maximum length is 1 073 741 823.

If the length specification is omitted, a length of 1M characters is assumed.

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.

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.

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 once 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 or ROWID 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. 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 on page 593.)

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 once 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 column.

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. 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 on page 593.)

DEFAULT

The default value assigned to the column in the absence of a value specified on INSERT or LOAD. Do not specify DEFAULT for a ROWID column or an identity column (a column that is defined AS IDENTITY); DB2 generates default values. 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
Fixed-length string	Blanks
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 only have the default value of a zero-length string as listed above or 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.

USER

Specifies the value of the USER special register at the time of INSERT or LOAD as the default value for the column. If USER 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 USER special register, which is 8 bytes.

CURRENT SQLID

Specifies the value of the SQL authorization ID of the process at the time of INSERT 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, which is 8 bytes.

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. You can specify a cast function only if the data type of the column is a distinct type and you have the EXECUTE privilege on the cast function.

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.

In a given column definition:

- DEFAULT and FIELDPROC cannot both be specified.
- NOT NULL and DEFAULT NULL cannot both be specified.
- DEFAULT cannot be specified for a ROWID column or an identity column.
- 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.

Table 35 on page 580 summarizes the effect of specifying the various combinations of the NOT NULL and DEFAULT clauses on the CREATE TABLE statement *column-description* clause.

Table 35. 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 INSERT 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.
	USER	The value of the USER special register at the time of INSERT or LOAD is used as the default value.
	CURRENT SQLID	The SQL authorization ID of the process at the time of INSERT 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.
	USER	The value of the 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

Indicates that DB2 generates values for the column. You must specify GENERATED if the column is to be considered an identity column (a column defined with the AS IDENTITY clause) or the data type of the column is a ROWID (or a distinct type that is based on a ROWID).

ALWAYS

Indicates that DB2 will always 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

Indicates that DB2 will generate a value for the column when a row is inserted into the table unless a 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 statement and the LOAD utility cannot be used to add rows to the table. If the value of special register

CURRENT RULES is 'STD' when the CREATE TABLE statement is processed, 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. Without a unique index, DB2 can guarantee unique values only among the set of system-generated values.

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, DECIMAL with a scale of zero, or a distinct type based on one of these types).

An identity column is implicitly NOT NULL.

START WITH *numeric-constant*

Specifies the first value 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. The default is 1.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value that is not 0, 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, the sequence of values for the identity column ascends. If the value is negative, the sequence of values descends.

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.

CACHE *integer*

Specifies the number of values of the identity column sequence that DB2 preallocates and keeps in memory. The minimum value that can be specified is 2, and the maximum is the largest value that can be represented as an integer. The default is 20.

During a system failure, all cached identity column values that are yet to be assigned are lost, and thus, will never be used. Therefore, the value specified for CACHE also represents the

CREATE TABLE

maximum number of values for the identity column that could
be lost during a system failure.

In a data sharing environment, each member gets its own
range of consecutive values to assign. For example, if CACHE
20 is specified, DB2A might get values 1-20 for a particular
sequence, and DB2B might get values 21-40. Therefore, if
transactions from different members generate values for the
same identity column, the values that are assigned might not
be in the order in which they are requested.

The minimum value is 2. The maximum is the largest value that
can be represented as an integer. The default is CACHE 20.

NO CACHE
Specifies that values for the identity column are not
preallocated.

In a data sharing environment, use NO CACHE if you need to
guarantee that the identity values are generated in the order in
which they are requested.

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 a *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 the *references-clause* in the definition of a LOB or ROWID
| column; a LOB or ROWID column cannot be a foreign key.

check-constraint

The *check-constraint* of a *column-definition* has the same effect as specifying a table check constraint in a separate ADD *check-constraint* clause. For conformance with the SQL standard, a table check constraint specified in the definition of column C should not reference any columns other than C.

| Do not specify a table check constraint in the definition of a LOB or ROWID
| 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 Appendix B (Volume 2) of *DB2 Administration Guide*. Field procedures can only be specified for short string columns that do not have a nonnull default value. For more information about string comparisons with field procedures, see "String comparisons" on page 94.

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.

_____ End of column-definition _____

_____ unique-constraint _____

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 or ROWID column, and the same column must not be identified more than once. The number of identified columns must not exceed 64, and the sum of their length attributes must not exceed 255.

The table is marked as unavailable until its primary index is explicitly created unless the CREATE TABLE statement is processed by the schema processor. 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 on page 593.)

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 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 and the sum of their length attributes must not exceed 255.

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 CREATE TABLE statement is processed by the schema processor. 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 on page 593.)

The total number of columns in *all* UNIQUE clauses in the CREATE TABLE statement is limited. If the limit is reached, you can still get the effect of the UNIQUE clause by using a unique index.

End of unique-constraint

referential-constraint

FOREIGN KEY *constraint-name (column-name,...)* **references-clause**

Each specification of the FOREIGN KEY clause defines a referential constraint with the specified name. A name is generated if *constraint-name* is not specified. The generated name is derived from the name of the first column of the foreign key in the same way that the name of an implicitly created table space is derived from the name of a table, except that the scope of uniqueness of *constraint-name* is the table. If specified, *constraint-name* must be different from the names of any referential or check constraints previously specified on the table.

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 or ROWID column, and the same column must not be identified more than once. 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.

End of referential-constraint

references-clause

REFERENCES *table-name (column-name,...)*

The table name specified after REFERENCES must identify a table that exists at the current server³⁴, but it must not identify a catalog 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 table³⁴).

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 or a ROWID column. The same column must not be identified more than once.

The list of column names must be identical to the list of column names in a unique index (UNIQUERULE in SYSINDEXES will be R, P, C, or U). The column names must be specified in the **same order** as in the unique index on T2.

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.

³⁴ 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 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 integrity” on page 23.

SET NULL must not be specified unless some column of the foreign key allows null values. The default value for the rule depends on the value of the CURRENT RULES special register when the CREATE TABLE statement is processed. If the value of the register is 'DB2', the delete rule defaults to RESTRICT; if the value is 'STD', the delete rule defaults to NO ACTION.

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let p denote such a row of T2. Then:

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of p in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of p in T1 is set to null.

Let T3 denote a table identified in another FOREIGN KEY clause (if any) of the CREATE TABLE statement. The delete rules of the relationships involving T2 and T3 must be the same and must not be SET NULL if:

- T2 and T3 are the same table.
- T2 is a descendent of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendents of the same table and the deletion of rows from that table cascades to both T2 and T3.

End of references-clause

check-constraint

CONSTRAINT *constraint-name*

Names the table check constraint. The constraint name must be different from the names of any referential or check 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 table check constraint.

CHECK (*check-condition*)

Defines a table 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 or ROWID columns.
- 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 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
 - Quantified predicates
 - EXISTS predicates
- If a check-condition refers to a long string column, 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
- 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.

End of check-constraint

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, and the privilege set must implicitly or explicitly include the SELECT privilege on the identified table or view. An identified table must not be an auxiliary table. An identified view must not include a column that is considered to be a ROWID column or an identity column. (For more information, see “Notes” on page 590.)

The use of LIKE is an implicit definition of n columns, where n is the number of columns in the identified table or view. The implicit definition includes all attributes of the n columns as they are described in SYSCOLUMNS with these 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 table is identified in the LIKE clause and a column in the table 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 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. Also, no column in the new table has a field procedure because the catalog descriptions of view columns do not include field procedures.

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.

#

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that a column of the new table inherits all of the identity attributes of the identity column. If the table identified by LIKE does not have an identity column, the INCLUDING IDENTITY clause is ignored. If the LIKE clause identifies a view, INCLUDING IDENTITY COLUMN ATTRIBUTES cannot be specified.

WITH RESTRICT ON DROP

Indicates that the table cannot be dropped. Also, the database and table space that contain the table cannot be dropped.

IN *database-name.table-space-name* or IN DATABASE *database-name*

Names the database and table space in which the table is created. Both forms are optional; the default is IN DATABASE DSNDB04.

You can name a database (with *database-name*), a table space (with *table-space-name*), or both. If you name a database, it must be described in the current server's catalog, and must not be DSNDB06 or a work file database.

If you use IN DATABASE, either explicitly or by default, a table space is implicitly created in *database-name*. The name of the table space is derived from the table name. Its other attributes are those it would have if it were created by a CREATE TABLESPACE statement with all optional clauses omitted.

If you name a table space, it must not be one that was created implicitly, be a partitioned table space that already contains a table, or be a LOB table space. If you name a partitioned table space, you cannot load or use the table until its partitioned index is created.

If you name both a database and a table space, the table space must belong to the database you name. If you name only a table space, it must belong to database DSNDB04.

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 name a table space, you must have SYSADM or SYSCTRL authority, DBADM authority for the database, or the USE privilege for the table space.

EDITPROC *program-name*

Designates *program-name* as the edit routine for the table. The edit routine, which must be provided by the current server's site, is invoked during the execution of LOAD, INSERT, UPDATE, 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. For information on writing an EDITPROC exit routine, see Appendix B (Volume 2) of *DB2 Administration Guide*.

You must not specify an edit routine for a table with a LOB, ROWID, or identity
column.

If you omit EDITPROC, the table has no edit procedure.

VALIDPROC *program-name*

Designates *program-name* as the validation exit routine for the table. Writing a validation exit routine is described in Appendix B (Volume 2) of *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 represented by any LOB columns in the table are not passed. 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 Section 3 (Volume 1) of *DB2 Administration Guide*.

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 performed by each unit of work. 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 each unit of work of 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 altered with an ALTER TABLE statement, the ALTER TABLE statement is audited only if AUDIT CHANGES or AUDIT ALL is specified and the appropriate audit trace class is active.

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';
```

CREATE TABLE

Here, *nnn* is the table name and *ccc* is the table's creator.

DATA CAPTURE

Specifies whether the logging of SQL INSERT, UPDATE, and DELETE operations on the table is augmented by additional information. For guidance on intended uses of the expanded log records, see:

- The description of data propagation to IMS in *DataPropagator NonRelational MVS/ESA Administration Guide*
- 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 Appendix C (Volume 2) of *DB2 Administration Guide*

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 columns is not available.

CCSID *encoding-scheme*

Specifies the encoding scheme for string data stored in the table. If the IN clause is specified, the value must agree with the encoding scheme that is already in use for the table space or database specified in the IN clause. The specific CCSIDs for SBCS, BIT, and MIXED 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 and the table space that DB2 implicitly creates. 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 and EBCDIC.

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 value of field DEF ENCODING SCHEME on installation panel DSNTIPF is used.

When you use the LIKE clause with the CREATE TABLE statement, the encoding scheme of the table being copied is not used.

Notes

Table design: Designing tables is part of the process of database design. For information on design, see Section 2 (Volume 1) of *DB2 Administration Guide*.

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.

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.

Distinct type columns based on LOB and ROWID 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) and ROWID columns apply to distinct type columns that are sourced on LOBs and row IDs. For example, a table cannot have both a ROWID column and a column with a distinct type that is sourced on a row ID.

Creating a table with LOB columns: If you create a base table with a LOB column (CLOB, DBCLOB, or BLOB), you must also define a ROWID column for the table. 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 value of special register CURRENT RULES is 'STD' when the CREATE STATEMENT is processed, DB2 implicitly creates the LOB table space, auxiliary table, and index on the auxiliary table for each LOB column in the base table. 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 size of the LOB column is greater than 1 GB, the LOG option for the LOB table space is LOG NO.

CREATE TABLE

Utility REPORT TABLESPACESET identifies the LOB table spaces that DB2 implicitly created.

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 36. The 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.

Table 36. 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.

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 37 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 37 (Page 1 of 2). Byte Counts of Columns by Data Type

Data Type	Byte Count
INTEGER	4
SMALLINT	2
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($p/2$)+1, where <i>p</i> is the precision
CHAR(<i>n</i>)	<i>n</i>
VARCHAR(<i>n</i>)	<i>n</i> +2 (For LONG VARCHAR, see Byte count of a LONG VARCHAR or LONG VARGRAPHIC column on page 593.)
CLOB	6
BLOB	6
GRAPHIC(<i>n</i>)	2 <i>n</i>
VARGRAPHIC(<i>n</i>)	2 <i>n</i> +2 (For LONG VARGRAPHIC, see Byte count of a LONG VARCHAR or LONG VARGRAPHIC column on page 593.)
DBCLOB	6
DATE	4
TIME	3
TIMESTAMP	10
ROWID	19

Table 37 (Page 2 of 2). Byte Counts of Columns by Data Type

Data Type	Byte Count
distinct type	The length of the source data type upon which the distinct type was based

Byte count of a LONG VARCHAR or LONG VARGRAPHIC column: To calculate the byte count, let:

m be the maximum row size (8 less than the maximum record size)

i be the sum of the byte counts of all columns in the table that are not LONG VARCHAR or LONG VARGRAPHIC

j be the number of LONG VARCHAR and LONG VARGRAPHIC columns in the table

k be the number of LONG VARCHAR and LONG VARGRAPHIC columns that allow nulls.

The count is $2 * (\text{INTEGER}((\text{INTEGER}((m-i-k)/j))/2))$.

Length of a LONG column: To find the character count:

1. Find the byte count from Byte count of a LONG VARCHAR or LONG VARGRAPHIC column on page 593.
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.

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, DB2 implicitly creates the unique indexes used to enforce the uniqueness of the primary or unique keys. 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.

For more information about the schema processor, see Section 2 (Volume 1) of *DB2 Administration Guide*.

Creating a table like a view: If the LIKE clause is specified and the definition of the table is being based on a view, the view must not include a ROWID column or an identity column. A view column is considered to be an identity column if the corresponding column of the table or view indirectly or directly maps to the name of an identity column in a base table with these exceptions:

- The select-list of the view definition identifies the same identity column more than once.

#

CREATE TABLE

- The select-list of the view definition references multiple identity columns and thus involves a join.
- A column in the view definition includes an expression that refers to an identity column.

Using an identity column: When a table has an identity column, DB2 can automatically generate unique, 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 and guarantees as unique. 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 sequence of 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.

Sometimes you may need to change the attributes of an identity column. For example, if you had defined an identity column with a data type of SMALLINT and then run out of assignable values, you need to redefine the column as INTEGER. To change the attributes of an identity column, you unload the data from the table, drop the table, recreate the table, and reload the data.

But when you recreate the table, you must specify GENERATED BY DEFAULT and a new START WITH value for the identity column. Using GENERATED BY DEFAULT allows LOAD to reload the previously existing identity column values. You cannot use GENERATED ALWAYS in this case, but not using it is not a problem since DB2 always generates a value if a column value is not provided during insertion of an identity column defined as GENERATED BY DEFAULT. The new START WITH value should be the next value in the sequence from where the original sequence of values left off; this value is the next value that DB2 would generate first.

Using tables with different encoding schemes: The CCSID clause determines whether the data for a table is encoded in ASCII or EBCDIC. All created tables that are referenced in an SQL statement must have the same encoding scheme—the tables must be either all ASCII or all EBCDIC. Once the data is created with the CREATE TABLE statement, you cannot mix the encoding schemes.

Dropping a table in a partitioned table space: You can only drop a table in a partitioned table space by using the DROP TABLESPACE statement.

Examples

Example 1: Create a table named DSN8610.DEPT in the table space DSN8S61D of the database DSN8D61A. Name the table's five columns DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, and LOCATION, allowing only MGRNO to contain nulls, and designating DEPTNO as the only column in the table's 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 DSN8610.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 DSN8D61A.DSN8S61D;
```

Example 2: Create a table named DSN8610.PROJ in an implicitly created table space of the database DSN8D61A. Assign the table a validation procedure named DSN8EAPR.

```
CREATE TABLE DSN8610.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 DSN8D61A
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 DSN8D61A.DSN8S61D;
```

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 DSN8D61A.DSN8S61E. Let DB2 always generate the values for the ROWID column.

CREATE TABLE

```
| CREATE TABLE DSN8610.EMP_PHOTO_RESUME  
| (EMPNO CHAR(6) NOT NULL,  
# EMP_ROWID ROWID NOT NULL GENERATED ALWAYS,  
| EMP_PHOTO BLOBL(110K),  
| RESUME CLOB(5K)),  
| PRIMARY KEY EMPNO  
| IN DSN8D61A.DSN8S61E  
| CCSID EBCDIC;
```

```
# Example 5: Create an EMPLOYEE table with an identity column named EMP_NO.  
# 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 DSN8D61A.DSN8S61D;
```

CREATE TABLESPACE

The CREATE TABLESPACE statement defines a simple, segmented, or partitioned 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:

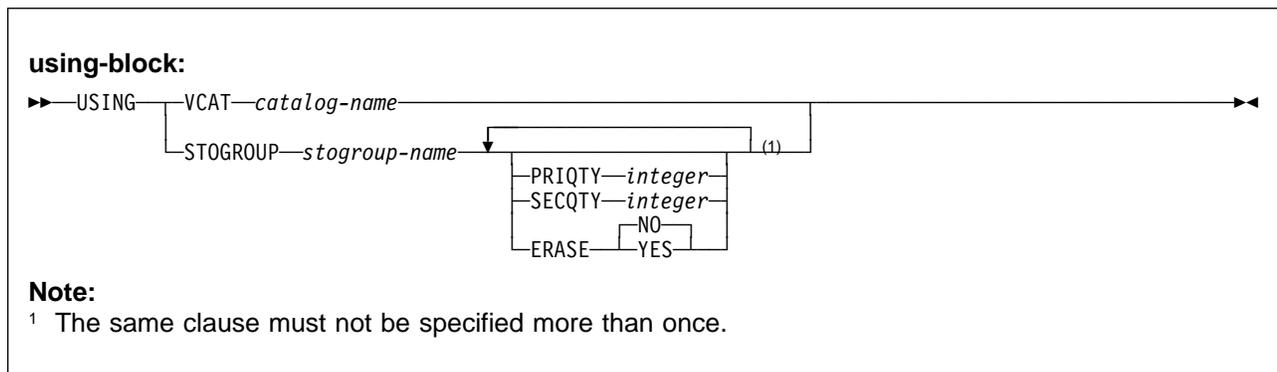
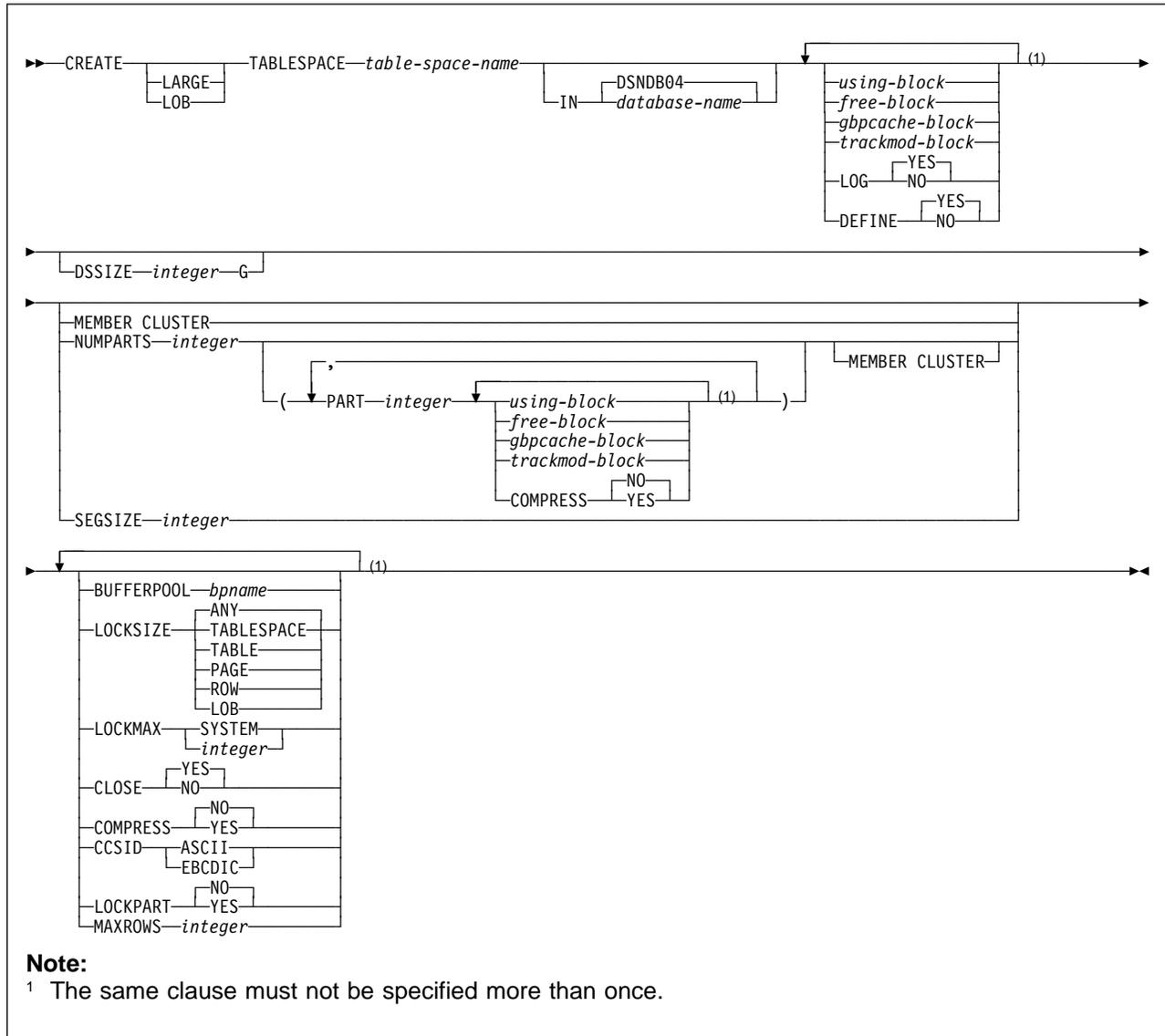
- The CREATETS privilege for the database
- DBADM, DBCTRL, or DBMAINT authority for the database
- SYSADM or SYSCTRL authority

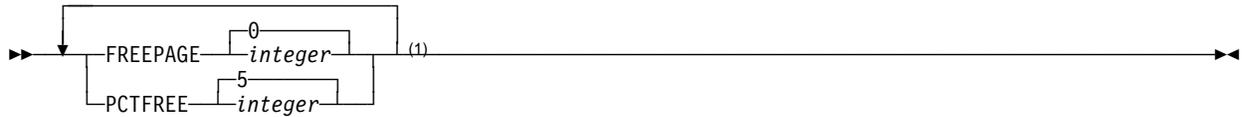
Additional privileges might be required, as explained in the description of the BUFFERPOOL and USING STOGROUP clauses.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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.

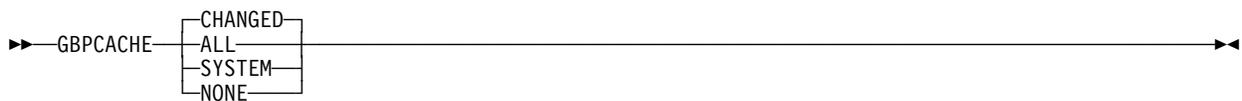
CREATE TABLESPACE

Syntax



free-block:**Note:**

¹ The same clause must not be specified more than once.

gbpcache-block:**trackmod-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 OS/390 prior to Version 6. Do not specify LARGE if LOB or DSSIZE is specified.

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.

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 a TEMP
 # database (a database that is defined AS TEMP). PUBLIC implicitly receives the
 # USE privilege (without GRANT authority) on any table space created in the
 # TEMP database. This implicit privilege is not recorded in the DB2 catalog, and
 # it cannot be revoked.

IN *database-name*

Identifies the database in which the table space is created. The name must identify a database that exists at the current server. DSNDB06 must not be specified for any type of table space, and a work file database must not be specified for a LOB table space. (If a work file database is specified, it must be in the stopped state.) If the table space is for declared temporary tables, a TEMP database (a database that is defined with AS TEMP) must be specified. The default is DSNDB04.

using-block

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*. Because *catalog-name* is a short identifier, an alias must be used if the catalog name is longer than eight characters.

Conventions for table space data set names are given in Section 2 (Volume 1) of *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.

STOGROUP *stogroup-name*

Specifies that DB2 will define and manage the data sets for the table space. Each data set will be defined on a volume of the identified storage group. The values specified (or the defaults) for PRIQTY and SECQTY determine the primary and secondary allocations for the data set. The storage group supplies the name of a volume for the data set and the first-level qualifier for the data set name. The first-level qualifier is also the name of, or an alias 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 MVS 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. The primary space allocation is at least *n* kilobytes, where *n* is the value of *integer* with the following exceptions:

- If PRIQTY *integer* is specified:
 - 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.
- If PRIQTY is omitted, *n* is 12, 24, 48, or 96 for 4KB, 8KB, 16KB, and 32KB page sizes, respectively.

For LOB table spaces, the exceptions are:

- If PRIQTY *integer* is specified:
 - 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 PRIQTY is omitted, *n* is 200, 400, 800, or 1600 for 4KB, 8KB, 16KB, and 32KB page sizes, respectively.

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 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. The secondary space allocation is at least *n* kilobytes, where *n* is the value of *integer* with the following exceptions:

- If SECQTY *integer* is specified and *integer* is greater than 4194304, *n* is 4194304. A value of 0 for *integer* indicates that no data set can be extended.
- If SECQTY and PRIQTY are omitted:
 - For 4KB page sizes, *n* is 12.
 - For 8KB page sizes, *n* is 24.
 - For 16KB page sizes, *n* is 48.
 - For 32KB page sizes, *n* is 96.
- If SECQTY is omitted and PRIQTY is specified, *n* is either 10% of PRIQTY or 3 times the page size of the table space, whichever is larger.

For LOB table spaces the exceptions are:

- If SECQTY *integer* is specified:
 - 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.
- If SECQTY is omitted, *n* is either 10% of PRIQTY or 50 times the page size of the table space, whichever is larger.

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 *DFSMS/MVS: Access Method Services for the Integrated Catalog*.

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 PART clause for the partition
- A USING clause that is not in any PART 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 Section 2 (Volume 1) of *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.

CREATE TABLESPACE

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 PART clause. If that USING STOGROUP clause is not specified, the defaults are those specified in the description of PRIQTY, SECQTY, and ERASE.

_____ End of using-block _____

_____ free-block _____

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 or a TEMP database.

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 or a TEMP database.

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 PART clause for that partition
- The values given in a *free-block* that is not in any PART clause
- The default values are FREEPAGE 0 and PCTFREE 5.

_____ End of free-block _____

_____ gbpcache-block _____

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 or TEMP database, but it is ignored. Do not

|
|
#

|
#

#

specify GBPCAHCE for a table space in a work file or TEMP 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.

Hiperpools are not used for indexes or partitions that are defined with GBPCACHE ALL.

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 PART clause for that partition. Do not use more than one *gbpcache-block* in any PART clause.
2. The value given in a *gbpcache-block* that is not in any PART clause.
3. The default value CHANGED.

End of *gbpcache-block*

trackmod-block

```

|
|           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. For
#           a table space in a TEMP database, DB2 uses TRACKMOD NO regardless of
#           the value specified.
|
|           YES
|           DB2 tracks changed pages in the space map pages to improve the
|           performance of incremental image copy. YES is the default unless the table
|           space is in a TEMP database.
|
|           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 PART clause for that partition.
|           2. The value given in a trackmod-block that is not in any PART clause.
|           3. The default value YES.
|
|           _____ End of trackmod-block _____

```

```

|
|           LOG
|           Specifies whether changes to a LOB column in the table space are to be
|           written to the log. You can use the LOG clause only for a LOB table space.
|
|           YES
|           Indicates that changes to a LOB column are to be written to the log. You
|           cannot use YES if the auxiliary table in the table space stores a LOB
|           column that is greater than 1 gigabyte in length.
|
|           YES is the default.
|
|           NO
|           Indicates that changes to a LOB column are not to be written to the log.
|
|           LOG NO has no effect on a commit or rollback operation; the consistency
|           of the database is maintained regardless of whether the LOB value is
|           logged. All committed changes and changes that are rolled back reflect the
|           expected results.
|
|           Even when LOG NO is specified, changes to system pages and to the
|           auxiliary index are logged. During the log apply operation of the RECOVER
#           utility, LPL recovery, or GPB recovery, all LOB values that were not logged
#           are marked invalid and cannot be accessed by a SELECT or FETCH
#           statement. Invalid LOB values can be updated or deleted.

```

```

#           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.

```

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 or a
 # TEMP database; otherwise, an error occurs. 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.

DSSIZE integer G

A value in gigabytes that indicates the maximum size for each partition or, for LOB table spaces, each data set. If you specify DSSIZE, you must also specify Numparts or LOB.

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.

For all table spaces except LOB table spaces, if DSSIZE (or LARGE) is omitted, the default for the maximum size of each partition depends on the value of Numparts:

If Numparts is ...	Maximum partition size is...
1 to 16	4 GB
17 to 32	2 GB
33 to 64	1 GB
65 to 254	4 GB

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 4 GB. The maximum number of data sets is 254.

When you give a tablespace a DSSIZE value, you also give the same size to
 # all the indexes that point to that tablespace.

MEMBER CLUSTER

Specifies that data inserted by the INSERT statement is not clustered by the implicit clustering index (the first index) or the explicit clustering index. Instead,

CREATE TABLESPACE

DB2 chooses where to locate the data in the table space based on available space.

|
Do not specify MEMBER CLUSTER for a LOB table space , or a table space in a work file database or a TEMP database.

NUMPARTS *integer*

Indicates that the table space is partitioned.

|
| *integer* is the number of partitions, and can range from 1 to 254 inclusive.
| NUMPARTS must be specified if DSSIZE is specified and LOB is omitted, or
| LARGE is specified.

|
| The maximum size of each partition depends on the value specified for DSSIZE
| or LARGE. If DSSIZE or LARGE is not specified, the number of partitions
| specified determines the maximum size of each partition. For a summary of the
| values for the maximum size, see the description of DSSIZE on page 607.

|
If you omit NUMPARTS, the table space is not partitioned and initially occupies one data set. Do not specify NUMPARTS for a LOB table space, or a table space in a work file database or a TEMP database.

PART *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 PART 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.

See "Naming conventions" on page 50 for more details about *bpname*. See Chapter 2 of *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 or a TEMP 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 nonsegmented, 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.

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 table space in a TEMP 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, ROW, or LOB	0

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.

CREATE TABLESPACE

YES
Eligible for closing before CLOSE NO data sets. This is the default unless
the table space is in a TEMP database.

NO
Eligible for closing after all eligible CLOSE YES data sets are closed.

For a table space in a TEMP 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 TEMP 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 PART clause for the partition
- The value specified in the COMPRESS clause that is not in any PART clause
- An implicit COMPRESS NO by default.

See Section 2 (Volume 1) of *DB2 Administration 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.

NO

Specifies no data compression for the table space or partition.

SEGSIZE *integer*

Indicates that the table space will be segmented. *integer* specifies how many pages are to be assigned to each segment. *integer* must be a multiple of 4 such that $4 \leq integer \leq 64$. If the SEGSIZE clause is not specified, the table space is not segmented.

SEGSIZE must be specified for a table space in a TEMP database because the
table space must be segmented. Do not specify SEGSIZE for a LOB table
space or a table space in work file database; neither can be segmented.

A segmented table space cannot be partitioned. Therefore, do not specify Numparts if you specify SEGSIZE.

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. If the database in which the table space is to reside is already defined as EBCDIC, the EBCDIC CCSIDs associated with that database are used. Otherwise, the default EBCDIC CCSIDs of the server are used.

Usually, each encoding scheme requires only a single CCSID. Additional CCSIDs are needed when mixed or graphic data is used.

All data stored within a table space must use the same encoding scheme unless the table space is in a TEMP database.

Do not specify CCSID for a LOB table space or a table space in a TEMP
database. The encoding scheme for a LOB table space is inherited from the
base table space. A table space in a TEMP database does not have an
associated encoding scheme because the table space can contain declared
temporary tables with a mixture of encoding schemes.

LOCKPART

Indicates whether selective partition locking (SPL) is to be used when locking a partitioned table space.

YES If all the conditions that are required for SPL are met, specifies that only the partitions accessed will be locked. If all the conditions that are required for SPL are not met, every partition of the table space is locked. LOCKPART YES is not allowed with LOCKSIZE TABLESPACE.

NO Specifies that selective partition locking is not used. The table space is locked with a single lock on the last partition. This has the effect of locking all partitions in the table space.

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, LOAD, and REORG. For LOAD and REORG (which do not apply for a table space in the TEMP 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.

Notes

Simple table spaces: If neither LOB, NUMPARTS, nor SEGSIZE are specified, the table space that is created is a simple table space. See Section 2 (Volume 1) of *DB2 Administration Guide* for a discussion of types of table spaces.

Table spaces in a work file database: The following restrictions apply to table spaces created in a work file database:

- They can be created only when the database is explicitly stopped by the STOP DATABASE command without the SPACENAM option.
- 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.

CREATE TABLESPACE

- The following clauses are not allowed:

|

DEFINE NO	LOB	NUMPARTS
FREEPAGE	LOG	PCTFREE
GBPCACHE	LOCKSIZE	SEGSIZE

Table spaces in a TEMP database (table spaces for declared temporary tables): Declared temporary tables must reside in segmented table spaces in a database that is defined AS TEMP (the TEMP database). At least one segmented table space must exist in the TEMP database before a declared temporary table can be defined and used. DB2 does not implicitly create a table space for declared temporary tables. A table space for declared temporary tables can be shared. You cannot choose which table space in a TEMP database is used for a specific declared temporary table. Therefore, multiple application processes can use the same table space for their declared temporary tables.

When you create table spaces for in the TEMP database, it is recommended that you give them all 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 declared temporary tables in all application processes.

When you create a table space in a TEMP database, the following clauses are not allowed:

CCSID	GBPCACHE	LOG
COMPRESS	LARGE	MEMBER CLUSTER
DEFINE NO	LOB	NUMPARTS
DSSIZE	LOCKSIZE	PCTFEE
FREEPAGE	LOCKPART	TRACKMOD

|
|

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	NUMPARTS	TRACKMOD
LOCKSIZE TABLE		

Converting a partitioned table space to be larger: To increase the size of a partitioned table space so that it can hold more data, take 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. Create the table space, specifying an appropriate value for the DSSIZE clause. Also redefine the partitioning index (with different key range values), the table, and the nonclustering indexes.
4. Recreate views and synonyms. Reestablish appropriate authorizations.
5. Load data into the new table.
6. Rebind the plans and packages that changed.

Examples

Example 1: Create table space DSN8S61D in database DSN8D61A. Let DB2 define the data sets, using storage group DSN8G610. 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 DSN8S61D
  IN DSN8D61A
  USING STOGROUP DSN8G610
  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 DSN8D61A 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 DSN8G610. 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 may close the CLOSE NO data sets when the DSMAX is reached.

CREATE TABLESPACE

```
CREATE TABLESPACE SALESX
  IN DSN8D61A
  USING STOGROUP DSN8G610
  PRIQTY 4000
  SECQTY 130
  ERASE NO
  NUMPARTS 82
  (PART 80
    COMPRESS YES,
  PART 81
    COMPRESS YES,
  PART 82
    ERASE YES
    COMPRESS 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 DSN8D61A for the
| auxiliary table that will hold the BLOB data.

|
| Let DB2 define the data sets for the table space, using storage group DSN8G610.
| 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.

```
CREATE LOB TABLESPACE PHOTOLTS
  IN DSN8D61A
  USING STOGROUP DSN8G610
  PRIQTY 3200
  SECQTY 1600
  LOCKSIZE LOB
  BUFFERPOOL BP16K0
  GBPCACHE SYSTEM
  LOG NO
  CLOSE NO;
```

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 all of the following:

- Either of these privileges:
 - The CREATEIN privilege for the schema or all schemas
 - SYSADM or SYSCTRL authority
- The TRIGGER privilege on the table. The privilege set 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
 - SYSADM or SYSCTRL authority
- 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 EXECUTE privilege on any user-defined function or stored procedure that is invoked in the triggered action
- The necessary privileges to invoke the triggered SQL statements in the triggered action

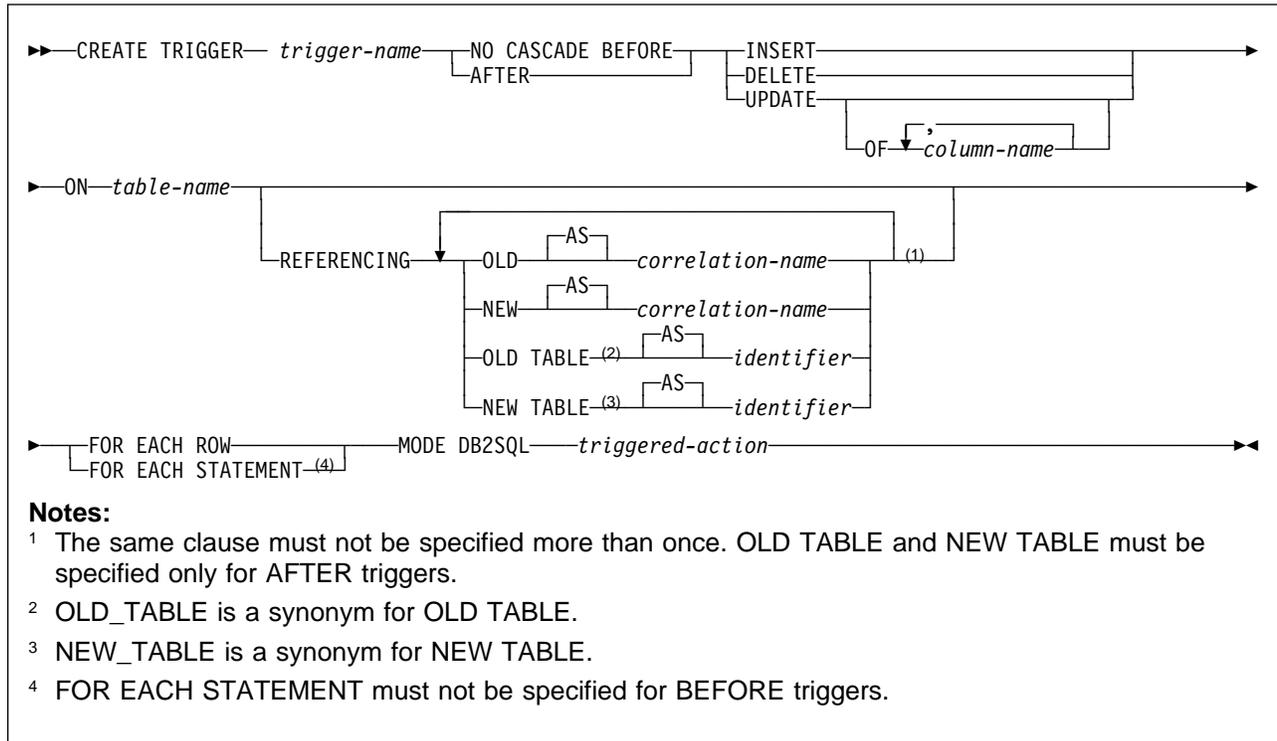
Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package.

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 trigger name can include a schema name (a qualifier). However, if the specified name includes a schema name that 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 CREATEIN privilege on the schema.

CREATE TRIGGER

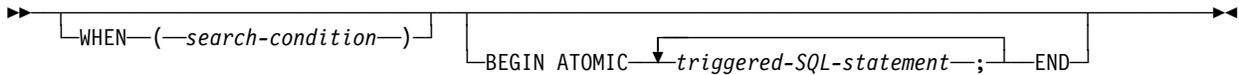
Syntax



Notes:

- 1 The same clause must not be specified more than once. OLD TABLE and NEW TABLE must be specified only for AFTER triggers.
- 2 OLD_TABLE is a synonym for OLD TABLE.
- 3 NEW_TABLE is a synonym for NEW TABLE.
- 4 FOR EACH STATEMENT must not be specified for BEFORE triggers.

triggered-action



Description

trigger-name

Names the trigger. The name is implicitly or explicitly qualified by a schema. 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.

- The unqualified form of *trigger-name* is a short SQL identifier. The unqualified name is implicitly qualified with a schema name according to the following rules:

If the statement is embedded in a program, the schema name of the trigger 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 schema name of the trigger is the owner of the package or plan.

If the statement is dynamically prepared, the schema name of the trigger is the SQL authorization ID of the process.

- The qualified form of *trigger-name* is a short SQL identifier (the schema name) followed by a period and a short SQL identifier. The schema name must not begin with 'SYS' unless the name is 'SYSADM'. The schema name that qualifies the trigger name is the trigger's owner.

The owner of the trigger is determined by how the CREATE TRIGGER 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.

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 triggering 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.

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 triggering table.

INSERT

Specifies that the trigger is an insert trigger. DB2 executes the triggered action whenever there is an insert operation on the triggering table. However, if the insert trigger is defined on PLAN_TABLE, DSN_STATEMENT_TABLE, or DSN_FUNCTION_TABLE, and the insert operation was caused by DB2 adding a row to the table, the triggered action is not executed.

DELETE

Specifies that the trigger is a delete trigger. DB2 executes the triggered action whenever there is a delete operation on the triggering table.

UPDATE

Specifies that the trigger is an update trigger. DB2 executes the triggered action whenever there is an update operation on the triggering table.

If you do not specify a list of column names, an update operation on any column of the triggering 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.

ON *table-name*

Identifies the subject table with which the trigger is associated. The name must identify a base table at the current server. It must not identify a temporary table, an auxiliary table, an alias, a synonym, or a catalog table.

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. A table name identifies the complete set of modified rows.

Each row that is modified by the triggering operation is available to the triggered action by using column names that are qualified with correlation names that are specified as follows:

OLD AS *correlation-name*

Specifies the correlation name that identifies the state of the row prior to the triggering SQL operation.

NEW AS *correlation-name*

Specifies the correlation name that identifies the state of 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 is modified by the triggering operation is available to the triggered action by using a temporary table name that is specified as follows:

OLD TABLE AS *identifier*

Specifies the name of a temporary table that identifies the state of the complete set of rows that are modified rows by the triggering SQL operation prior to any actual changes. *identifier* is a long SQL identifier.

NEW TABLE AS *identifier*

Specifies the name of a temporary table that identifies the state of 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. *identifier* is a long SQL identifier.

At most, the trigger definition can include two correlation names (OLD and NEW) and two table names (OLD TABLE and NEW TABLE). All the names must be unique from one another.

Table 38 on page 619 summarizes the allowable combinations of transition variables and transition tables that you can specify for the various trigger types. OLD and OLD TABLE are valid only if the triggering SQL operation is a delete or an update. For a delete operation, OLD captures the values of the columns in the deleted row, and OLD TABLE captures the values in the set of deleted rows. For an update operation, OLD captures the values of the columns of a row before the update, and OLD TABLE captures the values in the set of updated rows.

NEW and NEW TABLE are valid only if the triggering SQL operation is an insert or an update. For both operations, NEW captures the values of the columns in the inserted or updated row. For before triggers, the values of the updated rows include the changes from any SET statement in the triggered action if the trigger is a before trigger.

OLD and NEW are valid only if you also specify FOR EACH ROW, and OLD TABLE and NEW TABLE are valid only if you specify AFTER.

Table 38. Allowable combinations of attributes in a trigger definition

Activation time	Triggering SQL operation	Transition variables	Transition tables	Granularity
BEFORE	DELETE	OLD		FOR EACH ROW
	INSERT	NEW		
	UPDATE	OLD, NEW		
AFTER	DELETE	OLD	OLD TABLE	FOR EACH ROW
	INSERT	NEW	NEW TABLE	
	UPDATE	OLD, NEW	OLD TABLE, NEW TABLE	
	DELETE		OLD TABLE	FOR EACH STATEMENT
	INSERT		NEW TABLE	
	UPDATE		OLD TABLE, NEW TABLE	

A transition variable that has a character data type inherits the subtype and CCSID of the column of the triggering table. During the execution of the triggered action, the transition variables are treated like host variables. Therefore, character conversion might occur.

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 triggering table, it does inherit the triggering table's encoding scheme and field procedures.

The scope of the transition variables and transition tables is the triggered action. Do not refer to their names outside of 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 once for the triggering SQL operation. Even if the triggering SQL operation does not modify any rows, the triggered action is executed once. Do not specify FOR EACH STATEMENT for a before trigger.

MODE DB2SQL

Specifies the mode of the trigger. Currently, DB2 supports only MODE DB2SQL.

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 by an optional condition that controls whether the statements are executed.

WHEN (*search-condition*)

Specifies a condition that evaluates to true, false, or unknown. The condition for a before trigger must not include a subselect that references the triggering table.

#

CREATE TRIGGER

The triggered SQL statements are executed only if the search condition evaluates to true, or if WHEN is omitted.

BEGIN ATOMIC *triggered-SQL-statement*;... **END**

Specifies the SQL statements that are to be executed for the triggered action. The statements are executed in the order in which you specify them. The keywords BEGIN ATOMIC and END are required only if you specify more than one SQL statement. In which case, you must enclose the SQL statements in these keywords and end each statement with a semicolon (;).

SQL processor programs, such as SPUFI 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*.

Table 39 shows the list of allowable SQL statements, which differs depending on whether the trigger is being defined as BEFORE or AFTER. An 'X' in the table indicates that the statement is valid.

Table 39. Allowable SQL statements

SQL statement	Trigger activation time	
	BEFORE	AFTER
fullselect	X	X
CALL	X	X
SIGNAL SQLSTATE	X	X
VALUES	X	X
SET transition variable	X	
INSERT		X
DELETE (searched)		X
UPDATE (searched)		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 may 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

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.

The user executing the triggering SQL operation does not need authority to execute a trigger package. The trigger package does not need to be in the package list for the plan that is associated with the program that contains the SQL statement.

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 triggering 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 attributes:

- ACTION(ADD)
- CURRENTDATA(YES)
- DBPROTOCOL(DRDA)
- DEGREE(1)
- DYNAMICRULES(BIND)

CREATE TRIGGER

- ENABLE(*)
- EXPLAIN(NO)
- FLAG(I)
- ISOLATION(CS)
- NOREOPT(VARS) and NODEFER(PREPARE)
- OWNER(authorization ID)
- QUERYOPT(1)
- PATH(path)
- RELEASE(COMMIT)
- 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.

Activating a trigger: Only the SQL statements INSERT, DELETE, or UPDATE, or an update or delete operation that occurs as the result of a referential constraint with ON DELETE SET NULL or ON DELETE CASCADE can activate a trigger. Loading a table with the LOAD utility does not activate any triggers that are defined for the table.

Simultaneously activated triggers: Multiple triggers that have the same triggering SQL operation and activation time (BEFORE or AFTER) 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; and so on.

Adding columns to subject tables or tables that the triggered action references: If a column is added to a table for which a trigger is defined (the subject table), 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 SQL statements in the triggered action refer to the subject table, the new column is not accessible to the SQL statements until the trigger package is rebound.
- The transition tables contain the new column. 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 to which the SQL statements in the triggered action refers, the new column is not accessible to the SQL statements until the trigger package is rebound.

Adding triggers to enforce constraints: Creating a trigger on a table that already has rows does 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.

Defining triggers on plan, statement, and function tables: 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.

Renaming triggering tables or tables that the triggered action references: You cannot rename a table for which a trigger is defined (the triggering table). Except for the triggering 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.
- Revoking a privilege on which the trigger depends causes the trigger and its package to be invalidated.

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.

Values of special registers: The values of the special registers are saved before a trigger is activated and are restored on return from the trigger.

Table 40 gives the rules for special registers within a trigger. 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 40 (Page 1 of 2). 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 MVS TIMEZONE parameter

CREATE TRIGGER

Table 40 (Page 2 of 2). Rules for the values of special registers in triggers

Special register	The value is
CURRENT DEGREE	Inherited from the triggering SQL operation (delete, insert, update)
CURRENT LC_CTYPE	
CURRENT OPTIMIZATION HINT	
CURRENT PATH	
CURRENT PRECISION	
CURRENT RULES	
CURRENT SERVER	
CURRENT SQLID	
USER	

Errors when binding triggers: When a CREATE TRIGGER statement is bound, the SQL statements within the triggered action may not be fully parsed. Syntax errors in those statements might not be caught until the CREATE TRIGGER statement is executed.

Errors when 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 SQLSTATE statement or that contains a RAISE_ERROR function are returned with SQLCODE -438 and the SQLSTATE that is specified in the SIGNAL SQLSTATE statement or the RAISE_ERROR condition. Other non-severe errors are returned with SQLCODE -723 and SQLSTATE 09000.

Warnings are not returned.

Limiting processor time: DB2's resource limit facility allows you to specify the maximum amount of processor time for a dynamic, manipulative SQL statement (SELECT, INSERT, UPDATE, and DELETE). The execution of a trigger is counted as part of the triggering SQL statement.

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 PARTNO 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

```

CREATE TRIGGER

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 SQLSTATE 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
```

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 *subselect*, 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)

Additional authorization is required if the definition of the view references any user-defined functions or cast functions that were generated for a distinct type. The privilege set defined below must include the EXECUTE privilege on the referenced functions.

Authority requirements depend in part on the choice of the view's owner. For information on how to choose the owner, see the description of *view-name* in "Description" on page 628.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of the owner of the plan or package:

- 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 subselect. (It could refer instead, for example, to catalog tables or views thereof.) Otherwise, the owner of the view must be the owner of the plan or package.

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 view's subselect.

- If the privilege set lacks SYSADM or SYSCTRL authority, the owner of the view must be the owner of the application plan or package.

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 subselect. (It could refer instead, for example,

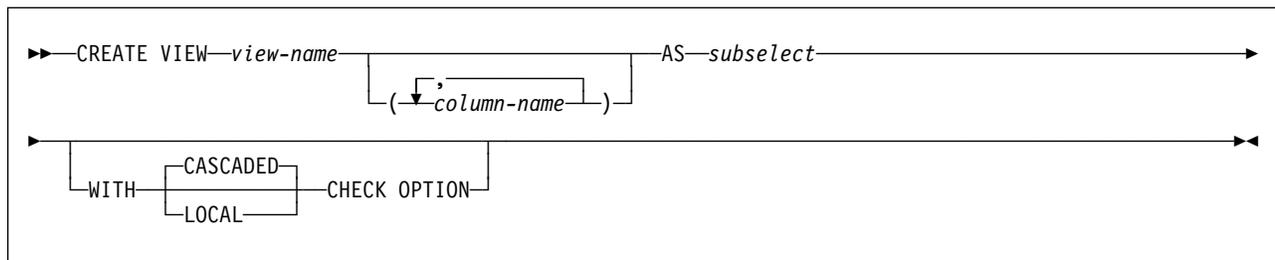
CREATE VIEW

to catalog tables or views thereof.) Otherwise, the owner of the view must be one of the authorization IDs of the process.

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 view's subselect.

- If the SQL authorization ID of the process lacks SYSADM or SYSCTRL authority, 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 must not identify a table, view, alias, or synonym that exists at the current server.

If 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.) In either case, the authorization ID that qualifies the name is the view's owner.

If the view name is unqualified and the statement is embedded in an application program, the owner of the view is the authorization ID that serves as the implicit qualifier for unqualified object names. This is the authorization ID of the QUALIFIER operand when the plan or package was created or last rebound. If QUALIFIER was not used, the owner of the view is the owner of the package or plan.

If the view name is unqualified and the statement is dynamically prepared, the owner of the view is the SQL authorization ID of the process.

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 is created on a table for which the owner has the SELECT privilege with the GRANT option and the 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 or view is identified in the first FROM clause of the subselect. 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 new 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 privilege 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.

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 subselect. 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 subselect.

You must specify a list of column names if the result table of the subselect 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).

AS subselect

Defines the view. At any time, the view consists of the rows that would result if the subselect were executed.

#

subselect must not refer to any declared temporary tables. It must also not refer to host variables or include parameter markers (question marks). For an explanation of *subselect*, see “subselect” on page 311.

WITH ... CHECK OPTION

Specifies the constraint that every row that is inserted or updated through the view must conform to the definition of the view. DB2 enforces this constraint whenever rows of the view are inserted or updated. If the search condition is not true for an inserted or updated row, an error occurs and no rows are inserted or updated.

The search condition of a view is the search condition that is specified in the first WHERE clause of the subselect that defines the view. If the view is defined without a search condition (a WHERE clause was not specified) then the view behaves as if it were defined with a search condition that is always true.

A check option must not be specified if any of the following conditions are true:

- The view is read-only.
- The search condition of the view includes a subquery, or the search condition of an underlying view includes a subquery.
- The search condition of the view includes a user-defined function that is nondeterministic or has an external action.
- The *subselect* refers to a created temporary table.

|
|

A check option is ignored if the view is updatable but does not have a search condition. If a check option is specified for an updatable view that does not allow inserts, the constraint applies only to updates.

If a check option is not specified, the search condition of the view is not used to check any insert or update operations that use the view. Rows that do not conform to the definition of the view can be inserted or updated, but then the rows are not accessible through the view (`SELECT * FROM V`).

The difference between the two forms of the check option, `CASCADED` and `LOCAL`, is meaningful only when views are defined on each other. The view upon 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.

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. See Example 2 on page 632 for an example of this situation.

Table 41 illustrates the effect of using the default check option, `CASCADED`. The information in Table 41 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 41 (Page 1 of 2). Examples using default check option, `CASCADED`

SQL statement	Description of result
<code>INSERT INTO V1 VALUES(5)</code>	Succeeds because <i>V1</i> does not have a check option and it is not dependent on any other view that has a check option.
<code>INSERT INTO V2 VALUES(5)</code>	Results in an error because the inserted row does not conform to the search condition of <i>V1</i> which is implicitly is part of the definition of <i>V2</i> .
<code>INSERT INTO V3 VALUES(5)</code>	Results in an error because the inserted row does not conform to the search condition of <i>V1</i> .

Table 41 (Page 2 of 2). Examples using default check option, CASCADED

SQL statement	Description of result
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.

Table 42 shows the views in which search conditions are checked during an INSERT or UPDATE operation:

Table 42. Views in which search conditions are checked during INSERT and UPDATE operations

View used in INSERT or UPDATE operation	x = LOCAL y = LOCAL	x = CASCADED y = CASCADED	x = LOCAL y = CASCADED	x = CASCADED y = LOCAL
V1	None	None	None	None
V2	V2	V2, V1	V2	V2, V1
V3	V2	V2, V1	V2	V2, V1
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

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
- The first SELECT clause specifies the keyword DISTINCT
- The outer subselect contains a GROUP BY clause
- The outer subselect contains a HAVING clause
- The first SELECT clause contains a column function
- It contains a subquery such that the base object of the outer subselect, and of the subquery, is the same table
- The first FROM clause identifies a read-only view

CREATE VIEW

A read-only view cannot be the object of an INSERT, UPDATE, or DELETE statement. A view that includes GROUP BY or HAVING cannot be referred to in a subquery of a basic predicate.

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 unions or references to remote objects. A view 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 DSN8610.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 DSN8610.VPROJRE1
  (PROJNO, PROJNAME, PROJDEP, RESPEMP,
   FIRSTNME, MIDINIT, LASTNAME)
AS SELECT ALL
  PROJNO, PROJNAME, DEPTNO, EMPNO,
  FIRSTNME, MIDINIT, LASTNAME
FROM DSN8610.PROJ, DSN8610.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: When a view that is defined WITH LOCAL CHECK OPTION is defined on a view that was defined without a check option. You can update or insert rows that do not conform to the definition of the view. Consider the following views:

```
CREATE VIEW UNDER AS SELECT * FROM DSN8610.EMP
  WHERE SALARY < 35000;

CREATE VIEW OVER AS SELECT * FROM UNDER
  WHERE SALARY > 30000 WITH LOCAL CHECK OPTION;
```

The following UPDATE statement that uses OVER is successful because the updated rows only need to conform to the definition of OVER (SALARY > 30000):

```
UPDATE OVER SET SALARY = SALARY + 5000;
```

However, not all of the rows that you can retrieve through view OVER (over 35,000 rows) are accessible using view UNDER. For example, issuing:

```
SELECT * FROM UNDER
```

returns no rows because no rows conform to the definition of UNDER (SALARY < 35000).

With the CASCADED CHECK OPTION, this situation cannot occur. If OVER had been defined with the WITH CASCADED CHECK OPTION, the UPDATE statement would have failed because the updated rows would not conform to the conjunction of the search conditions OVER and UNDER (SALARY > 3000 and SALARY < 35000).

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.

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)

The SELECT statement of the cursor is one of the following:

- The prepared select statement identified by *statement-name*
- The specified *select-statement*

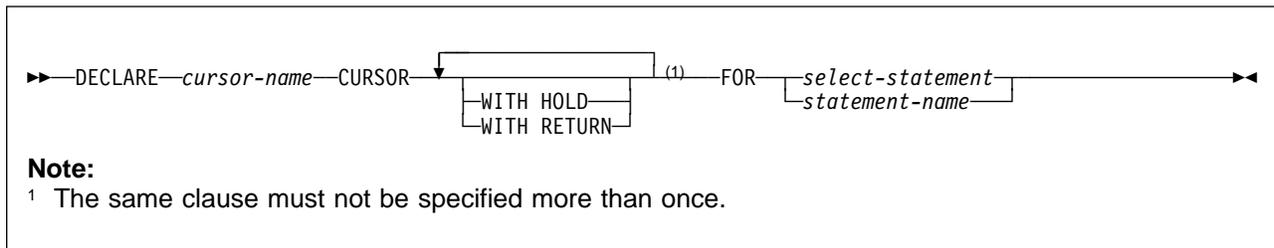
If *statement-name* is specified:

- The privilege set is determined by the DYNAMICRULES behavior in effect (run, bind, define, or invoke) and is summarized in Table 29 on page 342. (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 61.)
- 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



Description

cursor-name

Names the cursor. The name must not identify a cursor that has already been declared in the source program.

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 the changes in the current unit of work, but releases only locks that are not required to maintain the cursor. Afterwards, an initial FETCH statement is required before a positioned update or delete statement can be executed. After the initial FETCH, the cursor is positioned on the row following the one it was positioned on before the commit operation.

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 Section 3 of *DB2 Application Programming and SQL Guide*.

DECLARE CURSOR

WITH RETURN

Specifies that the cursor, if it is declared in a stored procedure, can return a result set to the caller.

select-statement

Specifies the result table of the cursor. The *select-statement* must not include parameter markers, but can include references to host variables. The declarations of the host variables must precede the DECLARE CURSOR statement in the source program. See “select-statement” on page 331 for an explanation of *select-statement*.

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 757.

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.

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 more than one table or view
- The first SELECT clause specifies the keyword DISTINCT
- The outer subselect contains a GROUP BY clause
- The outer subselect contains a HAVING clause
- The first SELECT clause contains a column function
- It contains a subquery such that the base object of the outer subselect, and of the subquery, is the same table
- The first FROM clause identifies a read-only view
- The first FROM clause identifies a catalog table with no updatable columns
- The first FROM clause identifies a table function
- The first FROM clause contains a nested table expression
- A UNION or UNION ALL operator is present
- An ORDER BY clause is present
- A FOR FETCH ONLY or a FOR READ ONLY clause is present
- A FOR UPDATE OF clause is not specified and the isolation level at which the statement is executed is UR

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.

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.

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 and the SET CURRENT PACKAGESET statement is used to select the packages.

Positioned deletes and isolation level UR: Specify FOR UPDATE OF 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.
- The cursor is declared with the WITH HOLD option if the stored procedure performs a 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.

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 DSN8610.DEPT. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO
FROM DSN8610.DEPT
WHERE ADMRDEPT = 'A00';
```

Example 2: Declare C2 as the cursor for a statement named STMT2.

```
EXEC SQL DECLARE C2 CURSOR FOR STMT2;
```

DECLARE CURSOR

Example 3: Declare C3 as the cursor for a query to be used in positioned updates
of the table DSN8610.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 DSN8610.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 4: In stored procedure SP1, declare C4 as the cursor for a query of the
table DSN8610.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 DSN8610.PROJ  
WHERE DEPTNO = 'A01';
```

DECLARE GLOBAL TEMPORARY TABLE

The DECLARE GLOBAL TEMPORARY TABLE statement defines a declared
 # temporary table for the current application process and instantiates an empty
 # instance of the table for the process.

Invocation

This statement can be embedded in an application program or issued interactively.
 # It is an executable statement that can be dynamically prepared.

Authorization

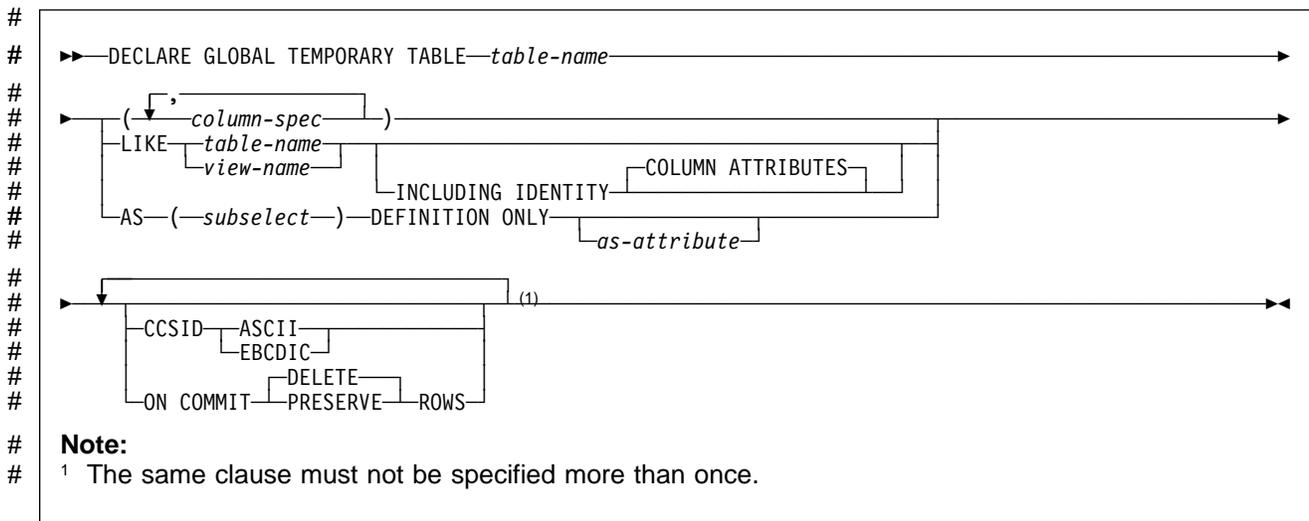
None required, unless the LIKE clause is specified when additional privileges might
 # be required.

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 TEMP, which is the database for declared
 # temporary tables.
- # • The USE privilege to use the table spaces in the database that is defined as
 # TEMP.
- # • All table privileges on the table and authority to drop the table. (Table privileges
 # for a declared temporary table cannot be granted or revoked.)

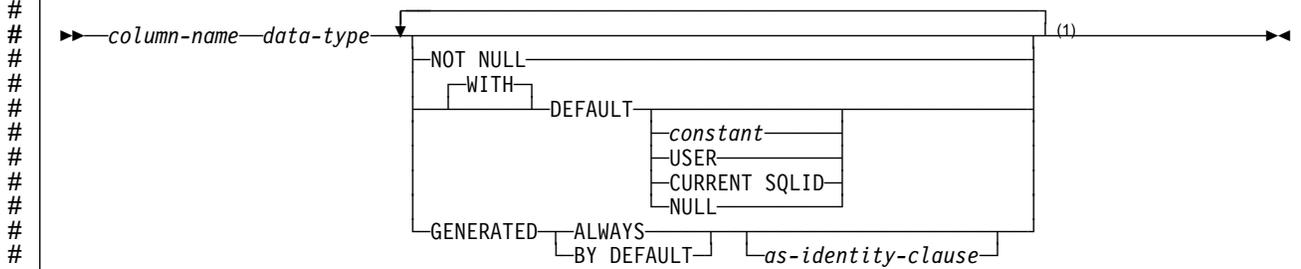
These implicit privileges are not recorded in the DB2 catalog and cannot be
 # revoked.

Syntax



DECLARE GLOBAL TEMPORARY TABLE

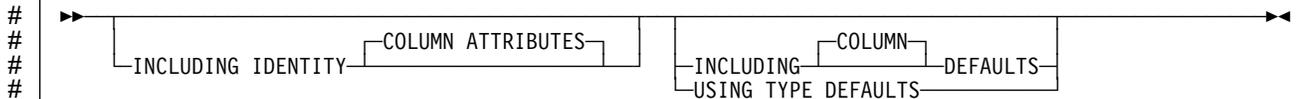
column-spec:



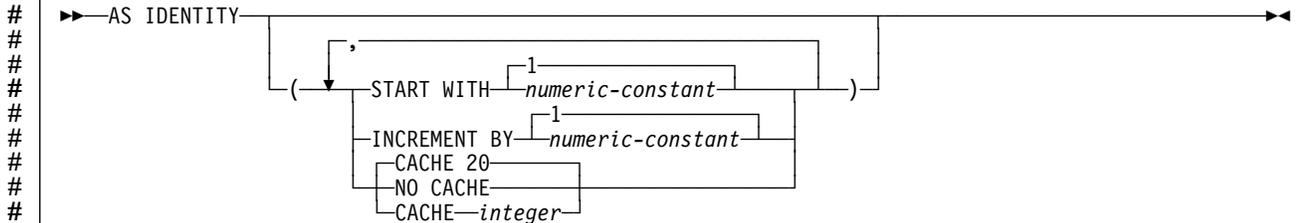
Note:

¹ The same clause must not be specified more than once. The FOR *sub-type* DATA clause can be specified as part of *data-type*.

AS-attribute:



as-identity-clause:



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.

```

# PUBLIC implicitly acquires ALL PRIVILEGES on the table and authority to drop
# the table. These implicit privileges are not recorded in the DB2 catalog and
# cannot be revoked.

# column-spec
# 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 any built-in data
# type that can be specified for the CREATE TABLE statement except for a LOB
# (BLOB, CLOB, and DBCLOB) or ROWID 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-data-type" on page
# built-in-data-type on page 575.

# NOT NULL
# Specifies that the column cannot contain nulls. Omission of NOT NULL
# indicates that the column can contain nulls.

# DEFAULT
# The default value assigned to the column in the absence of a value specified
# on INSERT. Do not specify DEFAULT for a column that is defined AS
# IDENTITY (an identity column); DB2 generates default values.
# If DEFAULT is not specified, the default value for the column is the null value.
# 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 string    Blanks
#           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.

# USER
# Specifies the value of the USER special register at the time of INSERT or
# LOAD as the default value for the column. If USER 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 USER special register, which is 8
# bytes.

```

DECLARE GLOBAL TEMPORARY TABLE

CURRENT SQLID
Specifies the value of the SQL authorization ID of the process at the time
of INSERT 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, which is 8 bytes.

NULL
Specifies null as the default value for the column.

In a given column definition:

- # • NOT NULL and DEFAULT NULL cannot both be specified.
- # • DEFAULT cannot be specified for an identity column.
- # • 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.

For more information on the effect of specifying various combinations of the
NOT NULL and DEFAULT clauses, see Table 35 on page 580, which provides
a summary.

GENERATED
Specifies that DB2 generates values for the column. You must specify
GENERATED if the column is to be considered an identity column (a column
defined with the AS IDENTITY clause).

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.

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, DECIMAL with a scale of zero). For more
information, see the description of the AS IDENTITY clause for “CREATE
TABLE” on page 570.

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
must identify a table, view, synonym, or alias that exists at the current server.
The identified table must not be an auxiliary table or a declared temporary
table.

The privilege set must include the SELECT privilege on the identified table or
view.

This clause is similar to the LIKE clause on CREATE TABLE, but it has the
following differences:

DECLARE GLOBAL TEMPORARY TABLE

- 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. When the identified object is a view, only the column name, data type, and nullability attributes are determined from the columns of the specified view.

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the new table inherits all of the identity attributes of the identity column. If the table identified by LIKE does not have an identity column, the INCLUDING IDENTITY clause is ignored. If the LIKE clause identifies a view, INCLUDING IDENTITY COLUMN ATTRIBUTES cannot be specified.

AS (*subselect*) DEFINITION ONLY

Specifies that the columns of the table are to have the same name and description as the columns that would appear in the derived result table of the *subselect* if the *subselect* were to be executed. The use of AS *subselect* is an implicit definition of *n* columns for the table, where *n* is the number of columns that would result from the *subselect*.

DEFINITION ONLY indicates that the *subselect* is not executed. Therefore, there is no result table with a set of rows with which to automatically populate the declared temporary table. However, you can use the INSERT INTO statement with the same *subselect* specified in the AS clause to populate the declared temporary table with the set of rows from the result table of the *subselect*.

The implicit definition includes all the attributes of the *n* columns of *subselect* that are applicable for a declared temporary table (see *column-spec*) with the exception of these column attributes:

- The default value assigned to a column when a value is not specified on INSERT
- The identity attribute, if any

The behavior of these column attributes are controlled with the INCLUDING or USING TYPE DEFAULTS clauses, which are defined below.

If *subselect* results in a column having a LOB data type, a ROWID data type, or a distinct type, the DECLARE GLOBAL TEMPORARY statement fails.

If *subselect* 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.

The implicitly defined columns of the declared temporary table inherit the names of the columns from the result table of the *subselect*. Therefore, a column name must be specified in the *subselect* for all result columns. For result columns that are derived from expressions, constants, and functions, the

DECLARE GLOBAL TEMPORARY TABLE

subselect must include the *AS column-name* clause immediately after the result
column.

The *subselect* must not refer to host variables or include parameter markers
(question marks).

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the declared temporary table inherits the identity attribute, if
any, of the columns resulting from *subselect*. In general, the identity
attribute is copied if the element of the corresponding column in the table,
view, or *subselect* is the name of a table column or the name of a view
column that directly or indirectly maps to the name of a base table column
with the identity property. The columns of the new table do not inherit the
identity attribute in the following cases:

- # • The select list of the *subselect* includes multiple instances of an identity
column name (that is, selecting the same column more than once).
- # • The select list of the *subselect* includes multiple identity columns (that
is, it involves a join).
- # • The identity column is included in an expression in the select list.
- # • The *subselect* includes a set operation (union).

If INCLUDING IDENTITY is not specified, the declared temporary table will
not have an identity column.

INCLUDING COLUMN DEFAULTS

Specifies that the declared temporary table inherits the default values of the
columns resulting from *subselect*. A default value is the value assigned to a
column when a value is not specified on an INSERT.

Do not specify INCLUDING COLUMN DEFAULTS, if you specify USING
TYPE DEFAULTS.

If neither INCLUDING COLUMN DEFAULTS nor USING TYPE DEFAULTS
is specified, the default values of the columns of the declared temporary
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 for the declared temporary 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 *subselect*, as follows:

#	Data type	Default value
#	Numeric	0
#	Fixed-length string	Blanks
#	Varying-length string	A string of length 0
#	Date	CURRENT DATE
#	Time	CURRENT TIME
#	Timestamp	CURRENT TIMESTAMP

Do not specify USING TYPE DEFAULTS, if you specify INCLUDING
COLUMN DEFAULTS.

DECLARE GLOBAL TEMPORARY TABLE

CCSID *encoding-scheme*
Specifies the encoding scheme for string data stored in the table. For declared
temporary tables, an encoding scheme cannot be specified for the table space
or database that contains the table. Therefore, the database that is defined AS
TEMP and the table spaces within it can contain declared temporary tables with
a mixture of encoding schemes.

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.

Usually, each encoding scheme requires only a single CCSID. Additional
CCSIDs are needed when mixed or graphic data is used.

The CCSID clause applies to all declared temporary tables. A temporary table
that is defined with the LIKE clause does not inherit the encoding scheme of
the copied table.

The default for the CCSID clause is the value of field DEF ENCODING
SCHEME on installation panel DSNTIPF, and the actual CCSID value is taken
from the default encoding scheme's CODED CHARACTER SET field on panel
DSNTIPF.

ON COMMIT *commit-action* **ROWS**
Specifies whether the contents of the table are to be deleted or preserved
across a commit operation.

DELETE
The rows of the table are deleted if no WITH HOLD cursors are open on
the table. DELETE is the default.

PRESERVE
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.

Notes

Instantiation, scope, and termination: Let P denote an application process and
let T be a declared temporary table in an application program in P:

- # • When a program in P issues a DECLARE GLOBAL TEMPORARY TABLE
statement, an empty instance of T is created.
- # • Any program in P can reference T and any of those references is a reference
to that same instance of T. (If a DECLARE GLOBAL TEMPORARY statement
is specified within the SQL procedure language compound statement,
BEGIN-END, the scope of the declared temporary table is the application
process and not the compound statement.)

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 application
server at which T was declared terminates, T is dropped and its instantiated
rows are destroyed.

DECLARE GLOBAL TEMPORARY TABLE

```
#           • If T is defined with the ON COMMIT DELETE ROWS 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
#           DELETE FROM T (all rows).
#
#           • When a rollback operation terminates a unit of work in P, the rollback undoes
#           the rows of T up to the last commit or specified external savepoint but leaves
#           all rows that existed up to that point.
#
#           • When the application process that declared T terminates, T is dropped and its
#           instantiated rows are destroyed.
#
#           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 PRESERVE COMMIT DELETE ROWS
#           attribute and the table was explicitly dropped with the DROP TABLE statement
#           before the thread's commit operation.
#
#           When the thread is reused, the declared temporary table is dropped and its rows
#           are destroyed. However, if you do not explicitly drop all declared temporary tables
#           before 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 TEMP database. So,
#           instead of using the implicit drop feature of thread reuse to drop your declared
#           temporary tables, it is recommended that you explicitly use the DROP TABLE
#           statement to drop your declared temporary tables before the thread performs a
#           commit operation and becomes idle. Explicitly dropping the tables 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 type 1 or type 2 thread.
#
#           Privileges: When a declared temporary table is defined, PUBLIC implicitly is
#           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.
#
#           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.
```

DECLARE GLOBAL TEMPORARY TABLE

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.

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. The bind
is completed at run time 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 after the table is explicitly dropped or the application process finishes
running.

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.

Parallelism support: Only I/O and CP parallelism are supported. Queries that
involve a declared temporary table are not eligible for Sysplex query parallelism.

Restrictions on the use of declared temporary tables: Declared temporary
tables cannot be:

- # • Specified in referential constraints.
- # • 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 you are referring to a base table.

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 ON	LABEL ON
# CREATE ALIAS	LOCK TABLE
# CREATE FUNCTION (TABLE LIKE clause)	RENAME TABLE
# CREATE PROCEDURE (TABLE LIKE clause)	REVOKE (table or view privileges)
# CREATE TRIGGER	

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;
```

DECLARE GLOBAL TEMPORARY TABLE

Example 2: Assume that base table USER1.EMPTAB exists and that it contains
three columns, one of which is an identity column. Declare a temporary table that
has the same column names and attributes (including identity attributes) as the
base table.

```
# DECLARE GLOBAL TEMPORARY TABLE TEMPTAB1  
# LIKE USER1.EMPTAB  
# INCLUDING IDENTITY  
# ON COMMIT PRESERVE ROWS;
```

In the above example, DB2 uses SESSION as the implicit qualifier for TEMPTAB1.

DECLARE STATEMENT

The DECLARE STATEMENT statement is used for application program documentation. It declares names that are used to identify prepared SQL statements.

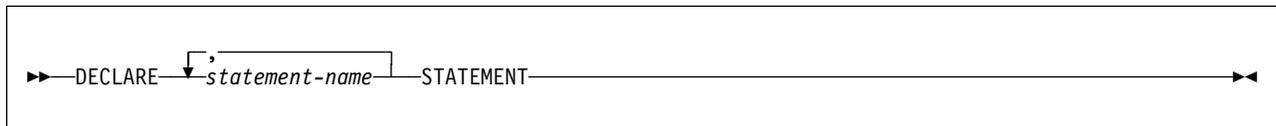
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax



Description

statement-name **STATEMENT**

Lists one or more names that are used in your application program to identify prepared SQL statements.

Example

This example shows the use of the DECLARE STATEMENT statement in a PL/I program.

```

EXEC SQL DECLARE OBJECT_STATEMENT STATEMENT;

EXEC SQL INCLUDE SQLDA;
EXEC SQL DECLARE C1 CURSOR FOR OBJECT_STATEMENT;

( SOURCE_STATEMENT IS "SELECT DEPTNO, DEPTNAME,
  MGRNO FROM DSN8610.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

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 Section 3 of *DB2 Application Programming and SQL Guide* and Chapter 2 of *DB2 Command Reference*.)

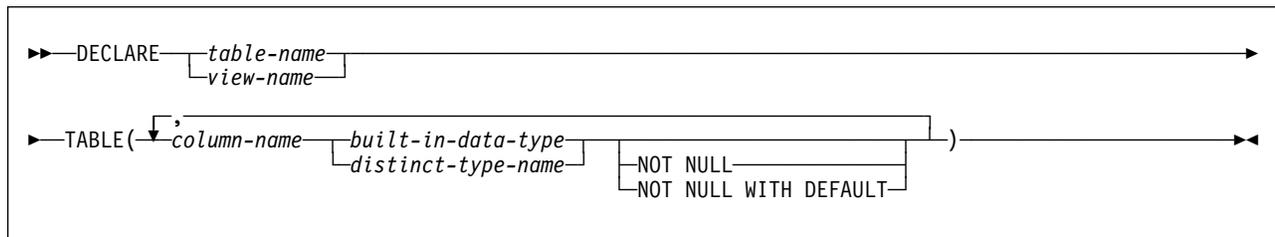
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax



Description

table-name or *view-name*
Is the name of the table or view you want 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
Is 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 types within your SQL statements.

| *built-in-data-type*
Is the built-in data type of the column. Use one of the built-in data types. See "built-in-data-type" on page built-in-data-type on page 575 for details.

| *distinct-type-name*
Is the distinct type (user-defined data type) of the column. An implicit or explicit schema name qualifies the name.

NOT NULL

Is used for a column that does not allow null values, and does not provide a default value.

NOT NULL WITH DEFAULT

Is used for a column that 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, we recommend that you use the built-in data type upon which the distinct type is sourced instead. Using the source 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, query column SOURCETYPE in catalog table SYSDATATYPES.

Examples

Example 1: Declare the sample employee table, DSN8610.EMP.

```
EXEC SQL DECLARE DSN8610.EMP TABLE
  (EMPNO      CHAR(6)      NOT NULL,
   FIRSTNME  VARCHAR(12)  NOT NULL,
   MIDINIT   CHAR(1)      NOT NULL,
   LASTNAME  VARCHAR(15)  NOT NULL,
   WORKDEPT  CHAR(3)      ,
   PHONENO   CHAR(4)      ,
   HIREDATE  DATE          ,
   JOB       CHAR(8)      ,
   EDLEVEL   SMALLINT    ,
   SEX       CHAR(1)      ,
   BIRTHDATE DATE          ,
   SALARY    DECIMAL(9,2) ,
   BONUS     DECIMAL(9,2) ,
   COMM      DECIMAL(9,2) );
```

Example 2: Assume that table CANADIAN_SALES keeps information for your company's sales in Canada. The table was created with the following definition:

```
CREATE TABLE CANADIAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER,
   YEAR          INTEGER,
   TOTAL         CANADIAN_DOLLAR);
```

CANADIAN_DOLLAR is a distinct type that was created with the following statement:

```
CREATE DISTINCT TYPE CANADIAN_DOLLAR
  AS DECIMAL(9,2) WITH COMPARISONS;
```

Declare the CANADIAN_SALES table, using the source type for CANADIAN_DOLLAR instead of the distinct type name.

DELETE

The DELETE statement deletes rows from a table or view. 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.

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 is used to delete exactly one row, as determined by the current position of a cursor.

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 user-defined 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

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
- SYSADM authority

SQL standard rules are in effect as follows:

DELETE

- 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'

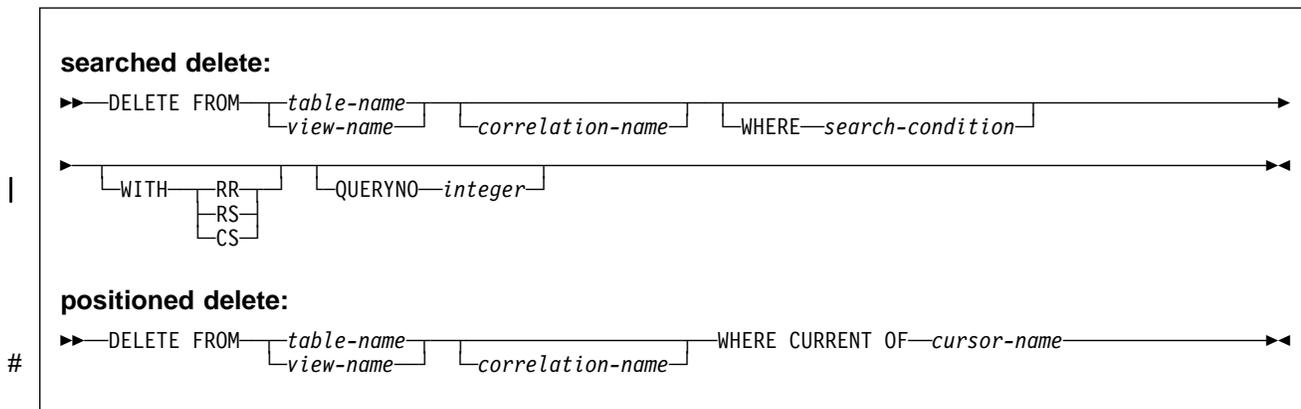
The owner of a view, unlike the owner of a table, might not have DELETE authority on the view (or might have DELETE authority without being able to grant it to others). The nature of the view itself can preclude its use for DELETE. For more information, see the description of authority in “CREATE VIEW” on page 627.

If an expression that refers to a function is specified, the privilege set must include any authority that is necessary to execute the function.

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 309.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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 29 on page 342. (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 61.)

Syntax



Description

FROM *table-name* or *view-name*

Identifies the object of the DELETE 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 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 627.)

In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must not be a remote DB2 Version 2 Release 3 subsystem.

correlation-name

Can be used within the *search-condition* or *positioned* DELETE to qualify
references to columns of the table or view. (For an explanation of correlation
names, see “Correlation names” on page 114.)

WHERE

Specifies the rows to be deleted. You can omit the clause, give a search condition or name 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 3. Language elements” on page 43. 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 and T2 must not be the same table.
- T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL.
- 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.

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 “Notes” on page 636. 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 not be read-only. (For an explanation of read-only result tables, see “DECLARE CURSOR” on page 634.) 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

DELETE

- 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 positioned on a row; that row is the one deleted. After the deletion, the cursor is positioned before the next row of its result table. If there is no next row, the cursor is positioned after the last row.

WITH

Specifies the isolation level used when locating the rows to be deleted by the statement.

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 columns of the plan tables for the rows that contain information about this SQL statement.

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, if hints are used. For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, see Section 5 (Volume 2) of *DB2 Administration Guide*.

Notes

Delete operation errors: If an error occurs during the execution of any delete operation, no rows are deleted. 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 INSERT, UPDATE, and DELETE operations 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.

Referential integrity: If the object table of the delete operation is a parent table:

- The rows selected for deletion must have no dependents in a relationship governed by a delete rule of RESTRICT or NO ACTION.
- The delete operation must not cascade to descendent rows that are dependents in a relationship governed by 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:

- The columns of foreign keys in any rows that are their dependents in a relationship governed by a delete rule of SET NULL and which allow nulls 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.

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.

#

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.

#

Triggers: If the identified table or the base table of the identified view has a delete trigger, the trigger is fired for each row deleted.

Number of rows deleted: Except as noted below, a DELETE operation sets SQLERRD(3) to the number of deleted rows. This number does not include any rows that were deleted as a result of a CASCADE delete rule.

#

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. The SQLERRD(3) field is also set to -1 if a table T is from a CREATED GLOBAL

DELETE

TEMPORARY TABLE. (For a complete description of the SQLCA, including
exceptions to the above, see “SQL communication area (SQLCA)” on page 883.

If the object table is SYSIBM.SYSSTRINGS, the rows selected for delete must be rows provided by the user (The value of the IBMREQD column is N).

Examples

Assume that the statements in these examples are embedded in PL/I programs.

Example 1: From the table DSN8610.EMP delete the row on which the cursor C1 is currently positioned.

```
EXEC SQL DELETE FROM DSN8610.EMP WHERE CURRENT OF C1;
```

Example 2: From the table DSN8610.EMP, delete all rows for departments E11 and D21.

```
EXEC SQL DELETE FROM DSN8610.EMP  
WHERE WORKDEPT = 'E11' OR WORKDEPT = 'D21';
```

DESCRIBE (prepared statement or table)

The DESCRIBE statement obtains information about a prepared statement or a designated table or view. For an explanation of prepared statements, see “PREPARE” on page 757 and “DESCRIBE PROCEDURE” on page 671.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

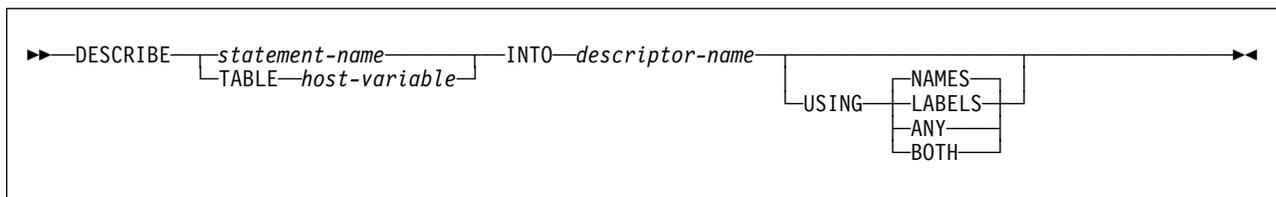
None required if the statement is used for a prepared statement. When it is used instead for a table or view, the privileges that are held by the authorization ID that owns the plan or package must include at least one of the following (if there is a plan, authorization checking is done only against 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

For an RRSF application that does not have a plan and in which the requester and the server are DB2 for OS/390 systems, authorization to execute the package is performed against the primary or secondary authorization ID of the process.

See “PREPARE” on page 757 for the authorization required to create a prepared statement.

Syntax



Description

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.

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- 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

DESCRIBE

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 Appendix C, “SQLCA and SQLDA” on page 883. See “Identifying an SQLDA in C or C++” on page 907 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 on page 661.

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 on page 662.

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 ON 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

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 894 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 894 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.

DESCRIBE

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 Appendix C, “SQLCA and SQLDA” on page 883.

SQLDAID DB2 sets the first 6 bytes to 'SQLDA ' (5 letters followed by the space character) and the eighth byte to a space character. The seventh byte is set to indicate the number of SQLVAR entries that are needed to describe each column of the result table as follows:

space The value of space occurs when:

- USING BOTH was not specified and the columns being described do not include LOBs or distinct types. Each column only needs one SQLVAR entry. If the SQL standard option is yes, DB2 sets SQLCODE to warning code +236. Otherwise, SQLCODE is zero.
- USING BOTH was specified and the columns being described do not include LOBs or distinct types. Each column needs two SQLVAR entries. DB2 sets SQLD to two times the number of columns of the result table. The second set of SQLVARs is used for the labels.

2 Each column needs two SQLVAR entries. Two entries per column are required when:

- USING BOTH was not specified and the columns being described include LOBs or distinct types or both. DB2 sets the second set of SQLVAR entries with information for the LOBs or distinct types being described.
- USING BOTH was specified and the columns include LOBs but not distinct types. DB2 sets the second set of SQLVAR entries with information for the LOBs and labels for the columns being described.

3 Each column needs three SQLVAR entries. Three entries are required only when USING BOTH is specified and the columns being described include distinct types. The presence of LOB data does not matter. It is the distinct types and not the LOBs that cause the need for three SQLVAR entries per column when labels are also requested. DB2 sets the second set of SQLVAR entries with information for the distinct types (and LOBs, if any) and the third set of SQLVAR entries with the labels of the columns being described.

#

A REXX SQLDA does not contain this field.

```

SQLDABC The length of the SQLDA in bytes. DB2 sets the value to
        SQLN×44+16.
#
        A REXX SQLDA does not contain this field.
|
SQLD    If the prepared statement is a query, DB2 sets the value to the
        number of columns in the object being described (the value is actually
        twice the number of columns in the case where USING BOTH was
        specified and the result table does not include LOBs or distinct types).
        Otherwise, if the statement is not a query, DB2 sets the value to 0.
|
SQLVAR  An array of field description information for the column being
        described. There are two types of SQLVAR entries—the base
        SQLVAR and the extended SQLVAR.
|
        If the value of SQLD is 0, or is greater than the value of SQLN, no
        values are assigned to any occurrences of SQLVAR. If the value of
        SQLN was set so that there are enough SQLVAR occurrences to
        describe the specified columns (columns with LOBs or distinct types
        and a request for labels increase the number of SQLVAR entries that
        are needed), the values are assigned to the first n occurrences of
        SQLVAR so that the first occurrence of SQLVAR contains a
        description of the first column, the second occurrence of SQLVAR
        contains a description of the second column, and so on. This first set
        of SQLVAR entries are referred to as base SQLVAR entries. Each
        column always has a base SQLVAR entry.
|
        If the DESCRIBE statement included the USING BOTH clause, or the
        columns being described include LOBs or distinct types, additional
        SQLVAR entries are needed. These additional SQLVAR entries are
        referred to as the extended SQLVAR entries. There can be up to two
        sets of extended SQLVAR entries for each column.
|
#
#
#
#
#
#
#
#
#
        For REXX, the SQLVAR is a set of stem variables that begin with
        stem.n, instead of a structure. The REXX SQLDA uses only a base
        SQLVAR. The way in which DB2 assigns values to the SQLVAR
        variables is the same as for other languages. That is, the stem.1
        variables describe the first column in the result table, the stem.2
        variables describe the second column in the result table, and so on. If
        USING BOTH is specified, the stem.n+1 variables also describe the
        first column in the result table, the stem.n+2 variables also describe
        the second column in the result table, and so on.
|
        The base SQLVAR:
|
SQLTYPE A code that indicates the data type of the column and
        whether the column can contain null values. For the
        possible values of SQLTYPE, see Table 64 on
        page 900.
|
SQLLEN  A length value depending on the data type of the result
        columns. SQLLEN is 0 for LOB data types. For the other
        possible values of SQLLEN, see Table 64 on page 900.
|
#
#
#
        In a REXX SQLDA, for DECIMAL or NUMERIC columns,
        DB2 sets the SQLPRECISION and SQLSCALE fields
        instead of the SQLLEN field.

```

DESCRIBE

#

SQLDATA The CCSID of a string column. For possible values, see Table 65 on page 901.

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 312.

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.

#

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 on page 664.

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 890.

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 OS/390 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 DSNTIPF 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 Section 3 of *DB2 Installation Guide* .

Avoiding double preparation when using REOPTVAR: If bind option REOPT(VARS) 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 variable values. If the statement has input variable values, it must then be prepared again when it is opened or executed. 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.

Errors occurring on DESCRIBE: In local and remote processing, the DEFER(PREPARE) and REOPT(VARS) bind options can cause some errors that are normally issued during PREPARE processing to be issued on DESCRIBE.

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 on page 661.

```
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 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 INTO descriptor-name
                   └── host-variable ───▶

```

Description

cursor-name or *host-variable*

Identifies a cursor by the specified *cursor-name* or the cursor name contained in the *host-variable*. The name must identify a cursor that has already been allocated in the source program.

A cursor name is a long identifier.

If a host variable is used:

- It must be a character string variable with a length attribute that is not greater than 18 bytes (A C NUL-terminated character string can be up to 19 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 Section 7 of *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

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 DSNTIPF 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.

|
|
|
|

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 757 and “DESCRIBE PROCEDURE” on page 671.

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 Appendix C, “SQLCA and SQLDA” on page 883. See “Identifying an SQLDA in C or C++” on page 907 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 (prepared statement or table)” on page 659). 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 662) 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-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 890.

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 890.

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 on page 664.

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;
.
.
.

```

DESCRIBE INPUT

| This example uses the first technique described in Allocating the SQLDA on
| page 661 to allocate the 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 INTO descriptor-name
                       └── host-variable ───┘
  
```

Description

procedure-name or *host-variable*

Identifies the stored procedure to describe by the specified procedure name or the procedure name contained in the host variable.

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 a long identifier that contains the location name that identifies the DBMS at which the procedure is stored. The second part is a short identifier that contains the schema name of the stored procedure. The last part is a long 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 one implicit qualifier. The implicit qualifier is the location name of the current server. An implicit schema name is not needed as a qualifier. Successful execution of the ASSOCIATE LOCATOR statement only requires that the unqualified procedure name in the statement is the same as the procedure name in the most recently executed CALL statement that was specified with an unqualified procedure name. (The implicit schema name for the unqualified name in the CALL statement is not considered in the match.) The rules for how the procedure name must be specified are described below.

DESCRIBE PROCEDURE

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 application server. Regardless of the application 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

When the DESCRIBE PROCEDURE statement is executed, the procedure name or specification must identify a stored procedure that the requester has already invoked using the CALL statement.

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 name was specified on the CALL statement, you must use a two-part name in the DESCRIBE PROCEDURE statement. However, there is one condition under which the names do not have to match. If the CALL statement was made with a three-part name and the current server is the same as the location in the three-part name, you can omit the location name and specify a two-part name.

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.

Notes

A value of -1 in the SQLIND field indicates that an estimated number of rows in the result set is not provided. DB2 for OS/390 always sets SQLIND to -1.

DESCRIBE PROCEDURE does not return information about the parameters expected by the stored procedure.

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;
```

DROP

The DROP statement deletes 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

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 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

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, or user-defined function, the privilege set that is defined below must include at least one of the following:

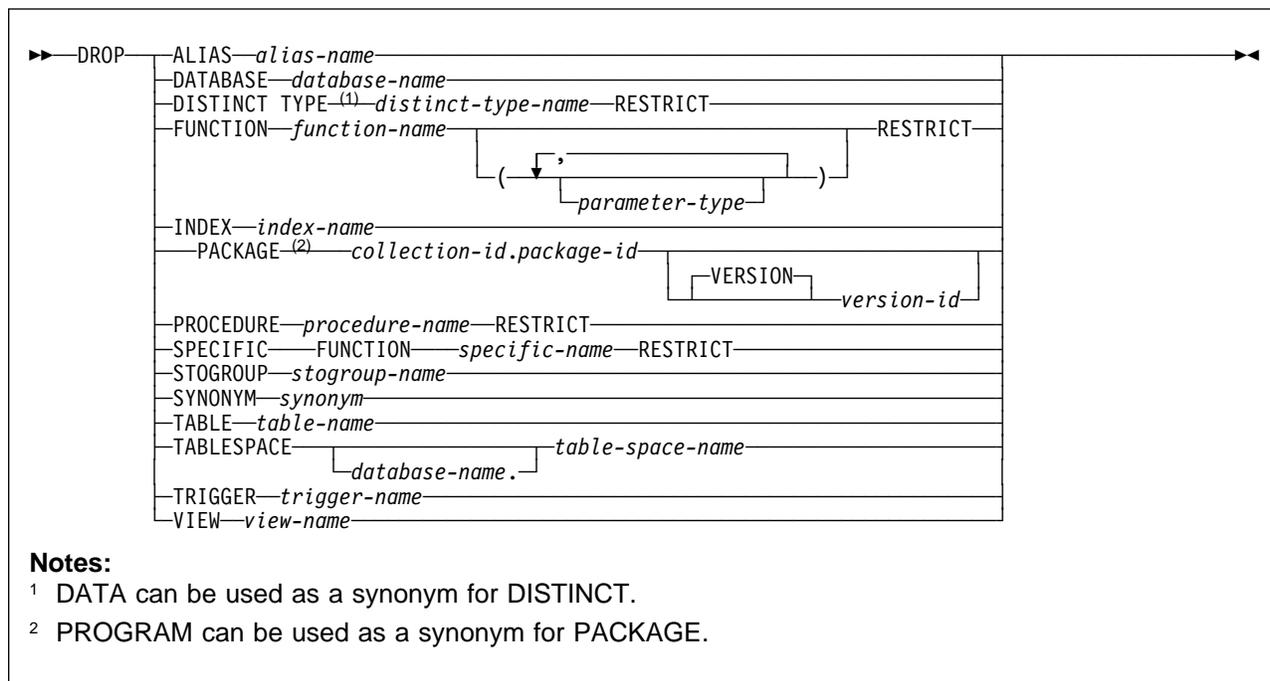
- Ownership of the object ³⁵
- The DROPIN privilege for the schema or all schemas
- SYSADM or SYSCTRL authority

³⁵ Not applicable for stored procedures defined in releases of DB2 for OS/390 prior to Version 6.

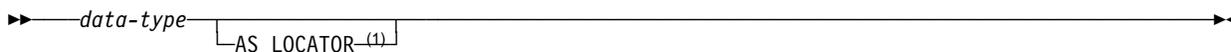
The authorization ID that matches the schema name implicitly has the DROPIN privilege on the schema.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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



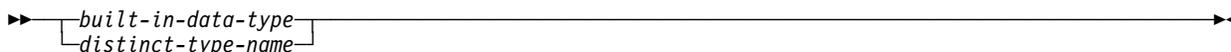
parameter type:



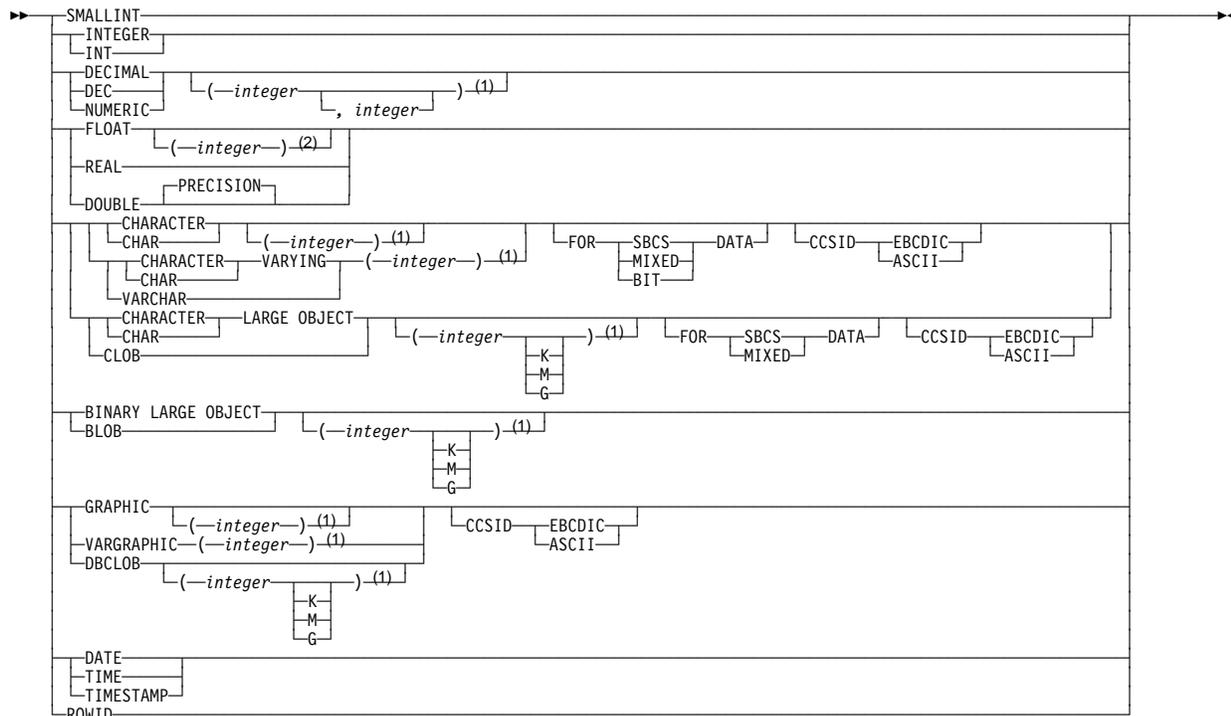
Note:

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-data-type:



Notes:

- 1 The values that are specified for length, precision, or scale attributes must match the values that were specified when the function was created. Coding specific values is optional. Empty parentheses, (), can be used instead to indicate that DB2 ignores the attributes when determining whether data types match.
- 2 The value that is specified does not have to match the value that was specified when the function was created because matching is based on data type (REAL or DOUBLE). 1<=integer<= 21 indicates REAL and 22<=integer<=53 indicates DOUBLE. Coding a specific value is optional. Empty parentheses cannot be used.

Description

ALIAS *alias-name*

Identifies the alias to be dropped. The name must identify an alias that exists at the current server. Dropping an alias has no effect on any view or synonym that was defined using the alias.

DATABASE *database-name*

Identifies the database to be dropped. The name must identify a database that exists at the current server. DSNDB04 or DSNDB06 must not be specified. If a work file database is specified, it must be in the stopped state and the privilege set must include SYSADM authority. A TEMP database can be dropped only if none of the table spaces or index spaces that it contains are actively being used.

Whenever a database is dropped, all of its table spaces, tables, index spaces, and indexes are also dropped.

#

DISTINCT TYPE *distinct-type-name* **RESTRICT**

Identifies the distinct type to be dropped. The name must identify a distinct type that exists at the current server. The required keyword **RESTRICT** enforces the rule that the distinct type is not dropped if any of the following dependencies exist:

- The definition of a column of a table or view 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.

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.

FUNCTION

Identifies the user-defined function to be dropped. The name must identify a function that has been defined with the **CREATE FUNCTION** statement at the current server. The name must not identify a cast function that was generated for a distinct type or a function that is in the **SYSIBM** schema. The required keyword **RESTRICT** enforces the rule that 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.

Whenever a function is dropped, all privileges on the user-defined function are also dropped. Any plans or packages that are dependent on the function dropped are made inoperative.

You can identify the particular function to be dropped 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), 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* **RESTRICT**

Identifies the function by its name. There must be exactly one function with *function-name* in the implicitly or explicitly specified schema; otherwise, an error occurs.

FUNCTION *function-name (parameter-type,...)* **RESTRICT**

Provides the function signature, which uniquely identifies the function. There must be exactly one function with the function signature in the implicitly or explicitly specified schema; otherwise, an error occurs.

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 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.

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.

The data types of the parameters must match the data types that were specified on the CREATE FUNCTION statement in the corresponding position. The number of data types and the logical concatenation of the data types are used to identify the function.

For data types that have a length, precision, or scale attribute, you can specify a value or use a set of empty parentheses:

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match.
 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 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. For example:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. For a complete list of the default lengths of data types, see “CREATE TABLE” on page 570.

For data types with a subtype or encoding scheme attribute, specifying the FOR DATA clause or 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.

SPECIFIC FUNCTION *specific-name* **RESTRICT**

Identifies the function by its specific name, which was explicitly specified or implicitly created when the function was created. There must be exactly one function with *specific-name* in the implicitly or explicitly specified schema.

INDEX *index-name*

Identifies the index to be dropped. The name must identify a user-defined index that exists at the current server but must not identify a partitioning index, or a populated index on an auxiliary table. (For details on dropping user-defined

indexes on catalog tables, see “SQL statements allowed on the catalog” on page 915.) A partitioning index on table T can only be dropped by dropping the table space that contains T. A populated index on an auxiliary table can only be dropped by dropping the base table.

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 a unique index is dropped and that index was used to enforce the uniqueness of a parent key, the definition of the parent table is changed to incomplete. Otherwise, if the index was used to enforce a UNIQUE constraint, the definition of the table is not changed. The table can still be used, but the UNIQUE constraint implied by the index is no longer enforced.

If 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 an empty index on an auxiliary table is dropped, the base table is marked incomplete.

PACKAGE *collection-id.package-id*

Identifies the package version to be dropped. 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. A trigger package can only be dropped by dropping the associated trigger or triggering table.

Whenever the last or only version of a package is dropped, all privileges on the package are dropped and all plans that are dependent on the execute privilege of the package are invalidated.

version-id or **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 string 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 section on preparing an application program for execution in Section 6 of *DB2 Application Programming and SQL Guide*.

PROCEDURE *procedure-name* **RESTRICT**

Identifies the stored procedure to be dropped. The name must identify a stored procedure that has been defined with the CREATE PROCEDURE statement at the current server. The required keyword RESTRICT prevents the procedure from being dropped if a trigger definition contains a CALL statement with the name of the procedure.³⁶

³⁶ Dependencies can be checked only if the procedure name is specified as a literal and not via a host variable in the CALL statement.

When a procedure is directly or indirectly dropped, all privileges on the procedure are also dropped. In addition, any plans or packages that are dependent on the procedure are made inoperative.

STOGROUP *stogroup-name*

Identifies the storage group to be dropped. 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 on page 681.

SYNONYM *synonym*

Identifies the synonym to be dropped. 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 or alias that was defined using the synonym, nor does it invalidate any plans or packages that use such views or aliases.

TABLE *table-name*

Identifies the table to be dropped. 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, 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 can only be dropped by dropping the associated base table.

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.

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 an empty auxiliary table is dropped, the definition of the base table is marked incomplete.

TABLESPACE *database-name.table-space-name*

Identifies the table space to be dropped. 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.

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 table space in a TEMP database can be dropped only if it does not contain an active declared temporary table. A LOB table space can be dropped only if it does not contain an auxiliary table.

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.

TRIGGER *trigger-name*

Identifies the trigger to be dropped. 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.

VIEW *view-name*

Identifies the view to be dropped. 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.

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 PACKAGE or DROP TRIGGER (which implicitly drops the trigger package) cannot be performed while the package or trigger package is in use by an application. For applications that are bound with RELEASE(COMMIT), you can drop the package or trigger package at a commit point. For applications that are bound with RELEASE(DEALLOCATE), you can drop the package or trigger package only when the application's thread is deallocated.
- 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 44 on page 685.

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:

DROP

- 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 table space in a work file database: To drop a table space in a work file database, you must first issue the command -STOP DATABASE(*database-name*). Following your DROP, issue -START DATABASE(*database-name*). This process removes the table space you dropped from the pool of table spaces available to DB2.

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 Section 5 (Volume 2) of *DB2 Administration Guide* for details.

Dropping a temporary table: To drop a created temporary table or a declared temporary table, use the DROP TABLE statement.

Dropping an alias: Dropping a table or view does not drop its aliases. To drop an alias, use the DROP ALIAS statement.

|
|
|
|
|
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. 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. 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.

|
|
|
|
|
Table 43 on page 683 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 43. Effect of various DROP statements on auxiliary tables and indexes

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.

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 (R) 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, Table 44 on page 685 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 Table 44 on page 685, the 'D' in the VIEW column of the DROP TABLE row indicates that view B is dropped when table A is dropped. Next,

DROP

| 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.

Table 44. Effect of dropping objects that have dependencies

DROP statement	Type of object												
	ALIAS	DATABASE	DISTINCT TYPE	FUNCTION	INDEX	PACKAGE	PROCEDURE	STOGROUP	SYNONYM	TABLESPACE	TABLE	TRIGGER	VIEW
DROP ALIAS	-	-	-	-	-	V	-	-	-	-	-	-	-
DROP DATABASE	-	-	-	-	D ²	-	-	-	-	D	D	-	D
DROP DISTINCT TYPE	-	-	-	R ³	-	-	R ⁴	-	-	R	-	-	-
DROP FUNCTION	-	-	-	R ⁵	-	O	-	-	-	-	-	R	R
DROP INDEX ^{2,6}	-	-	-	-	-	V	-	-	-	-	-	V	-
DROP PACKAGE ⁷	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP PROCEDURE	-	-	-	-	-	O	-	-	-	-	-	R	-
DROP STOGROUP	-	-	-	-	R ⁸	-	-	-	-	-	R ⁸	-	-
DROP SYNONYM	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP TABLE ^{9,10}	-	-	-	-	D	V	-	-	D	-	-	D ¹¹	D
DROP TABLESPACE ¹²	-	-	-	-	D	V	-	-	-	D	-	-	-
DROP TRIGGER	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP VIEW	-	-	-	-	-	V	-	-	D	-	-	-	D

Legend:

- D** Dependent object is dropped.
- O** Dependent object is made inoperative.
- V** Dependent object is invalidated.
- R** DROP statement fails.

Notes:

1. The PACKAGE column represents packages for user-defined functions, procedures, and triggers, as well as other packages. The PACKAGE column also applies for plans.
2. The index space associated with the index is dropped.
3. If a function is dependent on the 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, the associated trigger and trigger package are also dropped.
12. A LOB table space cannot be dropped until the base table with the LOB columns is dropped. A table space in a TEMP database cannot be dropped if it contains an active declared temporary table.

DROP

Examples

Example 1: Drop table DSN8610.DEPT.

```
DROP TABLE DSN8610.DEPT;
```

Example 2: Drop table space DSN8S61D in database DSN8D61A.

```
DROP TABLESPACE DSN8D61A.DSN8S61D;
```

Example 3: Drop the view DSN8610.VPROJRE1:

```
DROP VIEW DSN8610.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 DISTINCT TYPE DOCUMENT RESTRICT;
```

| *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 RESTRICT;
```

| *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) RESTRICT;
```

| *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 RESTRICT;
```

| *Example 10:* Assume that you are SMITH and that stored procedure OSMOSIS is in schema BIOLOGY. Drop OSMOSIS.

```
| DROP PROCEDURE BIOLOGY.OSMOSIS RESTRICT;
```

| *Example 11:* Assume that you are SMITH and that trigger BONUS is in your schema. Drop BONUS.

```
| DROP TRIGGER BONUS;
```

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of a host variable declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax



►►—END DECLARE SECTION—◄◄

The diagram shows the statement 'END DECLARE SECTION' enclosed in a rectangular box. A horizontal line with arrowheads at both ends spans the width of the box, passing through the text.

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 a host variable declaration section. A host variable section starts with a BEGIN DECLARE SECTION statement described in “BEGIN DECLARE SECTION” on page 427.

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.
- BEGIN DECLARE SECTION and END DECLARE SECTION statements must be paired and must not be nested.
- Declare sections must not contain statements other than host variable declarations or SQL INCLUDE statements that include host variable declarations.

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 should not have the same name as variables declared within a declare section.

END DECLARE SECTION

Example

```
EXEC SQL BEGIN DECLARE SECTION;  
    (host variable declarations)  
EXEC SQL END DECLARE SECTION;
```

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

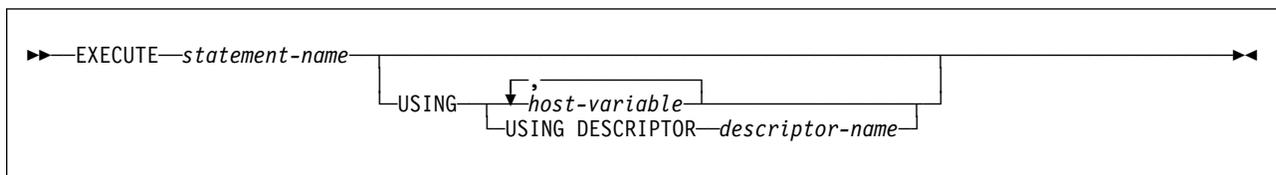
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

See “PREPARE” on page 757 for the authorization required to create a prepared statement.

Syntax



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 757.) 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 on page 690.

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. In the operational form of the clause, a reference to a structure is replaced by a reference to each of its variables. After all the replacements, 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 the EXECUTE statement is processed, the user must set the following fields in the SQLDA:

EXECUTE

#

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA

A REXX SQLDA does not contain this field.

- 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 results, 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 occurrences, see “SQL descriptor area (SQLDA)” on page 890.

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 907 for how to represent *descriptor-name* in C.

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 84. 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 on page 759.

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(VARS) bind options can cause some errors that are normally issued during PREPARE processing to be issued on EXECUTE.

Example

In this example, an INSERT statement with parameter markers is prepared and executed. S1 is a structure that corresponds to the format of DSN8610.DEPT.

```
EXEC SQL PREPARE DEPT_INSERT FROM  
      'INSERT INTO DSN8610.DEPT VALUES(?,?,?,?)';
```

(Check for successful execution and read values into S1)

```
EXEC SQL EXECUTE DEPT_INSERT USING :S1;
```

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a character string form of the statement
- Executes the SQL statement
- Destroys the executable form

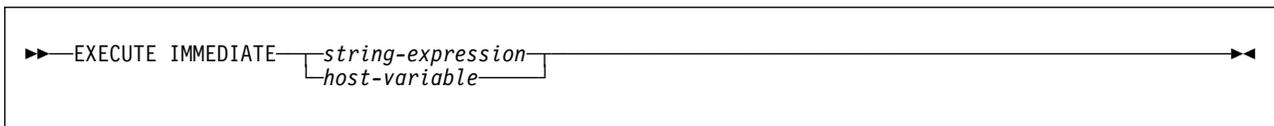
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

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 742 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

Syntax



Description

string-expression

string-expression is any PL/I expression that yields a character string. An optional colon can precede the *string-expression*. The colon introduces PL/I syntax. Therefore, host variables within a *string-expression* that includes operators or functions should not be preceded with a colon.

host-variable

For languages other than PL/I, *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 string variables. The host variable must not have a CLOB data type, and an indicator variable must not be specified. In Assembler language, C, and COBOL, the host variable must be a varying-length string variable. In C, it must not be a NUL-terminated string.

#

Notes

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:

ALTER	RENAME
COMMENT ON	REVOKE
COMMIT	ROLLBACK
CREATE	SET CURRENT DEGREE

	DELETE	SET CURRENT LOCALE LC_CTYPE
	DROP	SET CURRENT OPTIMIZATION HINT
	EXPLAIN	SET CURRENT PATH
	GRANT	SET CURRENT PRECISION
	INSERT	SET CURRENT RULES
	LABEL ON	SET CURRENT SQLID
	LOCK TABLE	UPDATE

The statement string must not include parameter markers or references to host variables, must not begin with EXEC SQL, and must not terminate with END-EXEC or a semicolon.

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.

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.

Example

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 DSN8610.DEPT
WHERE' || PREDs;
```

EXPLAIN

EXPLAIN

The information about this statement is Product-sensitive Programming Interface and Associated Guidance Information, as defined in Appendix H, “Notices” on page 1051.

The EXPLAIN statement obtains information about access path selection for an *explainable statement*. A statement is explainable if it is a SELECT or INSERT statement, or the searched form of an UPDATE or DELETE statement. The information obtained is placed in a user-supplied *plan table*.

Optionally, EXPLAIN can also obtain and place information in two 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.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

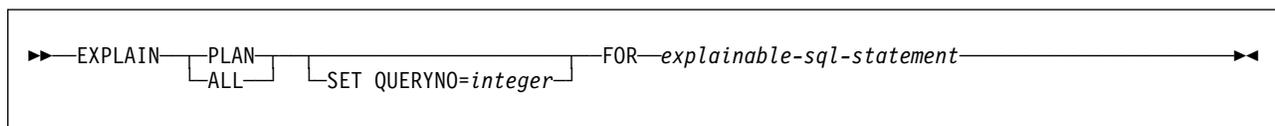
Authorization

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 authorization ID of the owner of the plan or package must also be the owner of a plan table named PLAN_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 must also be the owner of a plan table named PLAN_TABLE.

Syntax



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 Creating a plan table on page 697 and Table 45 on page 698 for the format and column descriptions of the plan table.

If a statement table exists, a row that provides a cost estimate of processing the explainable statement is inserted into the statement table. See Creating a statement table on page 703 and Table 46 on page 703 for the format and column descriptions 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 Creating a function table on page 704 and Table 47 on page 705 for the format and column descriptions of the function table.

ALL

Has the same effect as PLAN.

SET QUERYNO = *integer*

Associates *integer* with *explainable-sql-statement*. The column QUERYNO is given the value *integer* in every row inserted into the plan table, statement table, or function table by the EXPLAIN statement. If QUERYNO is not specified, DB2 itself assigns a number. For an embedded EXPLAIN statement, the number is the statement number that was assigned by the precompiler and placed in the DBRM.

FOR *explainable-sql-statement*

Specifies the SQL statement to be explained. *explainable-sql-statement* can be any explainable SQL statement. If EXPLAIN is embedded in a program, the statement can contain references to host variables. If EXPLAIN is dynamically prepared, the statement can contain parameter markers. Host variables that appear in the statement must be defined in the statement's program.

The statement must refer to objects at the current server.

explainable-sql-statement must not contain a QUERYNO clause. To specify the value of the QUERYNO column in plan table for the statement being explained, use the SET QUERYNO = clause of the EXPLAIN statement.

explainable-sql-statement cannot be a statement-name or a host-variable. To use EXPLAIN to get information about dynamic SQL statements, you must prepare the entire EXPLAIN statement dynamically.

To obtain information about an explainable SQL statement that references a
declared temporary table, the EXPLAIN statement must be executed in the
same application process in which the table was declared. For static EXPLAIN
statements, the information is not obtained at bind-time but at run-time when
the EXPLAIN statement is incrementally bound.

Notes

Output from EXPLAIN: Output from EXPLAIN is one or more rows of data inserted into the plan table. Rows are also inserted into the statement table and function table if the tables exist. The plan table must be created before the EXPLAIN statement is executed. Unless you need the information that the statement table or function table provides, it is not necessary to create either table to use EXPLAIN. The tables have the following names:

plan table	<i>userid</i> .PLAN_TABLE
statement table	<i>userid</i> .DSN_STATEMNT_TABLE
function table	<i>userid</i> .DSN_FUNCTION_TABLE

where *userid* is:

EXPLAIN

- The owner of the plan or package if the EXPLAIN statement is embedded in a plan or package.
- The SQL authorization ID of the process if the statement is dynamically prepared.

| For information on using the plan table and the statement table, see Section 5
| (Volume 2) of *DB2 Administration Guide*. For information on using the function
| table, see Section 4 of *DB2 Application Programming and SQL Guide*.

| **Output from BIND or REBIND:** DB2 can also add rows to a plan table, statement
| table, and function table when a plan or package is bound or rebound. This addition
| of rows occurs when the BIND or REBIND subcommand is executed with the
| EXPLAIN(YES) option in effect. The option requires that rows be 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.

| The plan table must exist when the BIND or REBIND subcommand is executed.
| Unless you need the information that the statement table or function table provides,
| neither table has to exist. Only the tables that exist receive new rows. The tables
| have the following names:

plan table	<i>userid</i> .PLAN_TABLE
statement table	<i>userid</i> .DSN_STATEMNT_TABLE
function table	<i>userid</i> .DSN_FUNCTION_TABLE

where *userid* is the owner of the plan or package.

Creating a plan table: To create a plan table, execute the following SQL statement:

```
CREATE TABLE userid.PLAN_TABLE
  (QUERYNO           INTEGER           NOT NULL,
   QBLOCKNO         SMALLINT          NOT NULL,
   APPLNAME         CHAR(8)           NOT NULL,
   PROGNAME         CHAR(8)           NOT NULL,
   PLANNO           SMALLINT          NOT NULL,
   METHOD            SMALLINT          NOT NULL,
   CREATOR          CHAR(8)           NOT NULL,
   TNAME            CHAR(18)          NOT NULL,
   TABNO            SMALLINT          NOT NULL,
   ACESSTYPE        CHAR(2)           NOT NULL,
   MATCHCOLS        SMALLINT          NOT NULL,
   ACCESSCREATOR    CHAR(8)           NOT NULL,
   ACCESSNAME       CHAR(18)          NOT NULL,
   INDEXONLY        CHAR(1)           NOT NULL,
   SORTN_UNIQ       CHAR(1)           NOT NULL,
   SORTN_JOIN       CHAR(1)           NOT NULL,
   SORTN_ORDERBY    CHAR(1)           NOT NULL,
   SORTN_GROUPBY    CHAR(1)           NOT NULL,
   SORTC_UNIQ       CHAR(1)           NOT NULL,
   SORTC_JOIN       CHAR(1)           NOT NULL,
   SORTC_ORDERBY    CHAR(1)           NOT NULL,
   SORTC_GROUPBY    CHAR(1)           NOT NULL,
   TSLOCKMODE       CHAR(3)           NOT NULL,
   TIMESTAMP        CHAR(16)          NOT NULL,
   REMARKS          VARCHAR(254)      NOT NULL,
   PREFETCH         CHAR(1)           NOT NULL WITH DEFAULT,
   COLUMN_FN_EVAL   CHAR(1)           NOT NULL WITH DEFAULT,
   MIXOPSEQ         SMALLINT          NOT NULL WITH DEFAULT,
   VERSION          VARCHAR(64)       NOT NULL WITH DEFAULT,
   COLLID           CHAR(18)          NOT NULL WITH DEFAULT,
   ACCESS_DEGREE    SMALLINT          ,
   ACCESS_PGROUP_ID SMALLINT          ,
   JOIN_DEGREE      SMALLINT          ,
   JOIN_PGROUP_ID   SMALLINT          ,
   SORTC_PGROUP_ID  SMALLINT          ,
   SORTN_PGROUP_ID  SMALLINT          ,
   PARALLELISM_MODE CHAR(1)           ,
   MERGE_JOIN_COLS  SMALLINT          ,
   CORRELATION_NAME CHAR(18)          ,
   PAGE_RANGE       CHAR(1)           NOT NULL WITH DEFAULT,
   JOIN_TYPE        CHAR(1)           NOT NULL WITH DEFAULT,
   GROUP_MEMBER     CHAR(8)           NOT NULL WITH DEFAULT,
   IBM_SERVICE_DATA VARCHAR(254)      NOT NULL WITH DEFAULT,
   WHEN_OPTIMIZE    CHAR(1)           NOT NULL WITH DEFAULT,
   QBLOCK_TYPE      CHAR(6)           NOT NULL WITH DEFAULT,
   BIND_TIME        TIMESTAMP         NOT NULL WITH DEFAULT,
   OPTHINT          CHAR(8)           NOT NULL WITH DEFAULT,
   HINT_USED        CHAR(8)           NOT NULL WITH DEFAULT,
   PRIMARY_ACESSTYPE CHAR(1)           NOT NULL WITH DEFAULT)
  IN database-name.table-space-name;
```

where *database-name.table-space-name* identifies a database and table space you have authorization to use.

EXPLAIN

Plan table column descriptions: Table 45 on page 698 explains the columns in PLAN_TABLE. The explanations apply both to rows resulting from the execution of an EXPLAIN statement and to rows resulting from a bind or rebind.

Each row in a plan table describes 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 involved, and the method used to carry out the step.

Table 45 (Page 1 of 5). Descriptions of columns in PLAN_TABLE

Column Name	Description
QUERYNO	<p>A number intended to identify the statement being explained. For a row produced by an EXPLAIN statement, specify the number in the QUERYNO clause. For a row produced by non-EXPLAIN statements, specify the number using the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE and DELETE statement syntax. Otherwise, DB2 assigns a number 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 greater than 32767 are reported as 0. Hence, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of TIMESTAMP is unique.</p>
QBLOCKNO	<p>The position of the query in the statement being explained (1 for the outermost query, 2 for the next query, and so forth). For better performance, DB2 might merge a query block into another query block. When that happens, the position number of the merged query block will not be in QBLOCKNO.</p>
APPLNAME	<p>The name of the application plan for the row. Applies only to embedded EXPLAIN statements executed from a plan or to statements explained when binding a plan. Blank if not applicable.</p>
PROGNAME	<p>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. Blank if not applicable.</p>
PLANNO	<p>The number of the step in which the query indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.</p>
METHOD	<p>A number (0, 1, 2, 3, or 4) that indicates the join method used for the step:</p> <ul style="list-style-type: none">0 First table accessed, continuation of previous table accessed, or not used.1 <i>Nested loop</i> join. For each row of the present composite table, matching rows of a new table are found and joined.2 <i>Merge scan</i> join. The present composite table and the new table are scanned in the order of the join columns, and matching rows are joined.3 Sorts needed by ORDER BY, GROUP BY, SELECT DISTINCT, UNION, a quantified predicate, or an IN predicate. This step does not access a new table.4 <i>Hybrid</i> join. The current composite table is scanned in the order of the join-column rows of the new table. The new table is accessed using list prefetch.
CREATOR	<p>The creator of the new table accessed in this step, blank if METHOD is 3.</p>

Table 45 (Page 2 of 5). Descriptions of columns in PLAN_TABLE

Column Name	Description
TNAME	The name of a table, created temporary table, declared temporary table, materialized view, table expression, or an intermediate result table for an outer join that is accessed in this step, blank if METHOD is 3. For an outer join, this column contains the created temporary table or the declared temporary table name of the work file in the form DSNWFQB(<i>qblockno</i>). Merged views show the base table names and correlation names. A materialized view is another query block with its own materialized views, tables, and so forth.
TABNO	Values are for IBM use only.
ACCESSTYPE	The method of accessing the new table: I By an index (identified in ACCESSCREATOR and ACCESSNAME) I1 By a one-fetch index scan N By an index scan when the matching predicate contains the IN keyword R By a table space scan M By a multiple index scan (followed by MX, MI, or MU) MX By an index scan on the index named in ACCESSNAME MI By an intersection of multiple indexes MU By a union of multiple indexes blank Not applicable to the current row
MATCHCOLS	For ACCESSTYPE I, I1, N, or MX, the number of index keys used in an index scan; otherwise, 0.
ACCESSCREATOR	For ACCESSTYPE I, I1, N, or MX, the creator of the index; otherwise, blank.
ACCESSNAME	For ACCESSTYPE I, I1, N, or MX, the name of the index; otherwise, blank.
INDEXONLY	Whether access to an index alone is enough to carry out the step, or whether data too must be accessed. Y=Yes; N=No. For exceptions, see Section 5 (Volume 2) of <i>DB2 Administration Guide</i> .
SORTN_UNIQ	Whether the new table is sorted to remove duplicate rows. Y=Yes; N=No.
SORTN_JOIN	Whether the new table is sorted for join method 2 or 4. Y=Yes; N=No.
SORTN_ORDERBY	Whether the new table is sorted for ORDER BY. Y=Yes; N=No.
SORTN_GROUPBY	Whether the new table is sorted for GROUP BY. Y=Yes; N=No.
SORTC_UNIQ	Whether the composite table is sorted to remove duplicate rows. Y=Yes; N=No.
SORTC_JOIN	Whether the composite table is sorted for join method 1, 2 or 4. Y=Yes; N=No.
SORTC_ORDERBY	Whether the composite table is sorted for an ORDER BY clause or a quantified predicate. Y=Yes; N=No.
SORTC_GROUPBY	Whether the composite table is sorted for a GROUP BY clause. Y=Yes; N=No.

EXPLAIN

Table 45 (Page 3 of 5). Descriptions of columns in PLAN_TABLE

Column Name	Description
TSLOCKMODE	<p>An indication of the mode of lock to be 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:</p> <p>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</p> <p>If the isolation cannot be determined at bind time, then the lock mode determined by the isolation at run time is shown by the following values.</p> <p>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.</p> <p>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.</p>
TIMESTAMP	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	A field into which you can insert any character string of 254 or fewer characters.
PREFETCH	Whether data pages are to be read in advance by prefetch. S = pure sequential prefetch; L = prefetch through a page list; blank = unknown or no prefetch.
COLUMN_FN_EVAL	When an SQL column 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	<p>The sequence number of a step in a multiple index operation.</p> <p>1, 2, ... n For the steps of the multiple index procedure (ACCESSTYPE is MX, MI, or MU.)</p> <p>0 For any other rows (ACCESSTYPE is I, I1, M, N, R, or blank.)</p>
VERSION	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. Blank if not applicable.
COLLID	The collection ID 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. Blank if not applicable.
<p>Note: The following nine columns, from ACCESS_DEGREE through CORRELATION_NAME, contain the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, each of them can contain null if the method it refers to does not apply.</p>	
ACCESS_DEGREE	The number of parallel tasks or operations activated by a query. This value is determined at bind time; the actual number of parallel operations used at execution time could be different. This column contains 0 if there is a host variable.
ACCESS_PGROUP_ID	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.

Table 45 (Page 4 of 5). Descriptions of columns in PLAN_TABLE

Column Name	Description
JOIN_DEGREE	The number of parallel operations or tasks used in joining the composite table with the new table. This value is determined at bind time and can be 0 if there is a host variable. The actual number of parallel operations or tasks used at execution time could be different.
JOIN_PGROUPE_ID	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.
SORTC_PGROUPE_ID	The parallel group identifier for the parallel sort of the composite table.
SORTN_PGROUPE_ID	The parallel group identifier for the parallel sort of the new table.
PARALLELISM_MODE	The kind of parallelism, if any, that is used at bind time: I Query I/O parallelism C Query CP parallelism X Sysplex query parallelism
MERGE_JOIN_COLS	The number of columns that are joined during a merge scan join (Method=2).
CORRELATION_NAME	The correlation name of a table or view that is specified in the statement. If there is no correlation name, then the column is blank.
PAGE_RANGE	Whether the table qualifies for page range screening, so that plans scan only the partitions that are needed. Y = Yes; blank = No.
JOIN_TYPE	The type of join. F FULL OUTER JOIN L LEFT OUTER 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	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	Values are for IBM use only.
WHEN_OPTIMIZE	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 reoptimized at run time using input variable values for input host variables, parameter markers, or special registers. The bind option REOPT(VARS) 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(VARS) must be specified for this to occur.

EXPLAIN

Table 45 (Page 5 of 5). Descriptions of columns in PLAN_TABLE

Column Name	Description																		
QBLOCK_TYPE	For each query block, the type of SQL operation performed. For the outermost query, it identifies the statement type. Possible values: <table border="0"> <tr><td>SELECT</td><td>SELECT</td></tr> <tr><td>INSERT</td><td>INSERT</td></tr> <tr><td>UPDATE</td><td>UPDATE</td></tr> <tr><td>DELETE</td><td>DELETE</td></tr> <tr><td>SELUPD</td><td>SELECT with FOR UPDATE OF</td></tr> <tr><td>DELCUR</td><td>DELETE WHERE CURRENT OF CURSOR</td></tr> <tr><td>UPDCUR</td><td>UPDATE WHERE CURRENT OF CURSOR</td></tr> <tr><td>CORSUB</td><td>Correlated subquery</td></tr> <tr><td>NCOSUB</td><td>Noncorrelated subquery</td></tr> </table>	SELECT	SELECT	INSERT	INSERT	UPDATE	UPDATE	DELETE	DELETE	SELUPD	SELECT with FOR UPDATE OF	DELCUR	DELETE WHERE CURRENT OF CURSOR	UPDCUR	UPDATE WHERE CURRENT OF CURSOR	CORSUB	Correlated subquery	NCOSUB	Noncorrelated subquery
SELECT	SELECT																		
INSERT	INSERT																		
UPDATE	UPDATE																		
DELETE	DELETE																		
SELUPD	SELECT with FOR UPDATE OF																		
DELCUR	DELETE WHERE CURRENT OF CURSOR																		
UPDCUR	UPDATE WHERE CURRENT OF CURSOR																		
CORSUB	Correlated subquery																		
NCOSUB	Noncorrelated subquery																		
BIND_TIME	The time at which the plan or package for this statement or query block was bound. For static SQL statements, this is a full-precision timestamp value. For dynamic SQL statements, this is the value contained in the TIMESTAMP column of PLAN_TABLE appended by 4 zeroes.																		
OPTHINT	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	If DB2 used one of your optimization hints, it puts the identifier for that hint (the value in OPTHINT) in this column.																		
PRIMARY_ACCESTYPE	Indicates whether direct row access will be attempted first: <table border="0"> <tr> <td>D</td> <td>DB2 will try to use direct row access. If DB2 cannot use direct row access at runtime, it uses the access path described in the ACCESTYPE column of PLAN_TABLE.</td> </tr> <tr> <td>blank</td> <td>DB2 will not try to use direct row access.</td> </tr> </table>	D	DB2 will try to use direct row access. If DB2 cannot use direct row access at runtime, it uses the access path described in the ACCESTYPE column of PLAN_TABLE.	blank	DB2 will not try to use direct row access.														
D	DB2 will try to use direct row access. If DB2 cannot use direct row access at runtime, it uses the access path described in the ACCESTYPE column of PLAN_TABLE.																		
blank	DB2 will not try to use direct row access.																		

Plan table creation options: A plan table can have a format with fewer columns than those shown in the foregoing CREATE statement. A plan table must include one of the following sets of columns:

- # • 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_PGROUPE_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 shown in the CREATE statement (COLCOUNT = 49)

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.

Missing columns are ignored when rows are added to a plan table.

Plan table migration: You can migrate existing plan tables to subsequent releases or fall back to prior releases. If you fall back to a prior release, the extra columns are simply ignored when EXPLAIN is executed. If you migrate to a subsequent release, the missing columns are likewise ignored.

Creating a statement table: To create a statement table, execute the following SQL statement:

```
CREATE TABLE userid.DSN_STATEMNT_TABLE
  (QUERYNO           INTEGER           NOT NULL WITH DEFAULT,
   APPLNAME          CHAR(8)           NOT NULL WITH DEFAULT,
   PROGNAME          CHAR(8)           NOT NULL WITH DEFAULT,
   COLLID            CHAR(18)          NOT NULL WITH DEFAULT,
   GROUP_MEMBER      CHAR(8)           NOT NULL WITH DEFAULT,
   EXPLAIN_TIME      TIMESTAMP         NOT NULL WITH DEFAULT,
   STMT_TYPE         CHAR(6)           NOT NULL WITH DEFAULT,
   COST_CATEGORY     CHAR(1)           NOT NULL WITH DEFAULT,
   PROCMS            INTEGER           NOT NULL WITH DEFAULT,
   PROCSU            INTEGER           NOT NULL WITH DEFAULT,
   REASON            VARCHAR(254)      NOT NULL WITH DEFAULT)
  IN database-name.table-space-name;
```

where *database-name.table-space-name* identifies a database and table space you have authorization to use.

Statement table column descriptions: Table 46 explains the columns in DSN_STATEMNT_TABLE. The explanations apply both to rows resulting from the execution of an EXPLAIN statement and to rows resulting from a bind or rebind.

Each row in the table provides a cost estimate, in service units and milliseconds, of processing an explainable statement.

Notice that the first five columns of the table are the same as the first five columns of PLAN_TABLE and DSN_FUNCTION_TABLE.

Table 46 (Page 1 of 2). Descriptions of columns in DSN_STATEMNT_TABLE

Column name	Description														
QUERYNO	A number intended to identify the statement being explained. See the description of the QUERYNO column in Table 45 on page 698 for more information. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.														
APPLNAME	The name of the application plan for the row, or blank. See the description of the APPLNAME column in Table 45 on page 698 for more information.														
PROGNAME	The name of the program or package containing the statement being explained, or blank. See the description of the PROGNAME column in Table 45 on page 698 for more information.														
COLLID	The collection ID for the package, or blank. See the description of the COLLID column in Table 45 on page 698 for more information.														
GROUP_MEMBER	The member name of the DB2 that executed EXPLAIN, or blank. See the description of the GROUP_MEMBER column in Table 45 on page 698 for more information.														
EXPLAIN_TIME	The time at which the statement is processed. This time is the same as the BIND_TIME column in PLAN_TABLE.														
STMT_TYPE	The type of statement being explained. Possible values are: <table border="0" style="margin-left: 20px;"> <tr> <td>SELECT</td> <td>SELECT</td> </tr> <tr> <td>INSERT</td> <td>INSERT</td> </tr> <tr> <td>UPDATE</td> <td>UPDATE</td> </tr> <tr> <td>DELETE</td> <td>DELETE</td> </tr> <tr> <td>SELUPD</td> <td>SELECT with FOR UPDATE OF</td> </tr> <tr> <td>DELCUR</td> <td>DELETE WHERE CURRENT OF CURSOR</td> </tr> <tr> <td>UPDCUR</td> <td>UPDATE WHERE CURRENT OF CURSOR</td> </tr> </table>	SELECT	SELECT	INSERT	INSERT	UPDATE	UPDATE	DELETE	DELETE	SELUPD	SELECT with FOR UPDATE OF	DELCUR	DELETE WHERE CURRENT OF CURSOR	UPDCUR	UPDATE WHERE CURRENT OF CURSOR
SELECT	SELECT														
INSERT	INSERT														
UPDATE	UPDATE														
DELETE	DELETE														
SELUPD	SELECT with FOR UPDATE OF														
DELCUR	DELETE WHERE CURRENT OF CURSOR														
UPDCUR	UPDATE WHERE CURRENT OF CURSOR														

EXPLAIN

Table 46 (Page 2 of 2). Descriptions of columns in DSN_STATEMNT_TABLE

Column name	Description
COST_CATEGORY	Indicates if DB2 was forced to use default values when making its estimates. Possible values: A Indicates that DB2 had enough information to make a cost estimate without using default values. B Indicates that some condition exists for which DB2 was forced to use default values. See the values in REASON to determine why DB2 was unable to put this estimate in cost category A.
PROCMS	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.
PROCSU	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.
REASON	A string that indicates the reasons for putting an estimate into cost category B. HOST VARIABLES The statement uses host variables, parameter markers, or special registers. TABLE CARDINALITY The cardinality statistics are missing for one or more of the tables used in the statement. UDF The statement uses user-defined functions. TRIGGERS Triggers are defined on the target table of an INSERT, UPDATE, or DELETE statement. REFERENTIAL CONSTRAINTS Referential constraints of the type CASCADE or SET NULL exist on the target table of a DELETE statement.

Creating a function table: To create a function table, execute the following SQL statement:

```
CREATE TABLE DSN_FUNCTION_TABLE
  (QUERYNO          INTEGER          NOT NULL WITH DEFAULT,
   QBLOCKNO        INTEGER          NOT NULL WITH DEFAULT,
   APPLNAME        CHAR(8)          NOT NULL WITH DEFAULT,
   PROGNAME        CHAR(8)          NOT NULL WITH DEFAULT,
   COLLID          CHAR(18)         NOT NULL WITH DEFAULT,
   GROUP_MEMBER    CHAR(8)          NOT NULL WITH DEFAULT,
   EXPLAIN_TIME    TIMESTAMP        NOT NULL WITH DEFAULT,
   SCHEMA_NAME     CHAR(8)          NOT NULL WITH DEFAULT,
   FUNCTION_NAME    CHAR(18)         NOT NULL WITH DEFAULT,
   SPEC_FUNC_NAME  CHAR(18)         NOT NULL WITH DEFAULT,
   FUNCTION_TYPE    CHAR(2)          NOT NULL WITH DEFAULT,
   VIEW_CREATOR    CHAR(8)          NOT NULL WITH DEFAULT,
   VIEW_NAME       CHAR(18)         NOT NULL WITH DEFAULT,
   PATH            VARCHAR(254)     NOT NULL WITH DEFAULT,
   FUNCTION_TEXT    VARCHAR(254)     NOT NULL WITH DEFAULT)
  IN database-name.table-space-name;
```

where *database-name.table-space-name* identifies a database and table space you have authorization to use.

Function table column descriptions: Table 47 on page 705 explains the columns in DSN_FUNCTION_TABLE. The explanations apply both to rows

resulting from the execution of an EXPLAIN statement and to rows resulting from a bind or rebind.

For each user-defined function that is referred to by the explainable statement, each row in the function table describes how DB2 resolved the function reference.

Notice that the first five columns of the table are the same as the first five columns of PLAN_TABLE and DSN_STATEMNT_TABLE.

Table 47. Descriptions of columns in DSN_FUNCTION_TABLE

Column name	Description
QUERYNO	A number intended to identify the statement being explained. See the description of the QUERYNO column in Table 45 on page 698 for more information. If QUERYNO is not unique, the value of EXPLAIN_TIME is unique.
APPLNAME	The name of the application plan for the row, or blank. See the description of the APPLNAME column in Table 45 on page 698 for more information.
PROGNAME	The name of the program or package containing the statement being explained, or blank. See the description of the PROGNAME column in Table 45 on page 698 for more information.
COLLID	The collection ID for the package, or blank. See the description of the COLLID column in Table 45 on page 698 for more information.
GROUP_MEMBER	The member name of the DB2 that executed EXPLAIN, or blank. See the description of the GROUP_MEMBER column in Table 45 on page 698 for more information.
EXPLAIN_TIME	The time at which the statement is processed. This time is the same as the BIND_TIME column in PLAN_TABLE.
SCHEMA_NAME	The schema name of the function invoked in the explained statement.
FUNCTION_NAME	The name of the function invoked in the explained statement.
SPEC_FUNC_ID	The specific name of the function invoked in the explained statement.
FUNCTION_TYPE	The type of function invoked in the explained statement. Possible values are: S Scalar function T Table function
VIEW_CREATOR	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the creator of the view. Otherwise, blank.
VIEW_NAME	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the name of the view. Otherwise, blank.
PATH	The value of the SQL path that was used to resolve the schema name of the function.
FUNCTION_TEXT	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 /.

FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host variables.

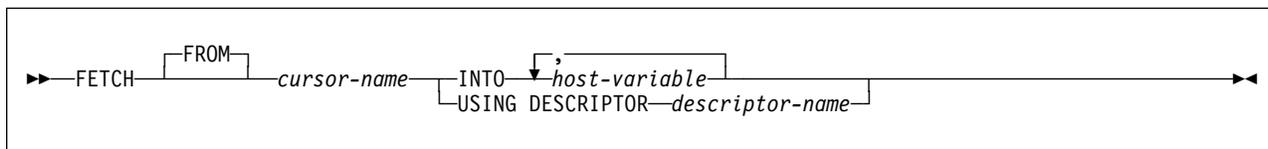
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

See “DECLARE CURSOR” on page 634 for an explanation of the authorization required to use a cursor.

Syntax



Description

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 “Notes” on page 636, or an allocated cursor, as explained in “ALLOCATE CURSOR” on page 346. When the FETCH statement is executed, the cursor must be in the open state.

If the cursor is currently positioned on or after the last row of its result table, the SQLCODE field of the SQLCA is set to +100, SQLSTATE is set to '02000', the cursor is positioned after the last row, and values are not assigned to host variables.

If the cursor is currently positioned before a row, the cursor is positioned on that row and values are assigned to host variables as specified by INTO or USING.

If the cursor is currently positioned on a row other than the last row, the cursor is positioned on the next row and values of that row are assigned to host variables as specified by INTO or USING.

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. In the operational form of INTO, 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.

FETCH

USING 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, 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.

- 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 set is to be assigned. If LOBs are present in the results, there must be additional SQLVAR entries for each column of the result table. 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 890.

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 907 for how to represent *descriptor-name* in C.

Notes

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 76. 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 3. Language elements” on page 43. 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.

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 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.

Cursor positioning: An open cursor has three possible positions:

- Before a row
- On a row
- After the last row

If a cursor is on a row, that row is called the current row of the cursor. A cursor referred to in an UPDATE or DELETE statement must be positioned on a row. A cursor can only be on a row as a result of a FETCH statement.

The current row of a cursor cannot be updated or deleted by another application process if it is locked or a temporary copy of a result table was created when the cursor was opened. 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

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.

Example

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 DSN8610.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;
```

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.

Authorization

None required.

Syntax

```

▶ FREE LOCATOR 'host_variable', ...

```

Description

host_variable,...

Identifies a *host-variable* locator variable that must have been previously declared according to the rules for declaring host-variable 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.

After the FREE LOCATOR statement is executed, each locator variable in the host-variable list is no longer associated with the string value it represented.

If a locator variable is not an established locator within the current unit of work, an invalid locator error occurs. When this error occurs and more than one host variable was specified in the FREE LOCATOR statement, only the locators up to the first invalid locator are freed. Locators listed after the first invalid locator are not 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
```

GRANT

The 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
- 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:

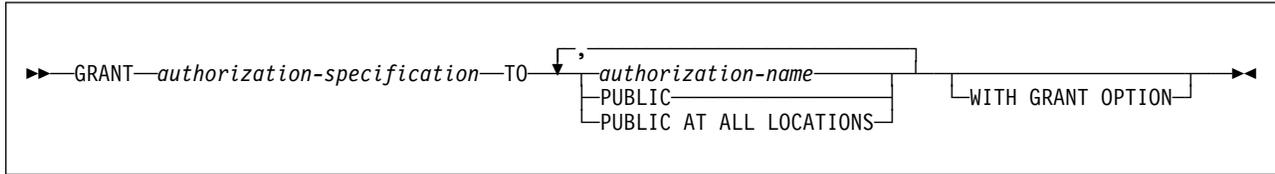
- DBADM on databases
- DELETE, INSERT, SELECT, and UPDATE on user tables or views
- EXECUTE on plans, packages, functions, or stored procedures
- PACKADM on collections
- SYSADM authority

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 627 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 authorization ID of 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.

GRANT

Syntax



Description

authorization-specification

Names one or more privileges in one of the formats described below. 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.

The value of CURRENT RULES determines whether you can use the ID of the GRANT statement itself (to grant privileges to yourself). When CURRENT RULES is:

DB2 You cannot use the ID of the GRANT statement.

STD You can use the ID of the GRANT statement.

PUBLIC

Grants the privileges to all users at the current server, including application requesters using DRDA access.

PUBLIC AT ALL LOCATIONS

Grants the privileges to all users in the network. Applies to table privileges only, excluding ALTER, INDEX, REFERENCES, and TRIGGER.

PUBLIC AT ALL LOCATIONS applies to DB2 private protocol access only.

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 or to PUBLIC AT ALL LOCATIONS. When WITH GRANT OPTION is used with either of these, a warning is issued, and the named privileges are granted, but without GRANT authority.

Notes

For more on DB2 privileges, read Section 3 (Volume 1) of *DB2 Administration Guide*. For information on access control authorization exits, see Appendix B (Volume 2) of *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 the authorization ID of the owner of the plan or package. The grantee, as recorded in the catalog, is an authorization ID, PUBLIC, or PUBLIC*, where PUBLIC* denotes PUBLIC AT ALL LOCATIONS.

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.

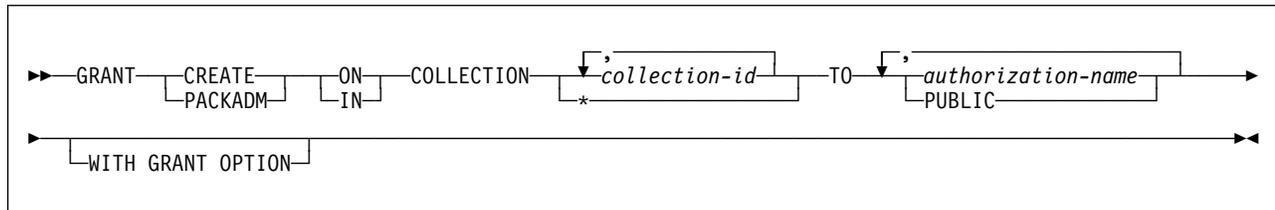
Whenever a grant is made for a database, distinct type, package, plan, schema, stored procedure, table, trigger, user-defined function, view, or USE privilege for an object that does not exist, an SQL return code is issued and the grant is not made.

GRANT (collection privileges)

GRANT (collection privileges)

This form of the GRANT statement grants privileges on collections.

Syntax



Description

CREATE IN

Grants the privilege to use the BIND subcommand to create packages in the designated collections.

The word ON can be used instead of IN.

PACKADM ON

Grants package administrator authority for the designated collections.

The word IN can be used instead of ON.

COLLECTION *collection-id*,...

Identifies the collections on which the specified privilege is granted. The collections do not have to exist.

COLLECTION *

Indicates that the specified privilege is granted on all collections including those that do not currently exist.

TO

Refer to “GRANT” on page 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 for a description of the WITH GRANT OPTION clause.

Example

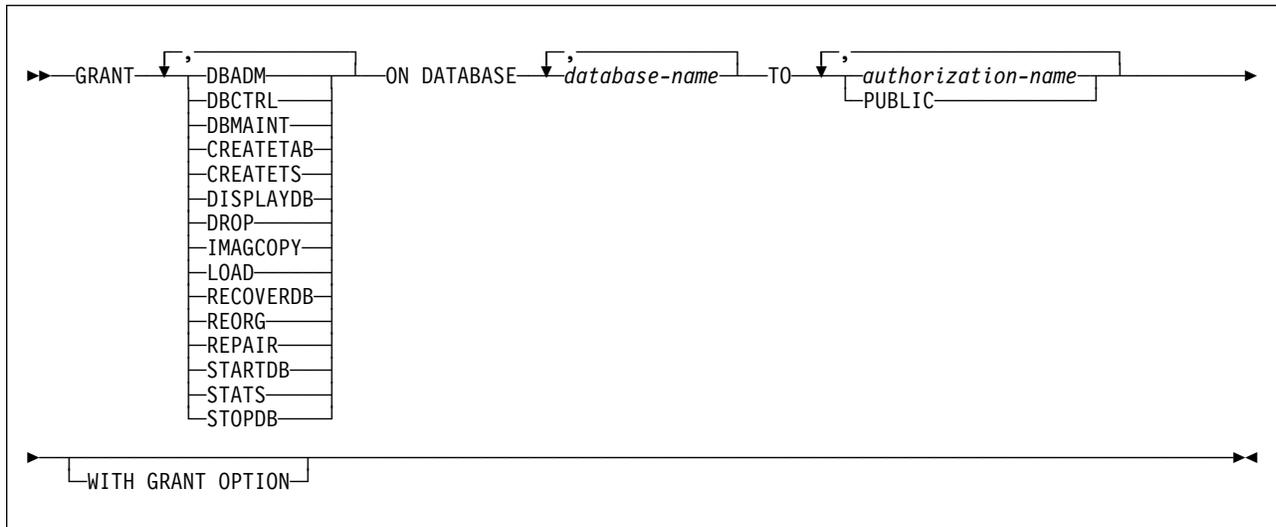
Grant the privilege to create new packages in collections QACLONE and DSN8CC61 to CLARK.

```
GRANT CREATE IN COLLECTION QACLONE, DSN8CC61 TO CLARK;
```

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. For a TEMP 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 utility.

GRANT (database privileges)

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, and the CHECK utility to test whether indexes are consistent with the data they index.

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.

TO

Refer to “GRANT” on page 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant drop privileges on database DSN8D61A to user PEREZ.

```
GRANT DROP
  ON DATABASE DSN8D61A
  TO PEREZ;
```

Example 2: Grant repair privileges on database DSN8D61A to all local users.

```
GRANT REPAIR
  ON DATABASE DSN8D61A
  TO PUBLIC;
```

Example 3: Grant authority to create new tables and load tables in database DSN8D61A to users WALKER, PIANKA, and FUJIMOTO, and give them grant privileges.

```
GRANT CREATETAB,LOAD  
ON DATABASE DSN8D61A  
TO WALKER,PIANKA,FUJIMOTO  
WITH GRANT OPTION;
```

GRANT (distinct type privileges)

GRANT (distinct type privileges)

This form of the GRANT statement grants the privilege to use distinct types (user-defined data types).

Syntax

```
GRANT USAGE ON DISTINCT TYPE (1) distinct-type-name TO authorization-name
PUBLIC
WITH GRANT OPTION
```

Note:

¹ DATA can be used as a synonym for DISTINCT. DISTINCT is the keyword that is used on CREATE.

Description

USAGE

Grants the privilege to use the identified distinct types.

DISTINCT 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. If you do not explicitly qualify the distinct type name, it 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 when the plan or package was created or last rebound. If QUALIFIER was not used, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

TO

Refer to “GRANT” on page 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 for a description of the WITH GRANT OPTION clause.

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 DISTINCT 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 DISTINCT 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.

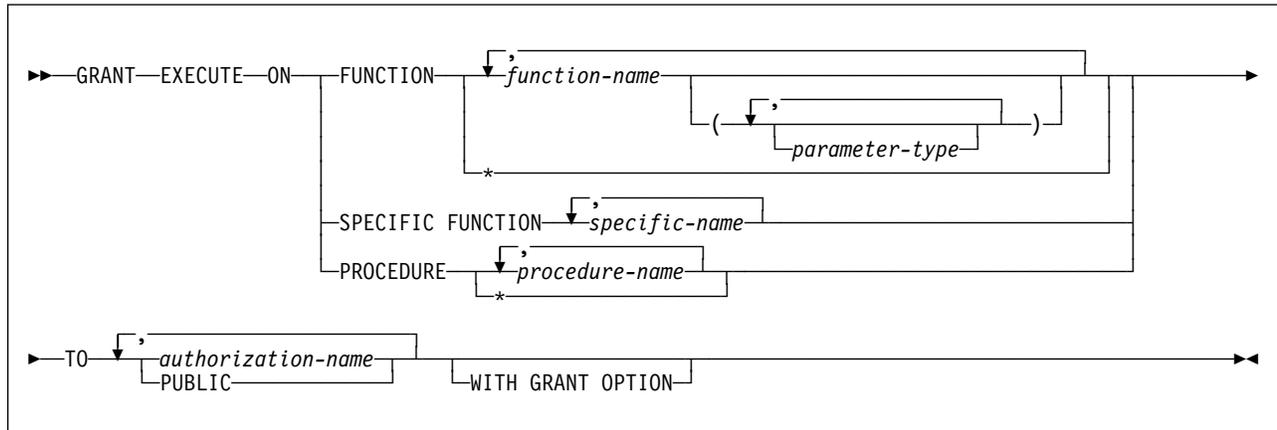
```
GRANT USAGE ON DISTINCT TYPE CANADIAN_DOLLAR TO ADMIN_A
WITH GRANT OPTION;
```

GRANT (function or procedure privileges)

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:

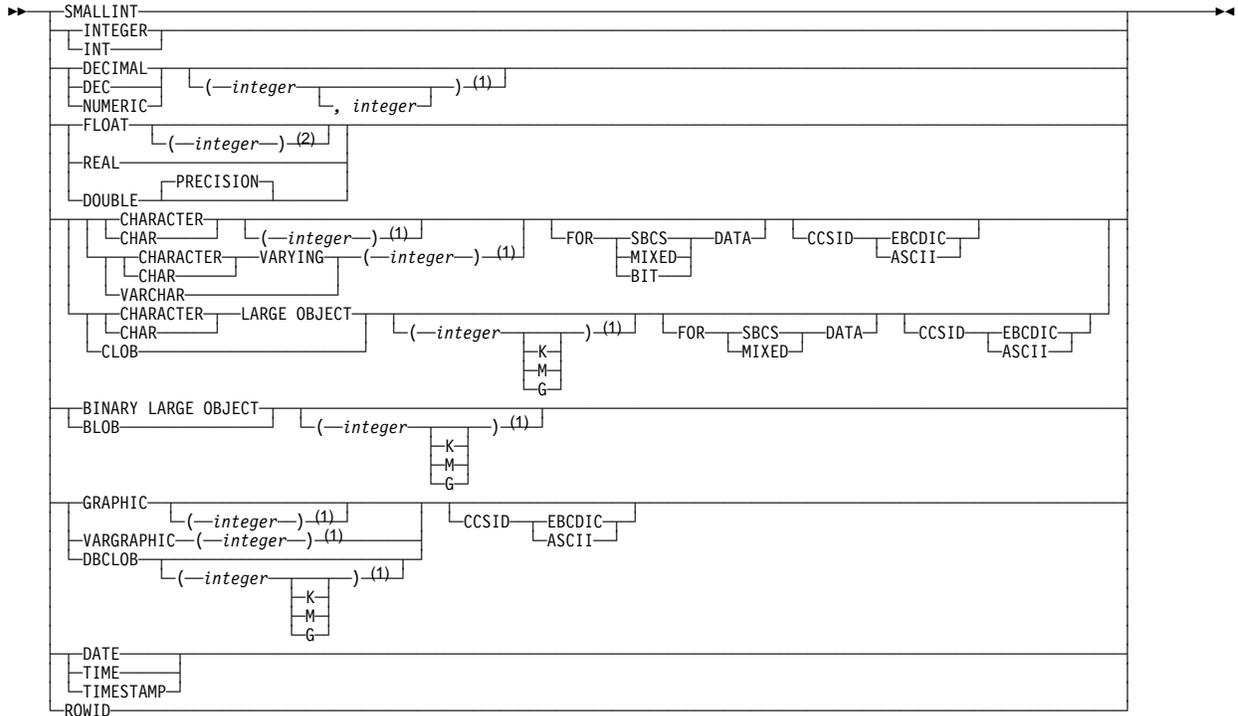
`data-type` [AS LOCATOR⁽¹⁾]

Note:

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-data-type`
`distinct-type-name`

built-in data type:**Notes:**

- ¹ The values that are specified for length, precision, or scale attributes must match the values that were specified when the function was created. Coding specific values is optional. Empty parentheses, (), can be used instead to indicate that DB2 ignores the attributes when determining whether data types match.
- ² The value that is specified does not have to match the value that was specified when the function was created because matching is based on data type (REAL or DOUBLE). $1 \leq integer \leq 21$ indicates REAL and $22 \leq integer \leq 53$ indicates DOUBLE. Coding a specific value is optional. Empty parentheses cannot be used.

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 DISTINCT TYPE statement.

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.

GRANT (function or procedure privileges)

FUNCTION *function-name*

Identifies the function by its name. You can identify a function by its name only if there is exactly one function with *function-name* in the schema. If you do not explicitly qualify the function name with a schema name, the function name 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 when the plan or package was created or last rebound. If QUALIFIER was not used, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

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. 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.

function-name

Specifies 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 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 is used to uniquely identify the function.

For data types that have a length, precision, or scale attribute, you can specify a value or use a set of empty parentheses:

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match.

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 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. For example:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. For a complete list of the default lengths of data types, see “CREATE TABLE” on page 570.

For data types with a subtype or encoding scheme attribute, specifying the FOR DATA clause or 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.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name.

PROCEDURE *procedure-name*

Identifies a stored procedure that is defined at the current server. If you do not explicitly qualify the procedure name with a schema name, the procedure name 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 when the plan or package was created or last rebound. If QUALIFIER was not used, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

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 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 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;
```

GRANT (function or procedure privileges)

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.

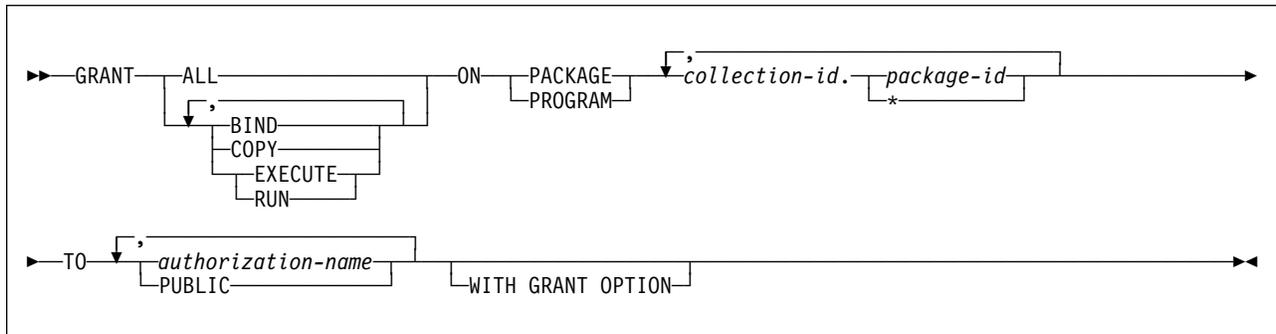
```
GRANT EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10))  
TO HR;
```

You can also code the CHAR(10) data type as CHAR().

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 is used 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 726.

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-id,...*

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-id* 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

GRANT (package privileges)

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.

The word PROGRAM can be used in place of PACKAGE.

TO

Refer to “GRANT” on page 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 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 primary authorization ID 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 primary authorization ID 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
- BIND package privilege (used only to add a new version of an existing package)

Examples

Example 1: Grant the privilege to copy all packages in collection DSN8CC61 to LEWIS.

```
GRANT COPY ON PACKAGE DSN8CC61.* TO LEWIS;
```

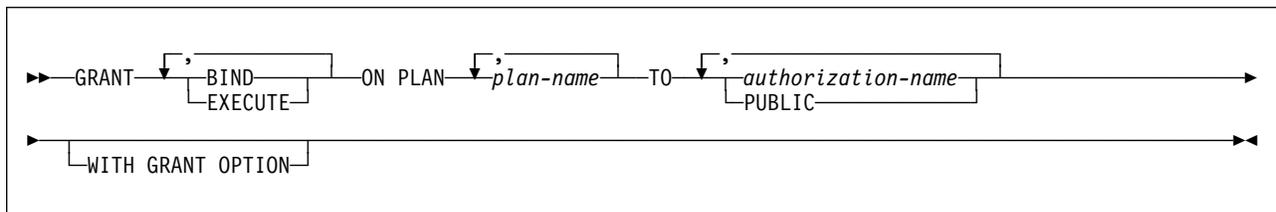
Example 2: You have the BIND privilege with GRANT authority over the package CLCT1.PKG1. You have the EXECUTE privilege with GRANT authority over the package CLCT2.PKG2. You have no other privileges with GRANT authority over any package in the collections CLCT1 AND CLCT2. Hence, the following statement, 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;
```

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 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 for a description of the WITH GRANT OPTION clause.

Examples

Example 1: Grant the privilege to bind plan DSN8IP61 to user JONES.

```
GRANT BIND ON PLAN DSN8IP61 TO JONES;
```

Example 2: Grant privileges to bind and execute plan DSN8CP61 to all users at the current server.

```
GRANT BIND,EXECUTE ON PLAN DSN8CP61 TO PUBLIC;
```

Example 3: Grant the privilege to execute plan DSN8CP61 to users ADAMSON and BROWN with grant option.

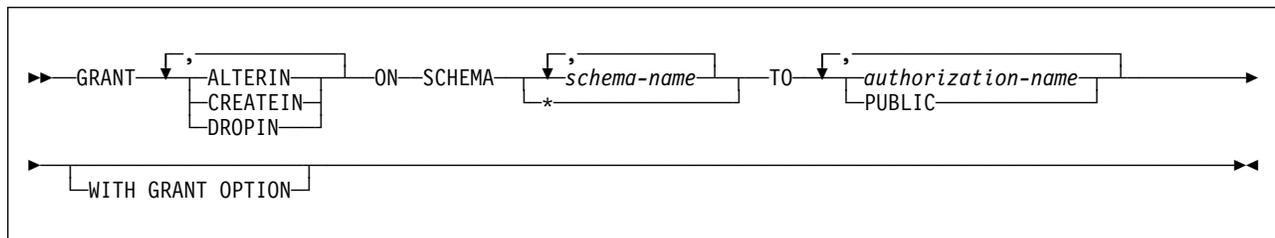
```
GRANT EXECUTE ON PLAN DSN8CP61 TO ADAMSON,BROWN WITH GRANT OPTION;
```

GRANT (schema privileges)

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, stored procedures, triggers, and user-defined functions in the designated schemas.

CREATEIN

Grants the privilege to create distinct types, stored procedures, triggers, and user-defined functions in the designated schemas.

DROPIN

Grants the privilege to drop distinct types, 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 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 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).

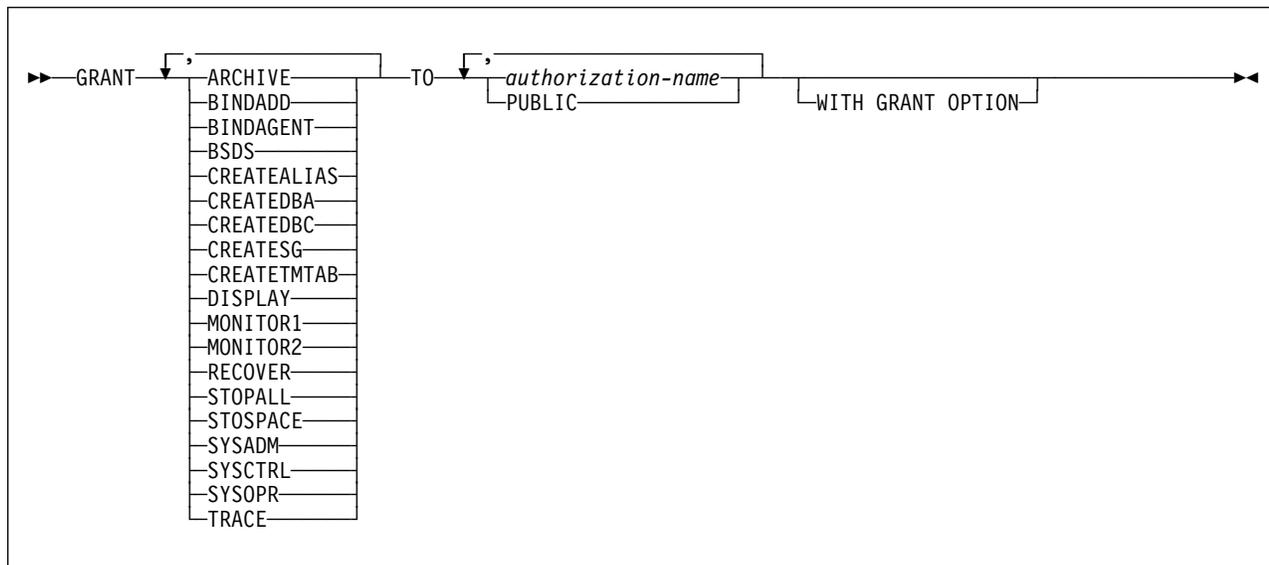
```
| GRANT CREATEIN, ALTERIN, DROPIN ON SCHEMA NEW_HIRE, PROMO, RESIGN TO HR;
```

GRANT (system privileges)

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

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 Section 3 (Volume 1) of *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 Section 3 (Volume 1) of *DB2 Administration Guide*.

SYSOPR

Grants the privilege to have system operator authority.

GRANT (system privileges)

TRACE

Grants the privilege to issue the MODIFY TRACE, START TRACE, and STOP TRACE commands.

TO

Refer to “GRANT” on page 711 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;
```

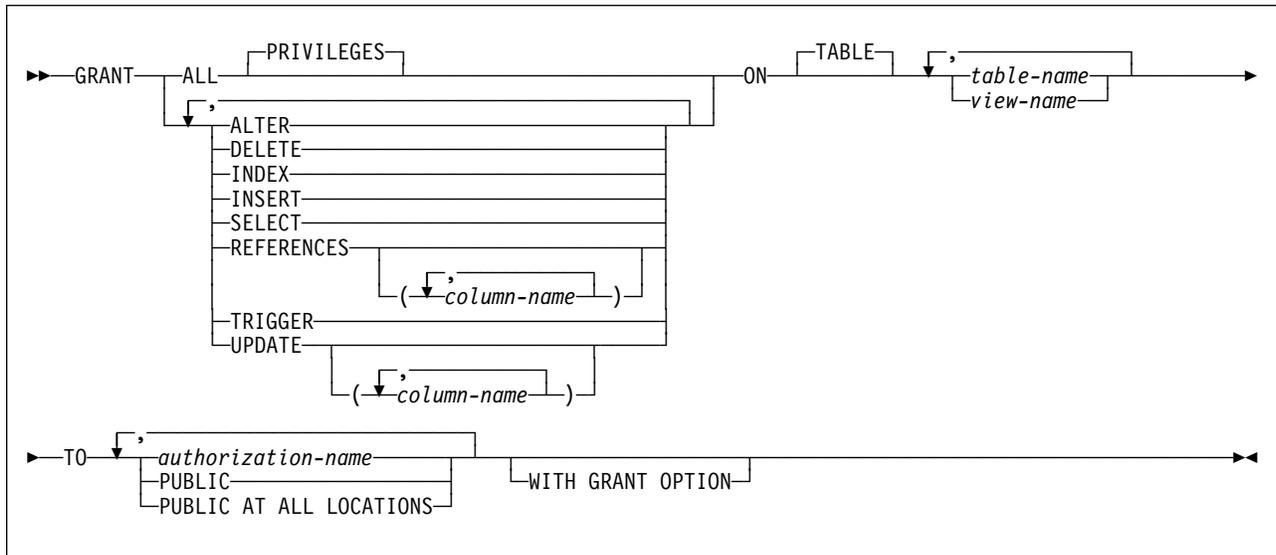
Example 3: Grant TRACE privileges to all local users.

```
GRANT TRACE  
TO PUBLIC;
```

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. Does not include ALTER, INDEX, REFERENCES, or TRIGGER for a grant to PUBLIC AT ALL LOCATIONS.

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 use the ALTER TABLE statement. ALTER cannot be granted to PUBLIC AT ALL LOCATIONS. Nor can it be used if the statement identifies an auxiliary table or a view.

DELETE

Grants the privilege to use the DELETE statement. DELETE cannot be granted on an auxiliary table.

INDEX

Grants the privilege to use the CREATE INDEX statement. INDEX cannot be granted to PUBLIC AT ALL LOCATIONS. Nor can it be used if the statement identifies a view.

INSERT

Grants the privilege to use the INSERT statement. INSERT cannot be granted on an auxiliary table.

GRANT (table or view privileges)

REFERENCES(*column-name*,...)

Grants the privilege to define and drop a referential constraint in which the table is a parent. Grantees can create referential constraints by using all the named columns in the parent key. If a list of columns is not specified, REFERENCES applies to all the columns of every table identified in the ON clause. REFERENCES cannot be granted to PUBLIC AT ALL LOCATIONS. Nor can it be used if the statement identifies an auxiliary table or a view.

If you specify a list of columns, each *column-name* must be the unqualified name of a column in a table identified in the ON clause.

SELECT

Grants the privilege to use the SELECT statement. SELECT cannot be granted on an auxiliary table.

TRIGGER

Grants the privilege to use the CREATE TRIGGER statement. TRIGGER cannot be granted to PUBLIC AT ALL LOCATIONS. Nor can it be used if the statement identifies an auxiliary table or a view.

UPDATE

Grants the privilege to use the UPDATE statement. UPDATE cannot be granted on an auxiliary table.

UPDATE(*column-name*,...)

Grants the privilege to use the UPDATE statement 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 or ON TABLE

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 must not be identified.

If you use GRANT ALL, then for each named table or view, the privilege set (described in “Authorization” in “GRANT” on page 711) must include at least one privilege with the GRANT option.

TO

Refer to “GRANT” on page 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 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.

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.

Examples

Example 1: Grant SELECT privileges on table DSN8610.EMP to user PULASKI.

```
GRANT SELECT ON DSN8610.EMP TO PULASKI;
```

Example 2: Grant UPDATE privileges on columns EMPNO and WORKDEPT in table DSN8610.EMP to all users at the current server.

```
GRANT UPDATE (EMPNO,WORKDEPT) ON TABLE DSN8610.EMP TO PUBLIC;
```

Example 3: Grant all privileges on table DSN8610.EMP to users KWAN and THOMPSON, with the WITH GRANT OPTION.

```
GRANT ALL ON TABLE DSN8610.EMP TO KWAN,THOMPSON WITH GRANT OPTION;
```

Example 4: Grant the SELECT and UPDATE privileges on the table DSN8610.DEPT to every user in the network.

```
GRANT SELECT, UPDATE ON TABLE DSN8610.DEPT  
TO PUBLIC AT ALL LOCATIONS;
```

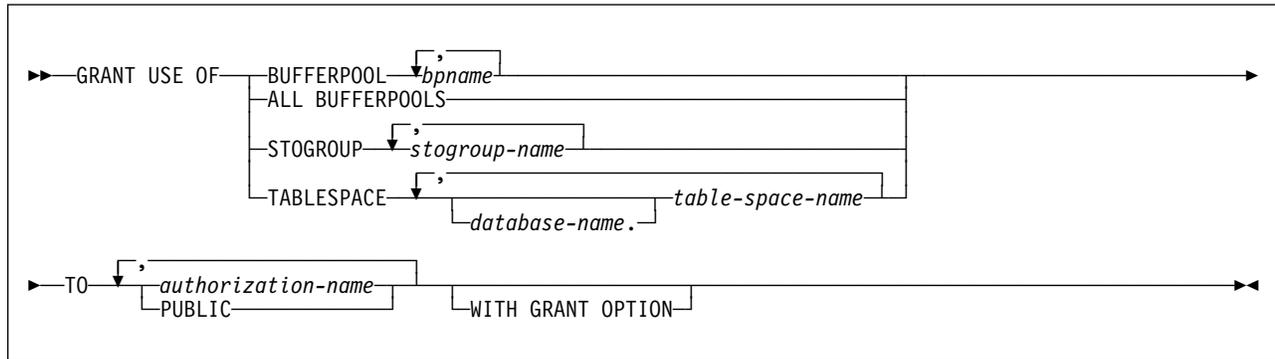
Even with this grant, it is possible that some network users do not have access to the table at all, or to any other object at the table's subsystem. Controlling access to the subsystem involves the communications databases at the subsystems in the network. The tables for the communication databases are described in Appendix D, "DB2 catalog tables" on page 911. Controlling access is described in Section 3 (Volume 1) of *DB2 Administration Guide*.

GRANT (use privileges)

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 50 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 tables spaces that are for declared temporary
tables (table spaces in the TEMP 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 711 for a description of the TO clause.

WITH GRANT OPTION

Refer to “GRANT” on page 711 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 DSN8S61D in database DSN8D61A.

```
GRANT USE OF TABLESPACE  
DSN8D61A.DSN8S61D  
TO PUBLIC;
```

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

```

▶ HOLD LOCATOR , host_variable ▶
  
```

Description

host_variable,...

Identifies a *host-variable* locator variable that must have been previously declared according to the rules for declaring host 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.

After the HOLD LOCATOR statement is executed, each locator variable in the host-variable list has the *hold* property.

If a locator variable is not an established locator within the current unit of work, an invalid locator error occurs. When this error occurs and more than one host variable was specified in the HOLD LOCATOR statement, only the locators up to the first invalid locator are held. Locators listed after the first invalid locator are not 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

INCLUDE

The INCLUDE statement inserts declarations or code into a source program.

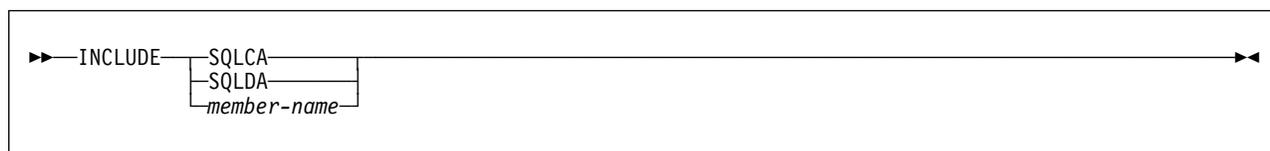
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

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. Do not use the INCLUDE statement with REXX.

For a description of the SQLCA, see “SQL communication area (SQLCA)” on page 883.

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 890.

member-name

Names a member of the partitioned data set to be the library input when your application program is precompiled. It must be a short, ordinary 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. 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 also inserts the row into the table on which the view is based.

There are two forms of this statement:

- The INSERT via VALUES is used to insert a single row into the table or view using the values provided or referenced.
- The INSERT via SELECT is used to insert one or more rows into the table or view using values from other tables, or views, or both.

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

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

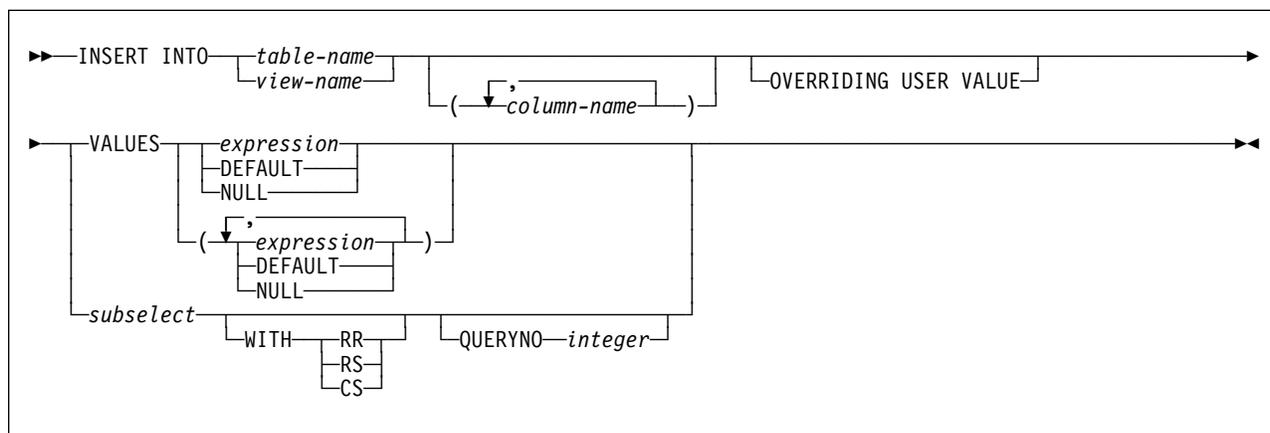
The owner of a view, unlike the owner of a table, might not have INSERT authority on the view (or can have INSERT authority without being able to grant it to others). The nature of the view itself can preclude its use for INSERT. For more information, see the discussion of authority in “CREATE VIEW” on page 627.

If an expression that refers to a function is specified, the privilege set must include any authority that is necessary to execute the function.

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 309.

If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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 29 on page 342. (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 61.)

Syntax



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 DB2 subsystem identified by the implicitly or explicitly specified location name. The name must not identify:

- An auxiliary table
- A catalog table for which inserts 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 627.)

A value cannot be inserted into a view column that is derived from:

- A constant, expression, or scalar function
- The same base table column as some other column of the view

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. In an IMS or CICS application, the DB2 subsystem that contains the identified table or view must not be a remote DB2 Version 2 Release 3 subsystem.

column-name,...

Specifies the columns for which insert values are provided. Each name must be an unqualified name that identifies 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 once. A view column that cannot accept insert values must not be identified.

Omission of the column list is an implicit specification of a list in which every column of the table or view is identified in left-to-right order. This list is established when the statement is bound and therefore does not include columns that were added to the table after the statement was bound.

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.

OVERRIDING USER VALUE

Specifies that the value specified in the VALUES clause or produced by a subselect for a column that is defined as GENERATED ALWAYS is ignored. Instead, a system-generated value is inserted, overriding the user-specified value.

Specify OVERRIDING USER VALUE only if the insert involves a column defined as GENERATED ALWAYS, such as a ROWID column or an identity column.

VALUES

Specifies one new row in the form of a list of values. The number of values in the VALUES clause must equal the number of names in the column list. 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.

expression

Any expression of the type described in “Expressions” on page 131. The expression must not include a column name. 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.

DEFAULT

The default value assigned to the column. If the column is a ROWID column or an identity column, DB2 will generate a unique 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 570.

NULL

The null value.

For a ROWID or an identity 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 (without a unique index, DB2 can guarantee unique values only among
the set of system-generated values).

subselect

Specifies a set of new rows in the form of the result table of a subselect. If the result table is empty, SQLCODE is set to +100, and SQLSTATE is set to '02000'.

(For an explanation of subselect, see “Chapter 5. Queries” on page 307.)

The number of columns in the result table must equal the number of names in the column list. 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 ROWID or identity column must conform to the rules that are described for those columns under the VALUES clause.

If the object table is self-referencing, the subselect must not return more than one row.

WITH

Specifies the isolation level at which the subselect is executed.

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 columns of the plan tables for the rows that contain information about this SQL statement.

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, if hints are used. For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, see Section 5 (Volume 2) of *DB2 Administration Guide*.

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 3. Language elements” on page 43.
- *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.
- *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 627.

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.

Number of rows inserted: Normally, after an INSERT statement completes execution, the value of SQLERRD(3) in SQLCA is the number of rows inserted. (For a complete description of the SQLCA, including exceptions to the above, see “SQL communication area (SQLCA)” on page 883.

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 statement. 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 catalog table SYSIBM.SYSSTRINGS: If the object table is SYSIBM.SYSSTRINGS, only certain values can be specified, as described in Appendix B (Volume 2) of *DB2 Administration Guide*.

Datetime representation when using datetime registers: As explained under Datetime special registers on page 105, 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.

Examples

Example 1: Insert values into sample table DSN8610.EMP.

```
INSERT INTO DSN8610.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 DSN8610.EMP.

```
INSERT INTO SMITH.TEMPEMPL
SELECT *
FROM DSN8610.EMP;
```

Example 3: Assume that SESSION.TEMPEMPL is a declared temporary table. Populate the table with data from department D11 in sample table DSN8610.EMP.

```
INSERT INTO SESSION.TEMPEMPL
SELECT *
FROM DSN8610.EMP
WHERE WORKDEPT='D11';
```

Example 4: Insert a row into sample table DSN8610.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 DSN8610.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.

INSERT

For example, assume that you want to copy the rows in DSN8610.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 DSN8610.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 DSN8610.EMP_PHOTO_RESUME;
```

LABEL ON

The LABEL ON 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.

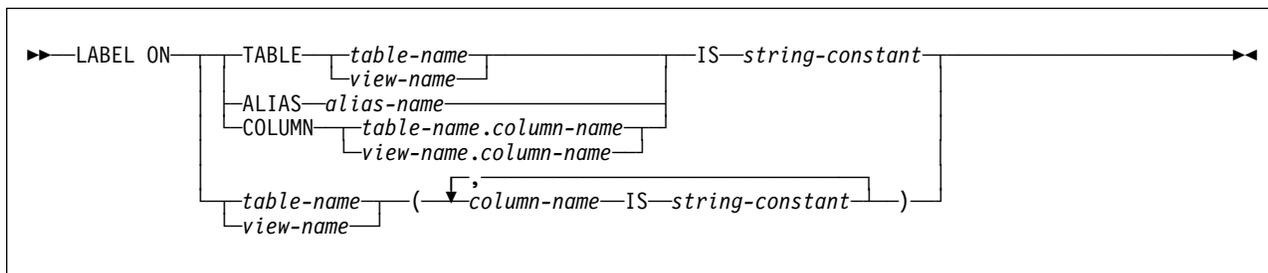
Authorization

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

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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 29 on page 342. (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 61.)

Syntax



Description

TABLE

Indicates that the label is for a table or a view.

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

Identifies the alias to which the comment applies.

#

LABEL ON

alias-name

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

Indicates that the label is for a 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,...
```

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 DSN8610.DEPT.

```
LABEL ON COLUMN DSN8610.DEPT.DEPTNO  
IS 'DEPARTMENT NUMBER';
```

Example 2: Enter labels on two columns in table DSN8610.DEPT.

```
LABEL ON DSN8610.DEPT  
(MGRNO IS 'MANAGER'S EMPLOYEE NUMBER',  
ADMDEPT 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
- Ownership of the table
- DBADM 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 authorization ID of 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 29 on page 342. (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 61.)

Syntax

```

▶▶ LOCK TABLE table-name [PART integer] IN [SHARE | EXCLUSIVE] MODE ▶▶

```

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.

PART *integer*

Identifies the partition of a partitioned table space to lock. The table identified by *table-name* must belong to a partitioned table space that is defined with LOCKPART YES. 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

LOCK TABLE

table space is locked, other processes can delete LOBs or update them to a null value, but they cannot insert LOBs with a non-null 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

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 Section 5 of *DB2 Application Programming and SQL Guide* or Section 5 (Volume 2) of *DB2 Administration Guide*.

Example

Obtain a lock on the sample table named DSN8610.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 DSN8610.EMP IN EXCLUSIVE MODE;
```

OPEN

The OPEN statement opens a cursor.

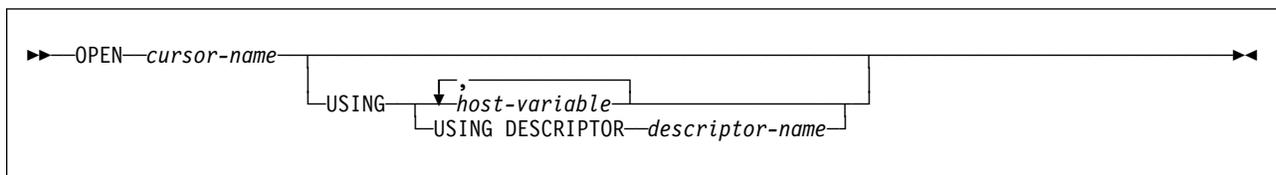
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

See “DECLARE CURSOR” on page 634 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 “Notes” on page 636 in the description of the DECLARE CURSOR statement. When the OPEN statement is executed, the cursor must be in the closed state.

The SELECT statement of the cursor is either:

- The *select-statement* specified in the DECLARE CURSOR statement, or
- The prepared *select-statement* identified by the *statement-name* 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 specified in the SELECT statement and the current values of any host variables 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 them. 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.” DB2 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) of a prepared statement. (For an

explanation of parameter markers, see “PREPARE” on page 757.) If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, you must use USING. If the prepared statement does not include parameter markers, USING is ignored.

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.

USING 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
- 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 results, 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 890.

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 907 for how to represent *descriptor-name* in C.

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 more on the process of replacement, see Parameter marker replacement on page 756.

The USING clause is intended for a prepared SELECT statement that contains parameter markers. However, it can also be used when the SELECT statement of the cursor is part of the DECLARE CURSOR statement and contains host variables. 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 attributes of 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. If a predicate of the SELECT refers to a host variable that does not have an indicator variable, the overriding value is always the value of the main variable because the indicator variable is ignored without a warning.

Notes

Errors occurring on OPEN: In local and remote processing, the DEFER(PREPARE) and REOPT(VARS) 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.
- A CONNECT has been performed. (CONNECT 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.
- 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 during OPEN that would otherwise not occur until some later FETCH statement. Moreover, INSERT, UPDATE, and DELETE statements executed while the cursor is open cannot affect the result table.

When a temporary copy of the result table is not used: INSERT, UPDATE, and DELETE statements executed while the cursor is open can affect the result table if they are issued from the same application process. 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.

OPEN

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 84. 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 on page 759.

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.

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.

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 DSN8610.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 character string form of the statement. The executable form is called a prepared statement. The character string form is called a statement string.

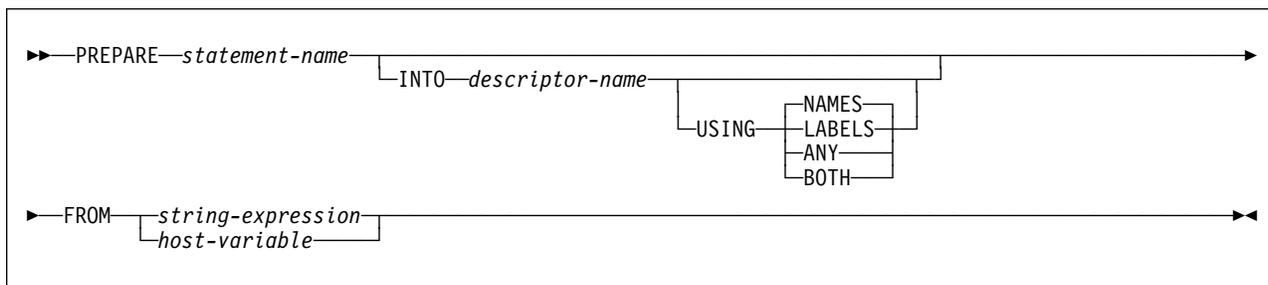
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The authorization rules are those defined for the dynamic preparation of the SQL statement specified by the PREPARE statement. For example, see “Chapter 5. Queries” on page 307 for the authorization rules that apply when a SELECT statement is prepared.

Syntax



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 (prepared statement or table)” on page 659 and “SQL descriptor area (SQLDA)” on page 890 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.

#

Notes

Rules for statement strings: The statement string must be one of the following SQL statements:

ALTER	REVOKE
COMMENT ON	ROLLBACK
COMMIT	SET CURRENT DEGREE
CREATE	SET CURRENT LOCALE LC_CTYPE
DELETE	SET CURRENT OPTIMIZATION HINT
DROP	SET CURRENT PATH
EXPLAIN	SET CURRENT PRECISION
FREE LOCATOR	SET CURRENT RULES
GRANT	SET CURRENT SQLID
HOLD LOCATOR	SIGNAL SQLSTATE
INSERT	UPDATE
LABEL ON	select-statement
LOCK TABLE	
RENAME	

The statement string must not:

- Begin with EXEC SQL and end with a statement terminator
- Include references to host variables
- Include comments

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 “EXECUTE” on page 689, “OPEN” on page 753, and Section 7 of *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 notation is not a function call, but rather is a “promise” that the data type of the host variable at run time will be the same as, or can be converted to, the data type that was specified.

In the following example, the value of the argument that is provided for the TRANSLATE function at run time must be VARCHAR(12) or a data type that can be converted to VARCHAR(12).

```
UPDATE EMPLOYEE
   SET LASTNAME = TRANSLATE(CAST(? AS VARCHAR(12)))
   WHERE EMPNO = ?
```

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 function.

Untyped parameters markers can be used in dynamic SQL statements in selected locations where host variables are supported. Table 48 shows these locations and the resulting data type of the parameter. The table groups the locations into expressions, predicates, and functions to help show where untyped parameter markers are allowed. When an untyped parameter marker is used in a function (including arithmetic operators, CONCAT and datetime operators) with an unqualified function name, the implicit qualifier is assumed to be 'SYSIBM' for function resolution.

Table 48 (Page 1 of 4). Untyped parameter marker usage

Location of untyped parameter marker	Data type (or error if not supported)
Expressions (including select list, CASE, and VALUES)	
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
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(255 - <i>n</i>). In all other cases, the data type is VARCHAR(255).
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 a LOB string	The data type of the other operand (the LOB string)
The value on the right-hand 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 48 (Page 2 of 4). Untyped parameter marker usage

Location of untyped parameter marker	Data type (or error if not supported)
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 99 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 99 to all the <i>result-expressions</i> that are not NULL or untyped parameter markers
Alone as a column-expression in a single-row VALUES clause that is not within an INSERT statement	Error
Alone as a column-expression in a single-row VALUES clause within an INSERT statement	The data type of the column or, if the column is defined as a distinct type, the source data type of the distinct type
Predicates	
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 99 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 99 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 99 on the operands in the IN list that are not parameter markers
The first operand of an IN predicate when the right-hand side is a subselect of fullselect, for example, ? IN (subselect)	The data type of the selected column

Table 48 (Page 3 of 4). Untyped parameter marker usage

Location of untyped parameter marker	Data type (or error if not supported)
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-hand 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 BLOB(4000), depending on the data type of the first operand that is not an untyped parameter marker
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 BLOB(4000), depending on the data type of the first operand that in 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 BLOB(1), depending on the data type of the first operand that in not an untyped parameter marker
Operand of a NULL predicate	Error
Functions	
All operands of COALESCE (also called VALUE) or NULLIF	Error
Any operand of COALESCE or NULLIF when at least one other operand is not an untyped parameter marker	The result of applying the “Rules for result data types” on page 99 on the operands that are not untyped parameter markers, the data type of the other operand
Both operands of POSSTR	VARCHAR(4000) for both operands
One operand of POSSTR when the other operand is a character data type	VARCHAR(4000)
One operand of POSSTR when the other operand is a graphic data type	VARGRAPHIC(2000)
One operand of POSSTR when the other operand is a BLOB	BLOB(4000)
First operand of SUBSTR	VARCHAR(4000)
Second or third operand of SUBSTR	INTEGER
One operand of TIMESTAMP	TIME
First operand of TIMESTAMP_FORMAT	VARCHAR(255)
First operand of TRANSLATE	Error

Table 48 (Page 4 of 4). Untyped parameter marker usage

Location of untyped parameter marker	Data type (or error if not supported)
Second or third operand of TRANSLATE	VARCHAR(4000) or VARGRAPHIC(2000), depending on whether the data type of the first operand is character or graphic
Fourth operand of TRANSLATE	VARCHAR(1) or VARGRAPHIC(1), depending on whether the data type of the first operand is character or graphic
First operand of VARCHAR_FORMAT	TIMESTAMP
Unary minus	DOUBLE PRECISION
Unary plus	DOUBLE PRECISION
Operand of any scalar function (except those that are described above in this table for COALESCE, NULLIF, POSSTR, SUBSTR, TIMESTAMP, TIMESTAMP_FORMAT, TRANSLATE, VALUE, and VARCHAR_FORMAT), including user-defined functions	Error
Operand of a column function	Error
Operand of a table function	Error

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(VARS) 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 <i>not</i> 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.

PREPARE

- SELECT, INSERT, UPDATE, 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 “Notes” on page 636 for more information about the scope of a *cursor-name*.

Preparation with PREPARE INTO and REOPTVAR: If bind option REOPT(VARS) 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. To avoid having a statement prepared twice, avoid using PREPARE INTO when REOPT(VARS) is in effect.

Example

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 DSN8610.DEPT VALUES(?,?,?,?)';
```

(Check for successful execution and read values into host variables)

```
EXEC SQL EXECUTE DEPT_INSERT USING :S1, :S2, :S3, :S4, :S5;
```

RELEASE (connection)

The RELEASE statement places one or more connections in the release pending state.

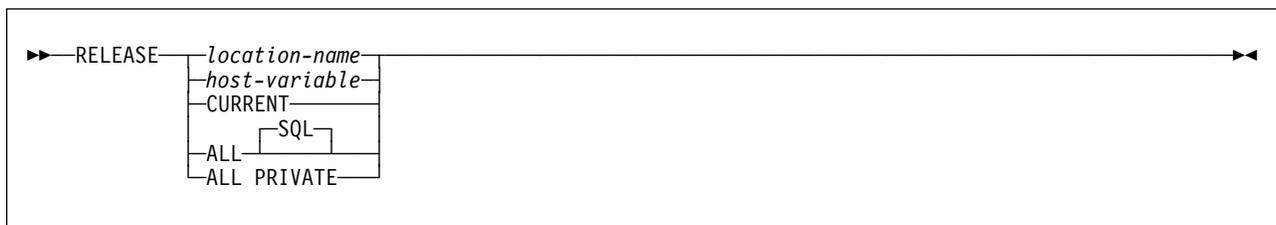
Invocation

This statement can only be embedded in an application program, except in REXX programs. It is an executable statement that cannot be dynamically prepared.

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.

If the RELEASE statement is successful, the identified connection is placed in the release pending status and will therefore be ended during the next commit operation. If the RELEASE statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.

CURRENT

Identifies the current SQL connection of the application process. The application process must be in the connected state.

RELEASE (connection)

If the RELEASE statement is successful, the identified connection is placed in the release pending state and will therefore be ended during the next commit operation. If the RELEASE statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.

ALL or ALL SQL

Identifies all existing connections (including local, SQL, and DB2 private connections) of the application process and places these connections in the release pending status. These connections are ended during the next commit operation. An error or warning does not occur if no connections exist when the statement is executed.

ALL PRIVATE

Identifies all existing DB2 private connections of the application process and places these connections in the release pending status. These DB2 private connections are ended during the next commit operation. An error or warning does not occur if no DB2 private connections exist when the statement is executed.

Notes

Using CONNECT (Type 1) semantics does not prevent using RELEASE.

RELEASE does not close cursors, does not release any resources, and does not prevent further use of the connection.

ROLLBACK does not reset the state of a connection from release pending to held.

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.

An application 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.

For further information, see “When a connection is ended” on page 37.

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 application 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;
```

Example 4: The first phase of an application involves the use of DB2 private protocol access with the local DB2 subsystem as the application server and the second phase involves explicit CONNECTs to remote servers. The existence of the DB2 private connections allocated during the first phase could cause a CONNECT operation to fail. Accordingly, the following statement is executed before the commit operation that separates the two phases:

```
EXEC SQL RELEASE ALL PRIVATE;
```


RENAME

The RENAME statement renames an existing 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.

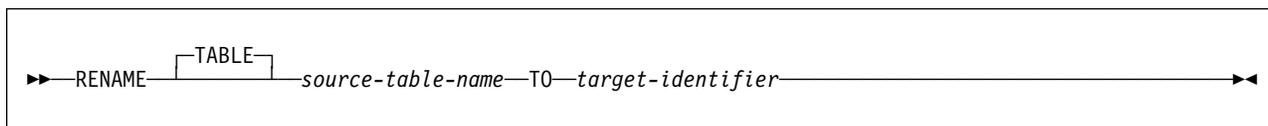
Authorization

The privilege set that is defined below must include at least one of the following:

- Ownership of the table
- DBADM, DBCTRL, or DBMAINT authority for the database that contains the table
- SYSADM or SYSCTRL authority

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges held by the authorization ID of 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 table that has a trigger defined on it, a declared temporary table, a catalog table, an active RLST table, a view, or a synonym. If you specify a three-part name or alias for the source, the source table must exist at the current server.

There is no support for changing the name of an alias. An alias on the source table continues to refer to the source table after the rename.

If any view definitions currently refer to the source table, an error is issued.

The specified table is renamed to the new name. All privileges and indexes on the table are preserved.

target-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 already exists at the current server.

#

Notes

Catalog Table Updates: Entries in the following catalog tables are updated to reflect the new table name:

- SYSAUXRELS
- SYSCHECKS
- SYSCHECKDEP
- SYSCOLAUTH
- SYSCOLDIST
- SYSCOLDISTSTATS
- SYSCOLSTATS
- SYSCOLUMNS
- SYSFIELDS
- SYSFOREIGNKEYS
- SYSINDEXES
- SYSPLANDEP
- SYSPACKDEP
- SYSRELS
- SYSSYNONYMS
- SYSTABAUTH
- SYSTABLES
- SYSTABSTATS

Entries in SYSSTMT and SYSPACKSTMT are not updated.

Invalidation of plans, packages, and dynamic statements: When any table except an auxiliary table is renamed, plans and packages that refer to that table are invalidated. If any dynamic statements in the statement cache refer to the table, they are invalidated; DB2 must refresh those statements in the cache the next time they are executed.

When an auxiliary table is renamed, plans and packages that refer to the auxiliary table are not invalidated.

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 defined over the source table are *transferred* to the new table. The index catalog tables are updated appropriately.

Object Identifier: Renamed tables keep the same object identifier (OBID) as the original table.

Renaming Registration Tables: If an application registration table or object registration table is specified as the source table for RENAME, then once RENAME completes, it is as if that table had been dropped. There is no ART (application registration table) or ORT (object registration table) once the ART or ORT table has been renamed.

Example

Change the name of the EMP table to EMPLOYEE:

```
RENAME TABLE EMP TO EMPLOYEE;
```

REVOKE

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
- 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.

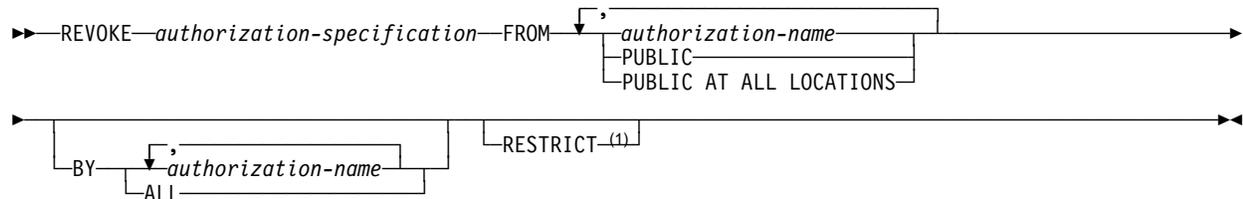
If the authorization mechanism was not activated when the DB2 subsystem was installed, an error condition occurs.

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 authorization ID of the statement is the authorization ID of the owner of the plan or package. If the statement is dynamically prepared, the authorization ID of the statement is the SQL authorization ID of the process.

Syntax



Note:

¹ The RESTRICT clause can be specified only for the forms of the REVOKE statement that require it.

Description

authorization-specification

Names one or more privileges, in one of the formats described below. 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 once.

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.

PUBLIC

Revokes a grant of privileges to PUBLIC.

PUBLIC AT ALL LOCATIONS

Revokes a grant of privileges to PUBLIC AT ALL LOCATIONS.

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 with SYSADM or SYSCTRL authority can use BY, even if the authorization ID 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 listed must have explicitly granted some named privilege to all named users.

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 can only be specified for the forms of the REVOKE statement that require it. These forms are revoking the USAGE privilege on distinct types and the EXECUTE privilege on user-defined functions and stored procedures.

Notes

For more on DB2 privileges, read Section 3 (Volume 1) of *DB2 Administration Guide*. For information on access control authorization exits, see Appendix B (Volume 2) of *DB2 Administration Guide*.

Revoked privileges: The privileges revoked from an authorization ID are those that are identified in the statement and which were granted to the authorization ID by the authorization ID of the statement. Other privileges can be revoked as the result of a cascade revoke.

REVOKE

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.

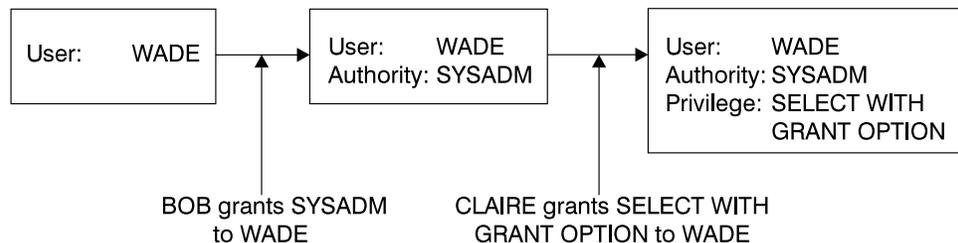
The rules also apply to the implicit grants that are made as a result of a CREATE VIEW statement.

Cascade revoke does not occur under any of the following conditions:

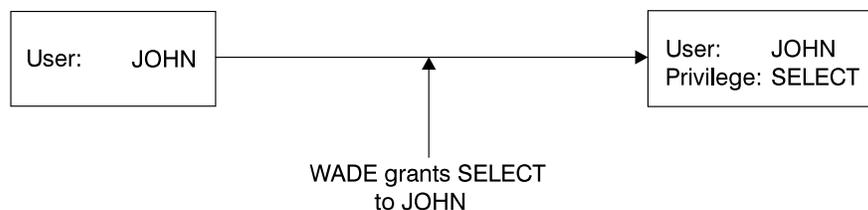
- The privilege was granted by a current install SYSADM.
- The privilege is the USAGE privilege on a distinct type and the user owns any of these items:
 - A user-defined function or stored procedure that uses the distinct type
 - A table that has a column that uses the distinct type
- The privilege is the EXECUTE privilege on a user-defined function and the user 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 user 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:

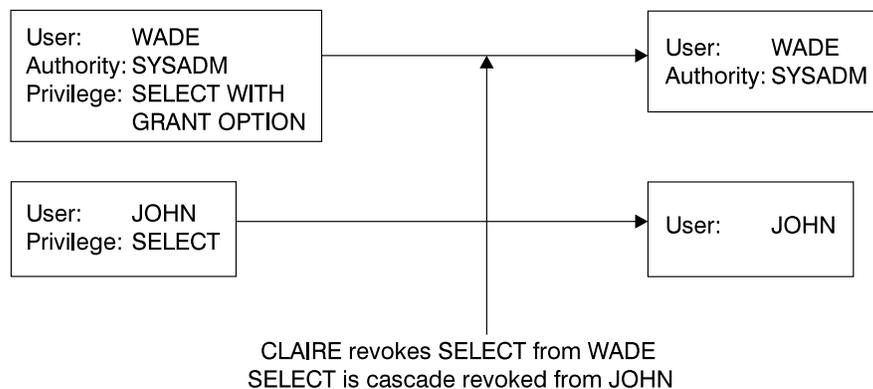
1. Suppose BOB grants SYSADM authority to WADE. Later, CLAIRE grants the SELECT privilege on a table with the WITH GRANT OPTION to WADE.



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 Section 3 (Volume 1) of *DB2 Administration Guide*. For another example of cascading revokes, see Section 3 (Volume 1) of *DB2 Administration Guide*.

Revoking a SELECT privilege that was exercised to create a view 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
 # authority 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 627.

Invalidation of plans and packages: A revoke or cascaded revoke of any privilege, excluding the EXECUTE privilege on a user-defined function, 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.³⁷

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.³⁷

³⁷ Dependencies on stored procedures can be checked only if the procedure name is specified as a literal and not via a host variable in the CALL statement.

REVOKE

Invalidation of dynamic statements in the cache: If a dynamic SQL statement is cached and its authorization involved a DELETE, INSERT, SELECT, or UPDATE table privilege, or the EXECUTE function or stored procedure privilege, revoking the privilege makes the statement invalid. DB2 will have to prepare the statement the next time that it is executed.

Multiple grants: If you granted the same privilege to the same user more than once, revoking that privilege from that user nullifies all those grants. It does not nullify any grant of that privilege made by others.

When a REVOKE statement revokes multiple grants, the grants are revoked, one at a time, in an undefined order. If, for some reason, a revocation is in error, the execution of the statement is stopped, and all the revoked grants are restored. Such an error certainly occurs if a table or view is specified twice after the keyword ON. One also occurs when a table and a view based on the table are both specified after ON. The error would occur if revoking some grant for the table causes the view to be dropped before all grants have been revoked for the view.

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:

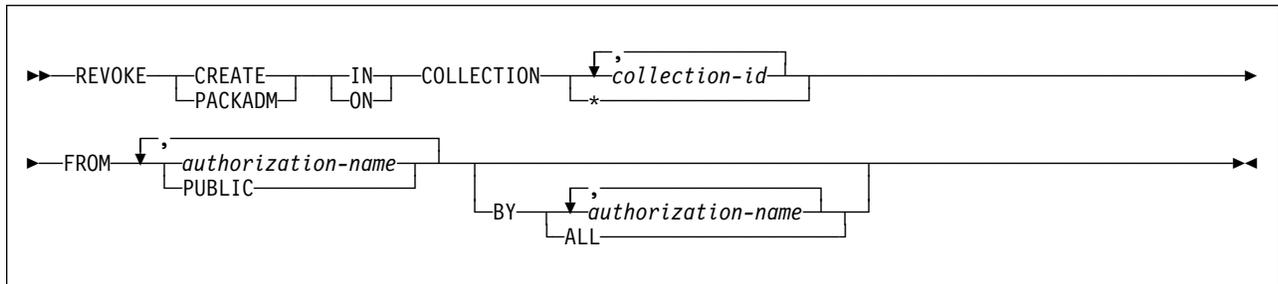
- If X is revoked, the user still has privilege P.
- If P is revoked, the user still has the privilege because it is inherent in X.

Ownership privileges: The privileges inherent in the ownership of an object cannot be revoked.

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 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 for a description of the BY clause.

Example

Revoke the privilege to create new packages in collections QAACLONE and DSN8CC61 from CLARK.

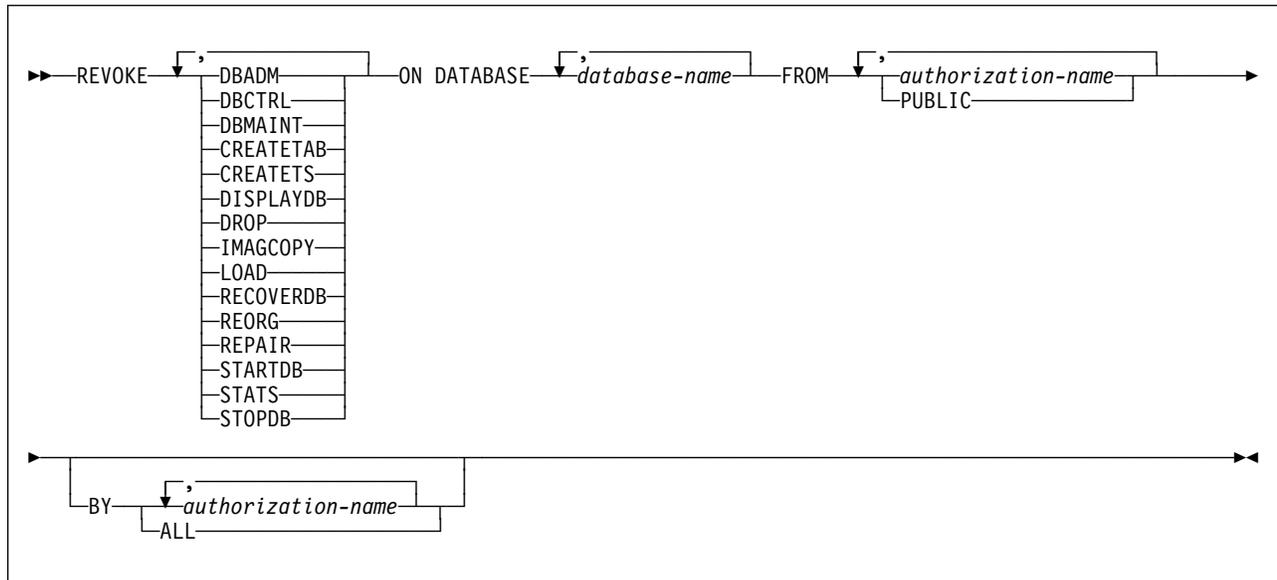
```
REVOKE CREATE IN COLLECTION QAACLONE, DSN8CC61 FROM CLARK;
```

REVOKE (database privileges)

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. For a TEMP database, you cannot
revoke the privilege from PUBLIC. When a TEMP 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 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.

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 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 for a description of the BY clause.

Examples

Example 1: Revoke drop privileges on database DSN8D61A from user PEREZ.

```
REVOKE DROP
  ON DATABASE DSN8D61A
  FROM PEREZ;
```

Example 2: Revoke repair privileges on database DSN8D61A from all local users. (Grants to specific users will not be affected.)

```
REVOKE REPAIR
  ON DATABASE DSN8D61A
  FROM PUBLIC;
```

Example 3: Revoke authority to create new tables and load tables in database DSN8D61A from users WALKER, PIANKA, and FUJIMOTO.

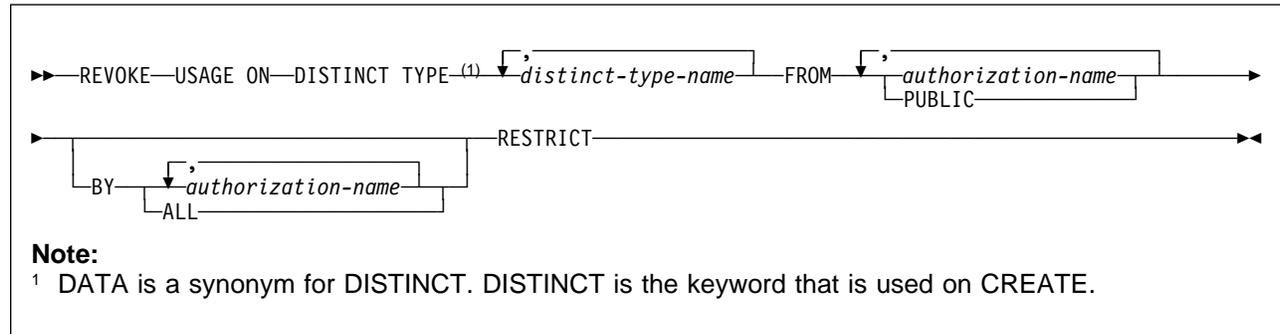
REVOKE (database privileges)

```
REVOKE CREATETAB,LOAD  
ON DATABASE DSN8D61A  
FROM WALKER,PIANKA,FUJIMOTO;
```

REVOKE (distinct type privileges)

This form of the REVOKE statement revokes the privilege to use distinct types (user-defined data types).

Syntax



Description

USAGE

Revokes the privilege to use the identified distinct types.

DISTINCT 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. If you specify an unqualified name, the name 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 when the plan or package was created or last rebound. If QUALIFIER was not used, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

FROM

Refer to “REVOKE” on page 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 for a description of the BY clause.

RESTRICT

Prevents the USAGE privilege from being revoked on a distinct type if any of the following conditions exist:

- The revokee owns a function or stored procedure that uses the distinct type.
- The revokee owns a table that has a column that uses the distinct type.

REVOKE (distinct type privileges)

Examples

Example 1: Revoke the USAGE privilege on distinct type SHOESIZE from user JONES.

```
REVOKE USAGE ON DISTINCT TYPE SHOESIZE FROM JONES RESTRICT;
```

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 DISTINCT TYPE US_DOLLAR FROM PUBLIC RESTRICT;
```

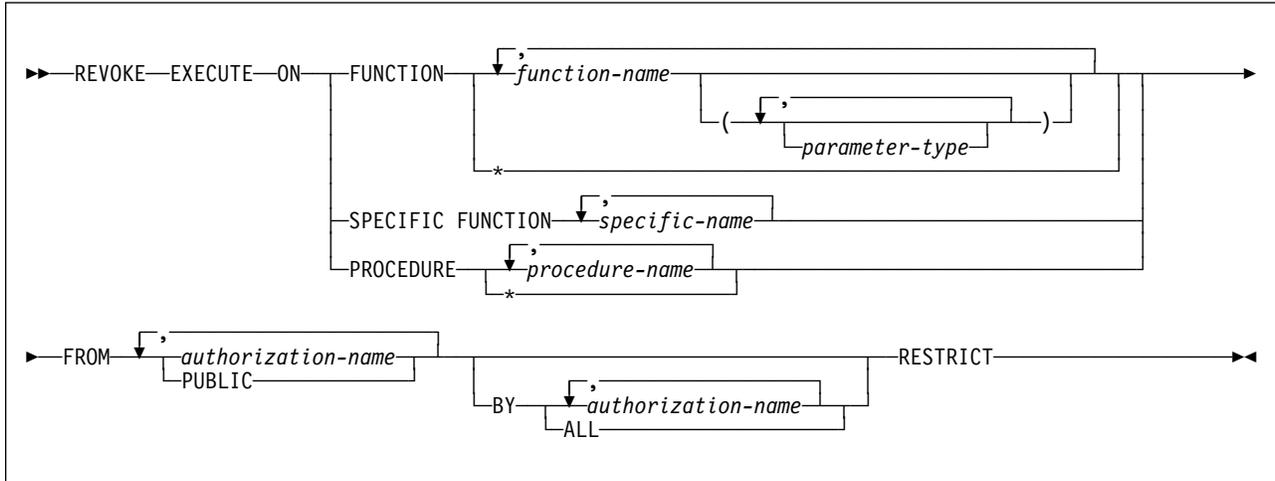
Example 3: Revoke the USAGE privilege on distinct type CANADIAN_DOLLARS from the administrative assistant (ADMIN_A) .

```
REVOKE USAGE ON DISTINCT TYPE CANADIAN_DOLLARS  
FROM ADMIN_A RESTRICT;
```

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* [AS LOCATOR ⁽¹⁾]

Note:

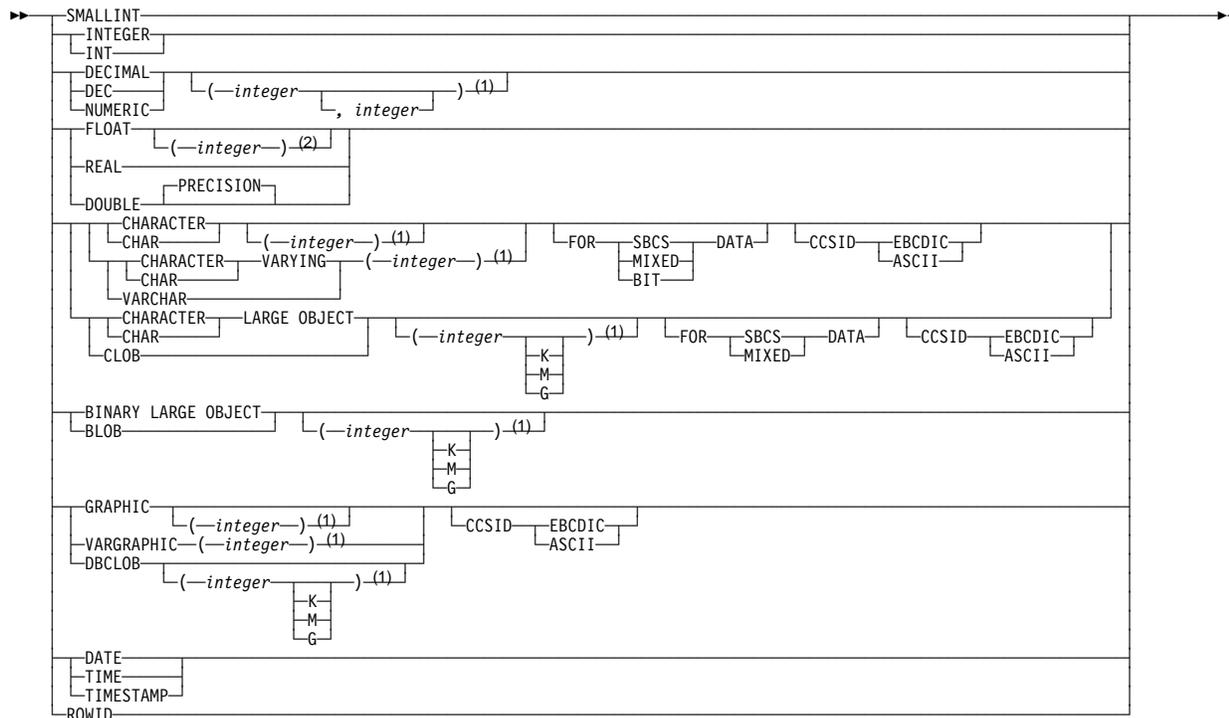
¹ AS LOCATOR can be specified only for a LOB data type or a distinct type that is based on a LOB data type.

data type:

► *built-in-data-type* | *distinct-type-name*

REVOKE (function or procedure privileges)

built-in data type:



Notes:

- 1 The values that are specified for length, precision, or scale attributes must match the values that were specified when the function was created. Coding specific values is optional. Empty parentheses, (), can be used instead to indicate that DB2 ignores the attributes when determining whether data types match.
- 2 The value that is specified does not have to match the value that was specified when the function was created because matching is based on data type (REAL or DOUBLE). $1 \leq integer \leq 21$ indicates REAL and $22 \leq integer \leq 53$ indicates DOUBLE. Coding a specific value is optional. Empty parentheses cannot be used.

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

The function that is identified 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 DISTINCT TYPE statement.

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. You can identify a function by its name only if there is exactly one function with *function-name* in the schema. If you do not explicitly qualify the function name with a schema name, the function name 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 when the plan or package was created or last rebound. If QUALIFIER was not used, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

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.

function-name

Specifies 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 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 is used to uniquely identify the function.

For data types that have a length, precision, or scale attribute, you can specify a value or use a set of empty parentheses:

- Empty parentheses indicate that DB2 ignores the attribute when determining whether the data types match.
 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 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.

REVOKE (function or procedure privileges)

- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default length of the data type is implied. For example:

CHAR	CHAR(1)
GRAPHIC	GRAPHIC(1)
DECIMAL	DECIMAL(5,0)
FLOAT	DOUBLE (length of 8)

The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. For a complete list of the default lengths of data types, see “CREATE TABLE” on page 570.

For data types with a subtype or encoding scheme attribute, specifying the FOR DATA clause or 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.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name.

PROCEDURE *procedure-name*

Identifies a stored procedure that is defined at the current server. If you do not explicitly qualify the procedure name with a schema name, the procedure name 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 when the plan or package was created or last rebound. If QUALIFIER was not used, the schema name is the owner of the plan or package.
- If the statement is dynamically prepared, the schema name is the SQL authorization ID in the CURRENT SQLID special register.

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 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 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:

- 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

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 RESTRICT;
```

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 RESTRICT;
```

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 RESTRICT;
```

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 RESTRICT;
```

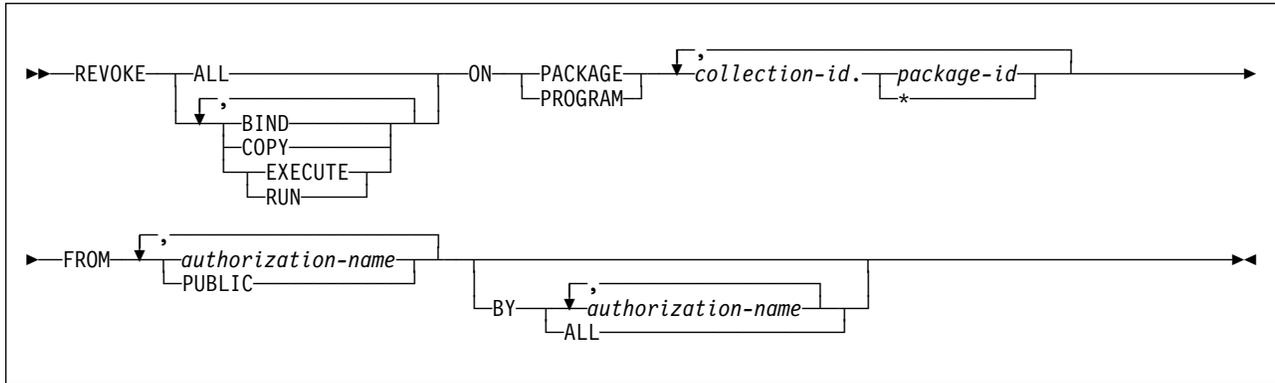
You can also code the CHAR(10) data type as CHAR().

REVOKE (package privileges)

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 726 for “GRANT (package privileges)” on page 725.)

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-id*,...

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-id*. The same package must not be specified more than once.

The word PROGRAM can be used in place of PACKAGE.

FROM

Refer to “REVOKE” on page 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 for a description of the BY clause.

Example

Revoke the privilege to copy all packages in collection DSN8CC61 from LEWIS.

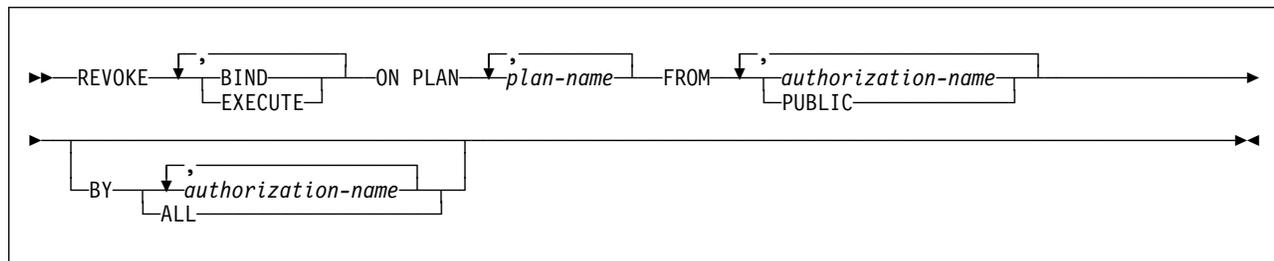
```
REVOKE COPY ON PACKAGE DSN8CC61.* FROM LEWIS;
```

REVOKE (plan privileges)

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 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 for a description of the BY clause.

Examples

Example 1: Revoke authority to bind plan DSN8IP61 from user JONES.

```
REVOKE BIND ON PLAN DSN8IP61 FROM JONES;
```

Example 2: Revoke authority previously granted to all users at the current server to bind and execute plan DSN8CP61. (Grants to specific users will not be affected.)

```
REVOKE BIND,EXECUTE ON PLAN DSN8CP61 FROM PUBLIC;
```

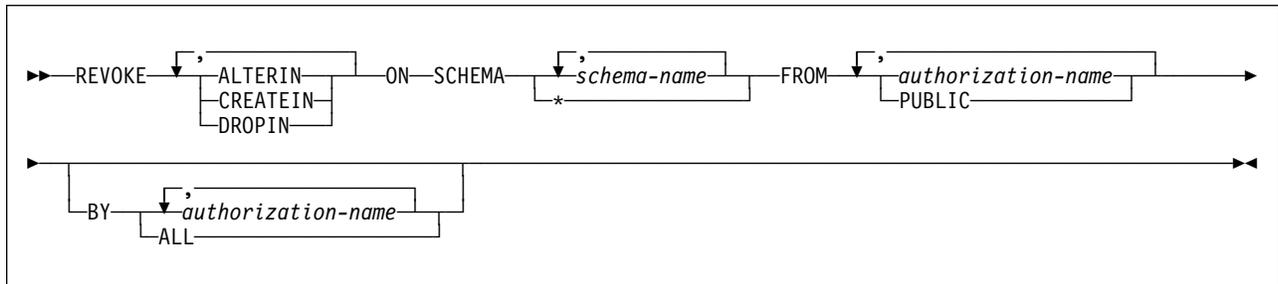
Example 3: Revoke authority to execute plan DSN8CP61 from users ADAMSON and BROWN.

```
REVOKE EXECUTE ON PLAN DSN8CP61 FROM ADAMSON,BROWN;
```

REVOKE (schema privileges)

This form of the REVOKE statement revokes privileges on schemas.

Syntax



Description

ALTERIN

Revokes 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, stored procedures, triggers, and user-defined functions in the designated schemas.

CREATEIN

Revokes the privilege to create distinct types, stored procedures, triggers, and user-defined functions in the designated schemas.

DROPIN

Revokes the privilege to drop distinct types, 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 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 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;
```

REVOKE (schema privileges)

| *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;

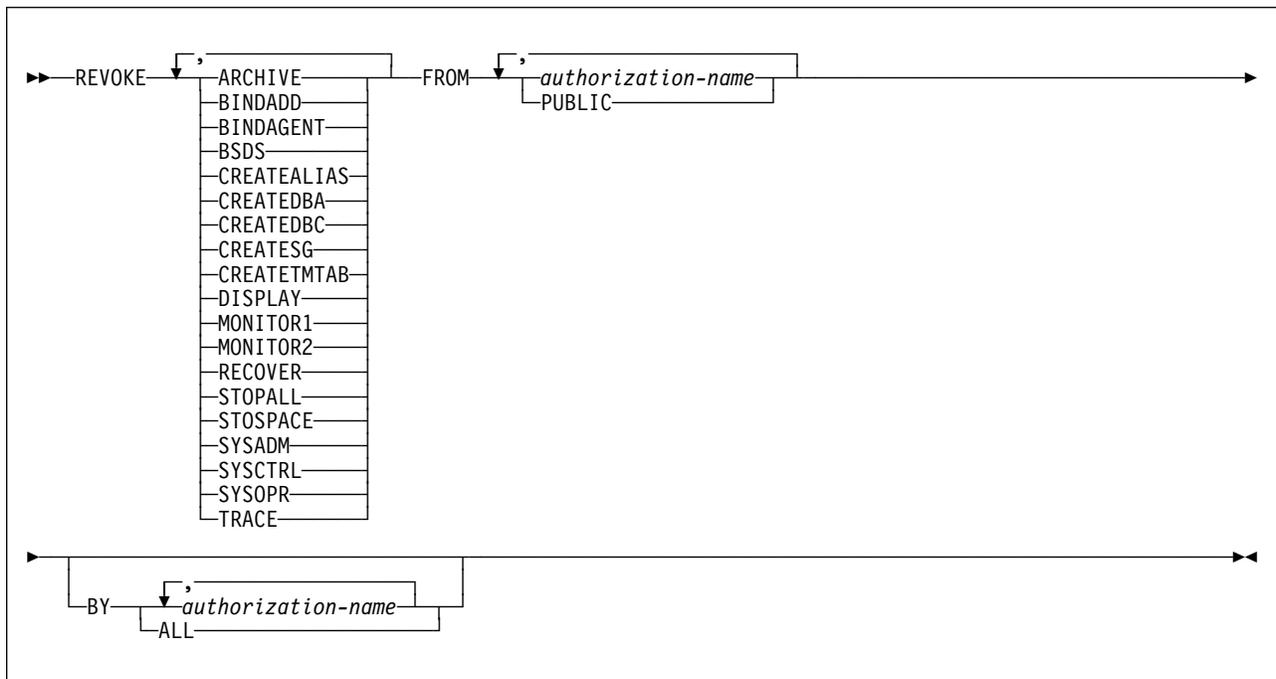
| *Example 4:* Revoke the ALTERIN and DROPIN privileges on schemas NEW_HIRE,
| PROMO, and RESIGN from HR (Human Resources).

| REVOKE ALTERIN, DROPIN ON SCHEMA NEW_HIRE, PROMO, RESIGN FROM HR;

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.

REVOKE (system privileges)

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

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 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 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.)

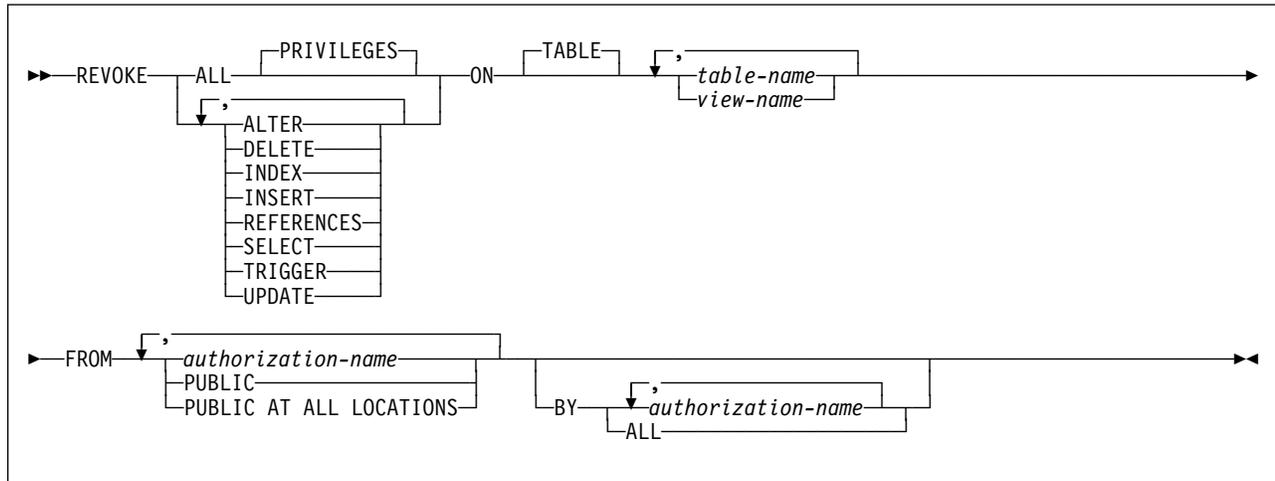
```
REVOKE TRACE  
FROM PUBLIC;
```

REVOKE (table or view privileges)

REVOKE (table or view privileges)

This form of the REVOKE statement revokes privileges on one or more tables or views.

Syntax



Description

ALL or ALL PRIVILEGES

If you specify ALL, the authorization ID of the statement must have granted a least one privilege on each identified table or view to each *authorization-name*. The privilege revoked from an authorization ID are those privileges on the table or view that the authorization ID of the statement granted to the authorization ID.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described, but only as it applies to the tables or views named in the ON clause.

ALTER

Revokes the privilege to use the ALTER statement.

DELETE

Revokes the privilege to use the DELETE statement.

INDEX

Revokes the privilege to use the CREATE INDEX statement.

INSERT

Revokes the privilege to use the INSERT statement.

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 use the SELECT statement. A view is dropped when the SELECT privilege that was used to create it is revoked, unless the owner of

#

the view was directly granted the SELECT privilege from another source before
the view was created.

|
|

TRIGGER

Revokes the privilege to use the CREATE TRIGGER statement.

UPDATE

Revokes the privilege to use the UPDATE statement. A list of column names can be used only with GRANT, not with REVOKE.

ON or ON TABLE

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 once, and a declared temporary table must not be identified.

#

FROM

Refer to “REVOKE” on page 772 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 772 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.

Examples

Example 1: Revoke SELECT privileges on table DSN8610.EMP from user PULASKI.

REVOKE SELECT ON TABLE DSN8610.EMP FROM PULASKI;

Example 2: Revoke update privileges on table DSN8610.EMP previously granted to all local DB2 users. (Grants to specific users are not affected.)

REVOKE UPDATE ON TABLE DSN8610.EMP FROM PUBLIC;

Example 3: Revoke all privileges on table DSN8610.EMP from users KWAN and THOMPSON.

REVOKE ALL ON TABLE DSN8610.EMP FROM KWAN, THOMPSON;

REVOKE (table or view privileges)

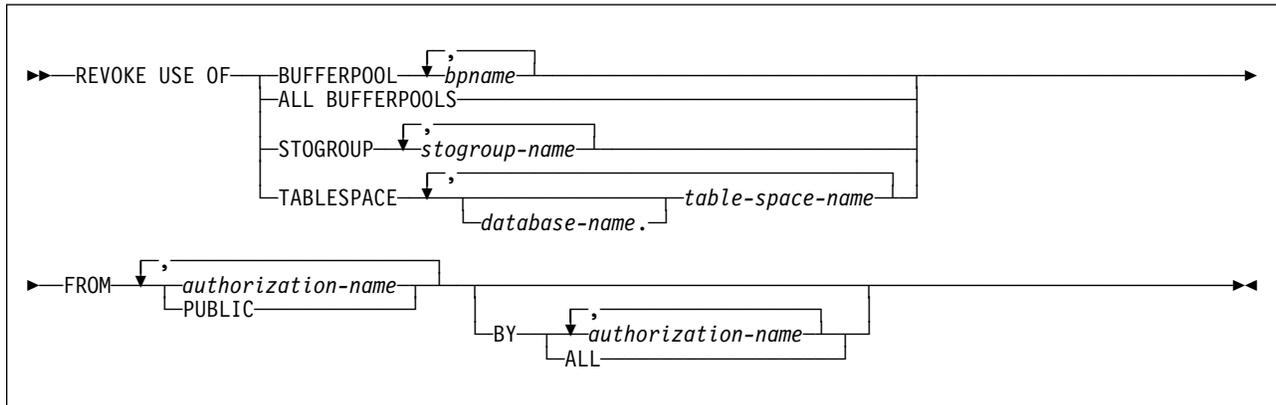
Example 4: Revoke the grant of SELECT and UPDATE privileges on the table DSN8610.DEPT to every user in the network. Doing so does not affect users who obtained these privileges from some other grant.

```
REVOKE SELECT, UPDATE ON TABLE DSN8610.DEPT  
FROM PUBLIC AT ALL LOCATIONS;
```

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 50 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 TEMP database, which are for declared temporary tables, you cannot revoke the privilege from PUBLIC. When a table space is created in the TEMP 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 772 for a description of the FROM clause.

BY

Refer to “REVOKE” on page 772 for a description of the BY clause.

REVOKE (use privileges)

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 DSN8S61D in the database DSN8D61A. 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 DSN8D61A.DSN8S61D
FROM PUBLIC;
```

ROLLBACK

The ROLLBACK statement ends a unit of recovery and backs out 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. Instead of rolling back the entire unit of recovery, the
ROLLBACK statement can be used to roll back changes only to a savepoint within
the unit of recovery, without ending the unit of recovery.

The ROLLBACK statement can be used to either:

- # • 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.
- # • 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.

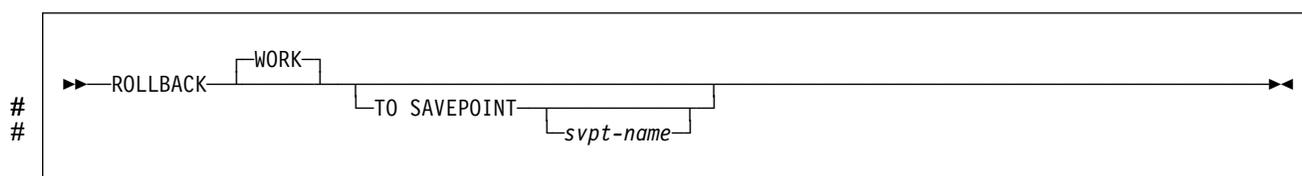
Invocation

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 effectively started. All changes made by ALTER, COMMENT ON, CREATE, DELETE, DROP, EXPLAIN, GRANT, INSERT, LABEL ON, RENAME, REVOKE, and UPDATE statements executed during the unit of recovery are backed out.

ROLLBACK without the TO SAVEPOINT clause also causes the following to occur:

- All locks implicitly acquired during the unit of recovery are released. See “LOCK TABLE” on page 751 for an explanation of the duration of explicitly acquired locks.
- All cursors are closed, all prepared statements are destroyed, and any cursors associated with the prepared statements are invalidated.

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.

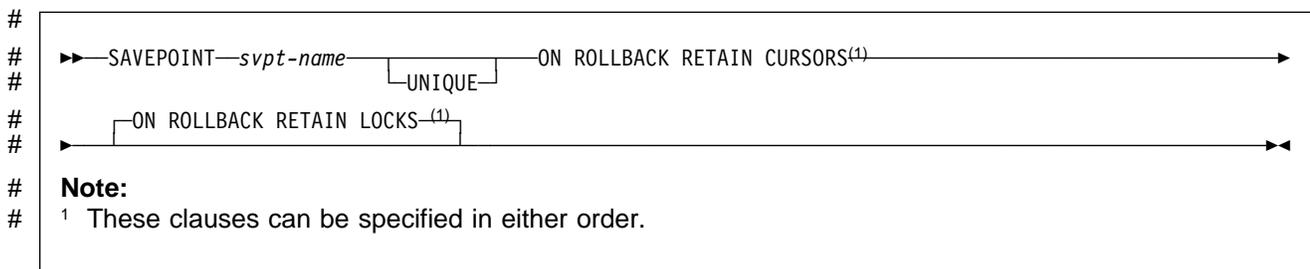
Invocation

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



Description

svpt-name
 # A savepoint identifier that names the savepoint. (A savepoint identifier is like an
 # SQL identifier except that it has maximum length of 128 bytes.)

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
 # *svpt-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 upon 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 upon 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 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

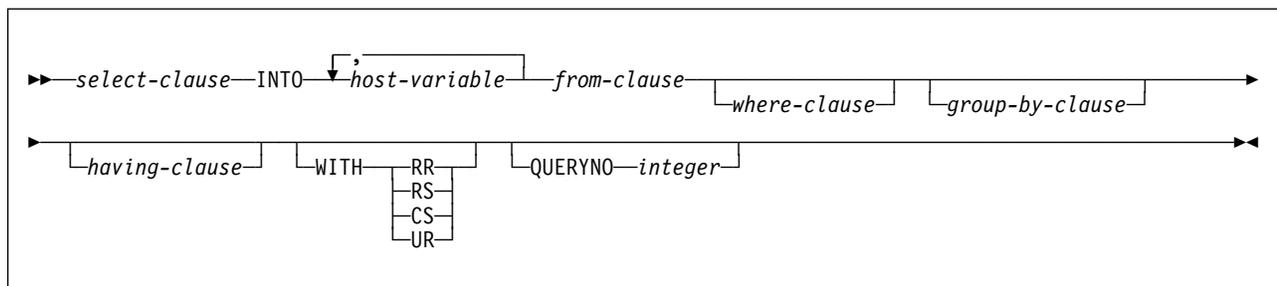
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

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
- The EXECUTE privilege on any user-defined function
- Ownership of the table or view
- DBADM authority for the database (tables only)
- SYSADM authority
- SYSCTRL authority (catalog tables only)

Syntax



Description

The table is derived by evaluating the *from-clause*, *where-clause*, *group-by-clause*,
 # *having-clause*, and *select-clause*, in this order. See “Chapter 5. Queries” on
 page 307 for a description of these clauses.

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 883.)

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 3. Language elements” on page 43. 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 3. Language elements” on page 43. 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.

WITH

#

Specifies the isolation level at which the statement is executed. (Isolation level does not apply to declared temporary tables because no locks are acquired.)

#

RR	Repeatable read
RS	Read stability
CS	Cursor stability
UR	Uncommitted read

WITH UR can be specified only if the result table is read-only.

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

SELECT INTO

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

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.

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, if hints are used. For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, see Section 5 (Volume 2) of *DB2 Administration Guide*.

Examples

Example 1: Put the maximum salary in DSN8610.EMP into the host variable MAXSALRY.

```
EXEC SQL SELECT MAX(SALARY)
        INTO :MAXSALRY
        FROM DSN8610.EMP;
```

Example 2: Put the row for employee 528671, from DSN8610.EMP, into the host structure EMPREC.

```
EXEC SQL SELECT * INTO :EMPREC
        FROM DSN8610.EMP
        WHERE EMPNO = '528671'
END-EXEC.
```

SET CONNECTION

The SET CONNECTION statement establishes the application server of the process by identifying one of its existing connections.

Invocation

This statement can only be embedded in an application program, except in REXX programs. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

```

▶▶ SET CONNECTION { location-name | host-variable } ▶▶

```

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.

SET 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 application server S is placed in the SQLERRP field of the SQLCA. If the application 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 OS/390
 - QSQ for OS/400
 - SQL for all other DB2 products
 - *vv* is a two-digit version identifier such as '06'.
 - *rr* is a two-digit release identifier such as '01'.
 - *m* is a one-digit modification level such as '0'.

For example, if the server is Version 6 of DB2 for OS/390 with the latest
maintenance, the value of SQLERRP is 'DSN06011'.

- 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.

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 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

```
▶▶ SET CURRENT DEGREE = string-constant | host-variable ▶▶
```

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.

The initial value of CURRENT DEGREE is determined by the value of field CURRENT DEGREE on installation panel DSNTIP4. The default for the initial value is 1 unless your installation has changed it to be ANY by modifying the value in that field.

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 Section 7 of *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 function that refers to a locale, such as LCASE, UCASE, and TRANSLATE (with a single argument).

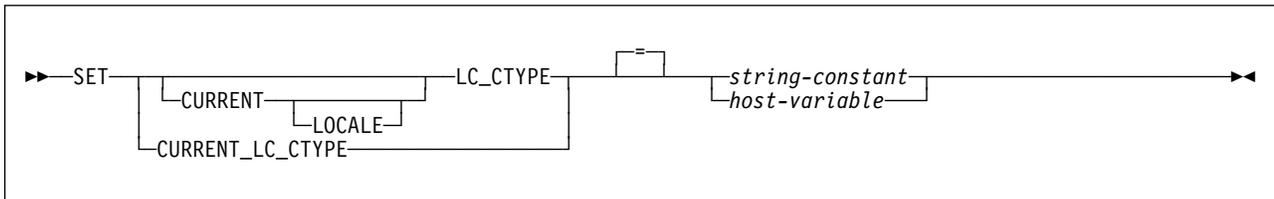
Invocation

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.

If a host variable is specified, its value must not be null. If it has an associated indicator, the value of the indicator value 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

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 *OS/390 C/C++ Programming Guide*. Some examples of locales include:

```
Fr_BE  
Fr_FR@EURO  
En_US  
Ja_JP
```

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 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

```
▶▶ SET CURRENT OPTIMIZATION HINT = string-constant | host-variable ▶▶
```

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 8 bytes.

Notes

The initial value of the CURRENT OPTIMIZATION HINT special register is set to the value that was used for the OPTHINT bind option. The OPTHINT bind option specifies whether optimization hints are used in determining the access path of static statements and identifies which user-defined hint (rows in the authid.PLAN_TABLE) is used. Therefore, if the SET CURRENT OPTIMIZATION HINT statement is not executed to change the value of the special register, DB2 uses the same optimization hint for dynamic statements that it uses for static statements. The default of OPTHINT for BIND PLAN and BIND PACKAGE is all blanks. All blanks indicate that DB2 uses normal optimization techniques and ignores optimization hints.

Example

Assume that delimited identifier 'NOHYB' identifies a user-defined optimization hint in authid.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'
```

SET CURRENT PACKAGESET

The SET CURRENT PACKAGESET statement assigns a value to the CURRENT PACKAGESET special register.

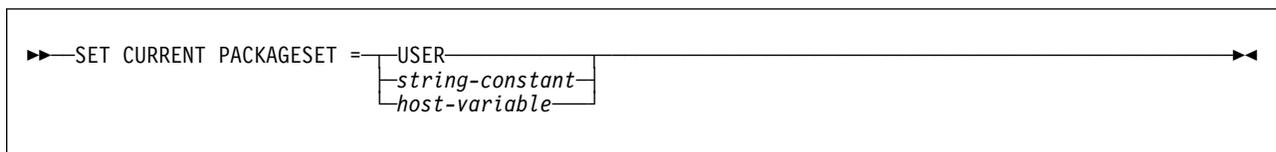
Invocation

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 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 18 bytes. If the length of the replacement is less than 18 bytes, it is padded on the right with blanks so that its length is 18 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.

SET CURRENT PACKAGESET

If the CURRENT PACKAGESET special register is blank, 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 Section 5 of *DB2 Application Programming and SQL Guide* .

SET CURRENT PACKAGESET is executed by the application 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 PATH

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 CURRENT PATH: These restrictions apply to the SET CURRENT 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. DB2 builds the string for the special register 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 the last one. The length of the resulting string cannot exceed 254 bytes.

Specifying SYSIBM and SYSPROC: Schemas SYSIBM and SYSPROC do not need to be specified in the special register. If either of these schemas is not explicitly specified in the CURRENT PATH special register, the schema is implicitly assumed at the front of the SQL path; if both are not specified, they are assumed in the order of SYSIBM, SYSPROC (see "Schemas and the SQL path" on page 57 for an example). Only the schemas that are explicitly specified in the CURRENT PATH register are included in the 254 byte limit.

To avoid having SYSIBM or SYSPROC 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 USER versus "USER": There is a difference between specifying USER with and without escape characters. To indicate that the value of the USER special register should be used in the SQL path, specify the keyword USER. If you specify USER as a delimited identifier instead (for example, "USER"), it is interpreted as a schema name of "USER". For example, assuming that the current value of the USER special register is SMITH, SET CURRENT PATH = SYSIBM, SYSPROC, USER, "USER" results in SYSIBM, SYSPROC, SMITH, USER being used in the SQL path.

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 CURRENT
 # PATH=*name* statement is a variable or a string constant:

- # • 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 CURRENT PATH.
- # • If *name* is not the same as a parameter or SQL variable in the SQL procedure,
 # DB2 uses *name* as a string constant and assigns the value *name* to CURRENT
 # 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 provides the SQL path, see “Schemas and the SQL
 | path” on page 57.

| **DRDA classification:** The SET CURRENT PATH statement is executed by the
 | application server and, therefore, is classified as a non-local SET statement in
 | DRDA.

| Examples

| *Example 1:* Set the CURRENT PATH special register to the list of schemas:
 | "SCHEMA1", "SCHEMA#2", "SYSIBM".

```
| SET CURRENT PATH = SCHEMA1,"SCHEMA#2", SYSIBM;
```

| If the special register provides the SQL path, then 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 CURRENT PATH = CURRENT PATH, SMITH, SYSPROC;
```

| The value of the special register becomes:

```
| "SCHEMA1", "SCHEMA#2", "SYSIBM", "SMITH", "SYSPROC"
```

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, and the value must be 'DEC15' or 'DEC31'. An error occurs if any other values are specified.

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 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

```

▶▶ SET CURRENT RULES = string-constant
                       host-variable

```

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'. An error occurs if any other values are specified.

Notes

For the effect of the values 'DB2' and 'STD' on the execution of certain SQL statements, see "CURRENT RULES" on page 109.

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.

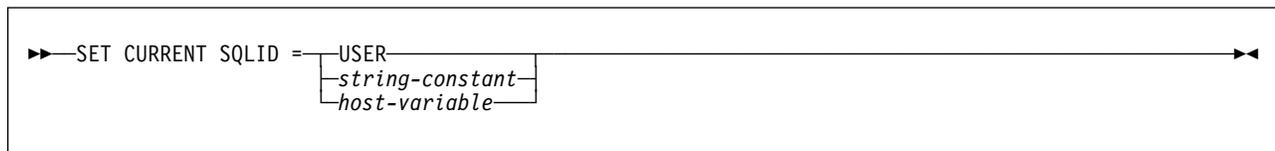
Invocation

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 and the implicit qualifier 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.

Syntax



Description

The value of CURRENT SQLID is replaced by the value of USER, *string-constant*, or *host-variable*. The value specified by a *string-constant* or *host-variable* must be a character string that is not longer than 8 bytes. If the length of the value is less than 8, it is padded on the right with blanks so that it is a string of 8 bytes. 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

The value of CURRENT SQLID is called the SQL authorization ID. 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 implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements

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 Chapter 2 of *DB2 Command Reference*.

The initial value of the SQL authorization ID is established during connection or sign-on processing. The value specified in the SET CURRENT SQLID is the SQL authorization ID until one of the following events occurs:

- The SQL authorization ID is changed by the execution of a new 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.

SET CURRENT SQLID is executed by the application server and is therefore classified as a non-local SET statement in DRDA.

Example

Set the CURRENT SQLID to the primary authorization ID.

```
SET CURRENT SQLID=USER;
```

SET host-variable assignment

SET host-variable assignment

The SET host-variable assignment statement assigns values, either of expressions
or NULL values, to host variables.

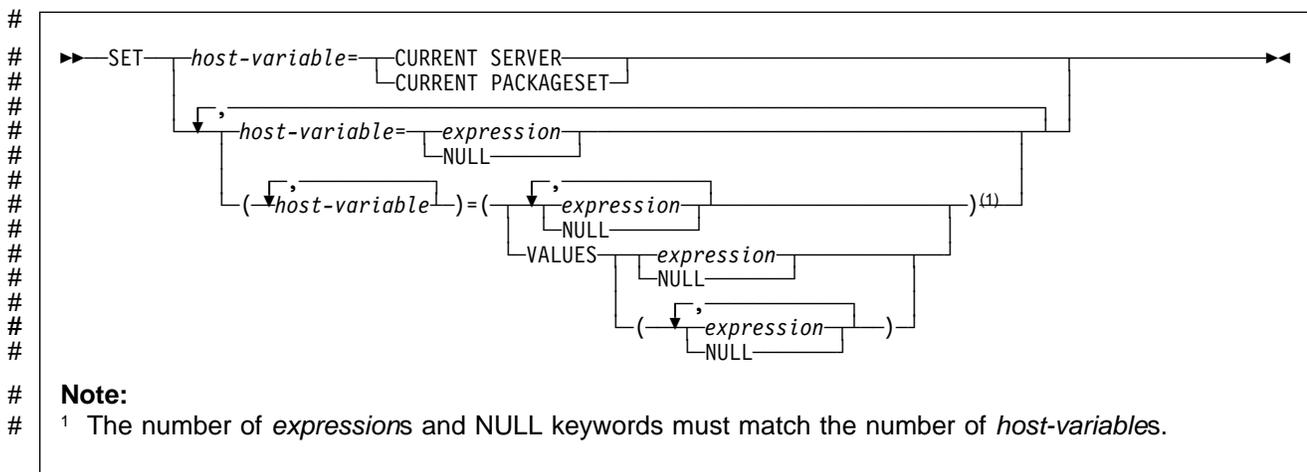
Invocation

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.

If the SET Assignment statement is used in the triggered action of a CREATE
TRIGGER statement, each *host-variable* must identify a transition variable. If
the statement is used in any other context, each *host-variable* must identify a
host variable.

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*).

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*.

```

#      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 131, except it cannot contain a reference 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 84. 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 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.

```

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 and the host variable XTIME to the local time at the current server.
#
#      SET :SERVER = CURRENT PATH,
#          :XTIME = CURRENT TIME;
#
#      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);
#
#      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);

```



```

#          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 triggering table. The name can identify an identity column that is
#          defined as GENERATED BY DEFAULT but not one defined as
#          GENERATED ALWAYS. 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 131. A reference to a local special register is the value of that special
#          register at the server when the trigger body is activated.
#
#          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 84.
#          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.
#
#          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 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 statement that limits  
# an increase to the COMMISSION column to 10% .
```

```
#         SET NEWROW.COMMISSION = 1.1 * OLDROW.COMMISSION;
```

SIGNAL SQLSTATE

The SIGNAL SQLSTATE statement is used to signal an error. It causes an error to be returned with the specified SQLSTATE and error description.

Invocation

This statement can only be used in the triggered action of a trigger.

Authorization

None required.

Syntax

```
►—SIGNAL SQLSTATE—sqlstate-string-constant—(—diagnostic-string-constant—)————►
```

Description

sqlstate-string-constant

Represents an SQLSTATE. It must be a character string constant with exactly 5 characters that follow these rules for application-defined SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented uppercase 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 character 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-constant

A character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it is truncated.

Example

Consider 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.

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table or view. Updating a row of a view updates a row of the table on which the view is based. 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 is used to update exactly one row, as determined by the current position of a cursor.

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

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

UPDATE

#

- 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 627.

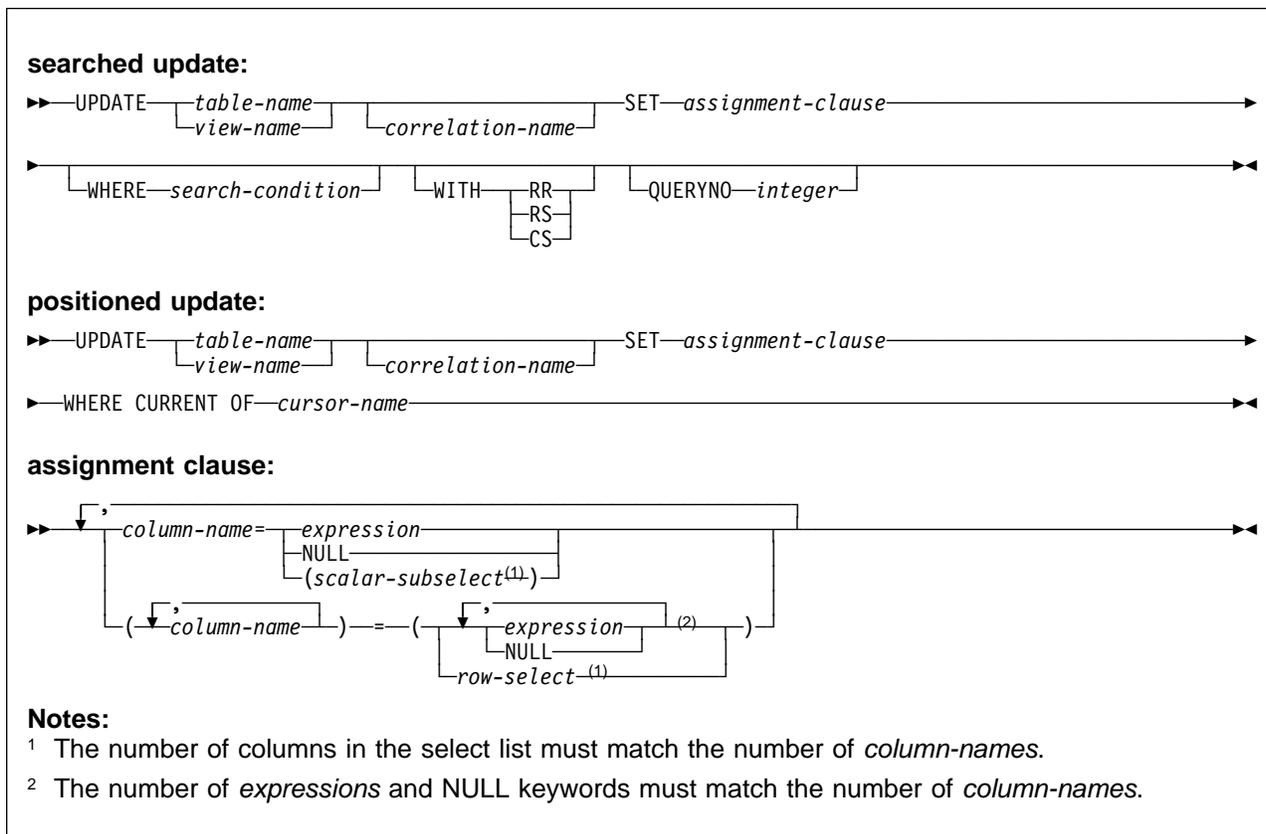
|
|

If an expression that refers to a function is specified, the privilege set must include any authority that is necessary to execute the function.

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 309.

Privilege set: If the statement is embedded in an application program, the privilege set is the privileges that are held by the authorization ID of 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 29 on page 342. (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 61).

Syntax



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. (For a description of a read-only view, see "CREATE VIEW" on page 627.)

In the IMS or CICS environments, the DB2 subsystem that contains the identified table or view must not be a remote Version 2 Release 3 subsystem.

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, then its description in Appendix D, "DB2 catalog tables" on page 911 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 Appendix B (Volume 2) of *DB2 Administration Guide*.

correlation-name

Can be used within *search-condition* or a *positioned* UPDATE to designate the
 # table or view. (For an explanation of *correlation-name*, see “Correlation names”
 # on page 114.)

SET

Introduces a list of one or more column names and the values to be assigned to the columns.

column-name

Identifies a column to be updated. *column-name* must identify a column of
 # the specified table or view, but must not identify a ROWID column, an
 # identity column that is defined as GENERATED ALWAYS, or a view
 | column that is derived from a scalar function, constant, or expression. The
 column names not be qualified, and the same column must not be specified
 more than once.

For a positioned update, allowable column names can be further restricted to those in a certain list. This list appears in the FOR UPDATE OF clause of the SELECT statement for the associated cursor. If the select statement is dynamically prepared, the FOR UPDATE OF clause must always be present. Otherwise, the clause can be omitted using the conditions described in “Positioned updates of columns” on page 172.

A view column 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 131. It must not include a column 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.

NULL

Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

scalar-subselect

Specifies a subselect that returns a single row with a single column. The
 # column value is assigned to the corresponding *column-name*. If the
 # subselect returns no rows, the null value is assigned; an error occurs if the
 # column to be updated is not nullable. An error also occurs if there is more
 # than one row in the result.

The subselect must not contain a GROUP BY or HAVING clause, and the
 # subselect (or subquery within the subselect) cannot use the table or view
 # being updated as its target. The subselect, however, can refer to columns
 # of the table or view to be updated. The value of such a column in the
 # subselect is the value of the column in the row before the row is updated.
 # Correlated references to these columns are allowed only in a searched
 # UPDATE and only in the search condition of the subselect. For example,
 # the following syntax is valid:

```

#           UPDATE TABLE1 T1
#             SET COL1 = (SELECT COUNT(*)
#                         FROM TABLE2 T2
#                         WHERE T2.COL5 = T2.COL3)
#             WHERE COL3 = 'ABC'

```

row-subselect

Specifies a subselect that returns a single row. The number of columns in
the row must match the number of *column-names* that are specified. The
column values are assigned to each corresponding *column-names*. If the
subselect returns no rows, the null value is assigned; an error occurs if the
column to be updated is not nullable. An error also occurs if there is more
than one row in the result.

The subselect must not contain a GROUP BY or HAVING clause, and the
subselect (or subquery within the subselect) cannot use the table or view
being updated as its target. The subselect, however, can refer to columns
of the table or view to be updated. The value of such a column in the
subselect is the value of the column in the row before the row is updated.
Correlated references to these columns are allowed only in a searched
UPDATE and only in the search condition of the subselect.

WHERE

Specifies the rows to be updated. You can omit the clause, give a search
condition, or name a cursor. If you omit the clause, all rows of the table or view
are updated.

search-condition

Is any search condition described in “Chapter 3. Language elements” on
page 43. Each *column-name* in the search condition, other than in a
subquery, must identify a column of the table or view. The table or view
that is the object of the UPDATE cannot be used to supply the values that
specify the rows to be updated.

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.

CURRENT OF *cursor-name*

Identifies the cursor to be used in the update operation. The cursor name
must identify a declared cursor as explained in “DECLARE CURSOR” on
page 634.

If the UPDATE statement is embedded in a program, the DECLARE
CURSOR statement must include a select-statement rather than a
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 OF clause of that SELECT

UPDATE

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 subselect.

The result table of the cursor must not be read-only.

When the UPDATE statement is executed, the cursor must be positioned on the row to be updated.

If the application process has another cursor positioned on the updated row, the position of that cursor is changed to be before the next row.

The successful or unsuccessful execution of a positioned update operation does not change the position of the cursor. However, it is possible for an error to make the position of the error invalid, in which case, the cursor is closed. It is also possible for an update operation to cause a rollback, in which case, the cursor is closed.

WITH

Specifies the isolation level used when locating the rows to be updated by the statement.

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 columns of the plan tables for the rows that contain information about this SQL statement.

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, if hints are used. For information on using optimization hints, such as enabling the system for optimization hints and setting valid hint values, see Section 5 (Volume 2) of *DB2 Administration 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 3. Language elements” on page 43.
- *Uniqueness constraints.* The updated row must conform to any constraints imposed on the table (or on the base table of the view) by any unique index on

an updated column. For a multiple-row update of a unique key, the uniqueness constraint is effectively checked at the end of the operation.

- *Referential constraints.* A nonnull update value of a foreign key must be equal to some value of the parent key of the parent table of the relationship.
- *Check constraints.* The table (or base table of the view) might have one or more check constraints. Each row updated must conform to the conditions imposed by those check constraints. Thus, each check condition must be true or unknown.
- *Field and validation procedures.* The updated row must conform to any constraints imposed by any field or validation procedures on the table (or on the base table of the view).
- *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 627.

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.

- *Triggers.* An UPDATE statement might cause triggers to be activated. A trigger might cause other statements to be executed or raise error conditions based on the update values.

|
|
|

#

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 above, see “SQL communication area (SQLCA)” on page 883.

#

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 statement. 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.

Updating keys of partitioning indexes: If an updated column is a partitioning key or part of a partitioning key and the update causes a row to move to a different partition, DB2 tries to take exclusive control of the following objects to perform the update:


```
EXEC SQL UPDATE DSN8610.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;
```


VALUES INTO

The VALUES INTO statement assigns one or more values to host variables.

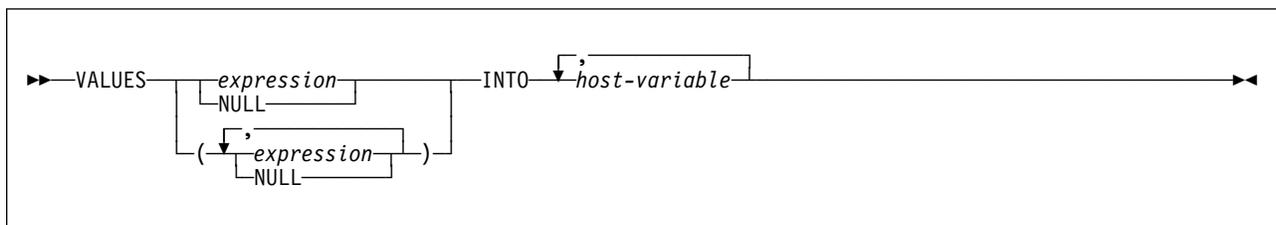
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

EXECUTE authority is needed on any user-defined function that is referenced in the VALUES statement.

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

Any expression of the type described in “Expressions” on page 131. 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 84. 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 883.)

host-variable

Identifies a variable that is described in the program according to the rules for declaring host variables.

Notes

The encoding scheme of the data (ASCII or EBCDIC) is the value of field DEF ENCODING SCHEME on installation panel DSNTIPF.

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.

Local special registers can be referenced only in a VALUES INTO statement that
results in the assignment of a single host variable and not those that result in
setting more than one value.

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: 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;
```

WHENEVER

The WHENEVER statement specifies the host language statement to be executed when a specified exception condition occurs.

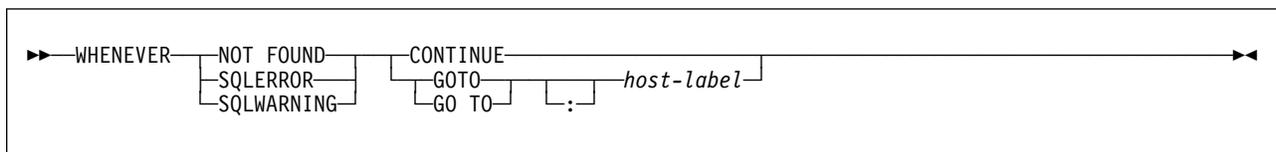
Invocation

This statement can only be embedded in an application program, except in REXX programs. It is not an executable statement.

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*.

WHENEVER

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.

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 7. SQL procedure statements

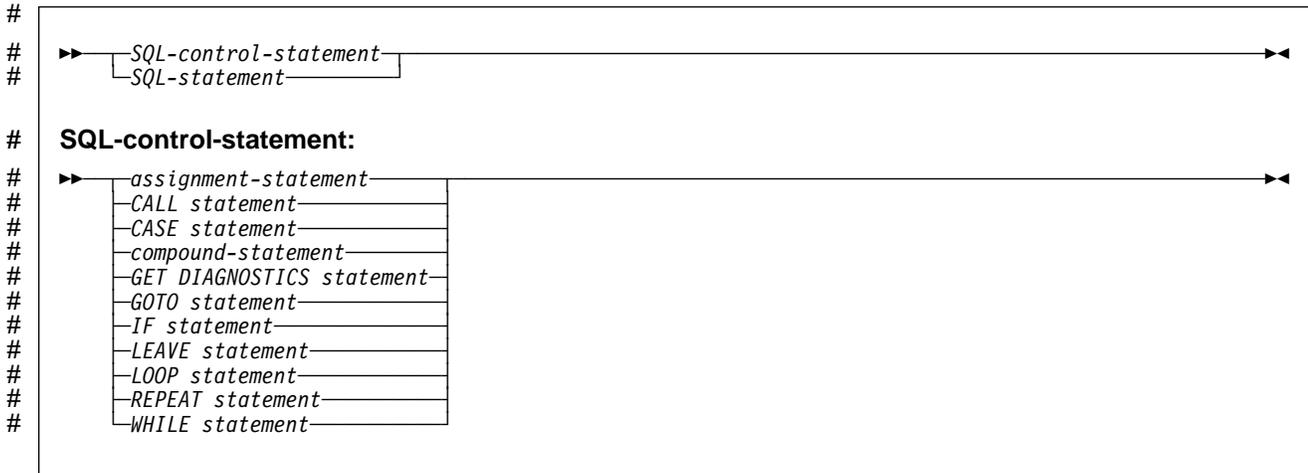
An SQL procedure consists of a CREATE PROCEDURE statement with a
procedure body. The procedure body contains the source statements for the stored
procedure, which are called *SQL procedure statements*.

This chapter contains syntax diagrams, semantic descriptions, rules, and examples
of the use of the statements that constitute the procedure body.

SQL-procedure-statement

If an SQL control statement is specified as the procedure body, multiple statements
 # can be specified within the 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 chapter. Control statements are supported in SQL procedures.

SQL-statement
 # Specifies an SQL statement as listed in Table 57 on page 879. These
 # statements are described in “Chapter 6. Statements” on page 337.

Notes

Comments: Comments can be included within the body of an SQL procedure. A
 # comment begins with /* and ends with */. The following rules apply:

- # • The beginning characters /* must be on the same line.
- # • The ending characters */ must be on the same line.
- # • Comments can be started wherever a space is valid.
- # • Comments can be continued to the next line.

Resolving names: The name of an SQL parameter or SQL variable in an SQL
 # procedure can be the same as the name of an identifier used in certain SQL
 # statements. If the name is not qualified, the following rules describe whether the
 # name refers to the identifier or to the SQL parameter or SQL variable:

- # • In the SET PATH statement, the name is checked as an SQL parameter or
 # SQL variable name. If not found as an SQL variable or SQL parameter name, it
 # will then be used as an identifier.
- # • In the CONNECT statement, the name is used as an identifier.

Assignment Statement (SQL procedure)

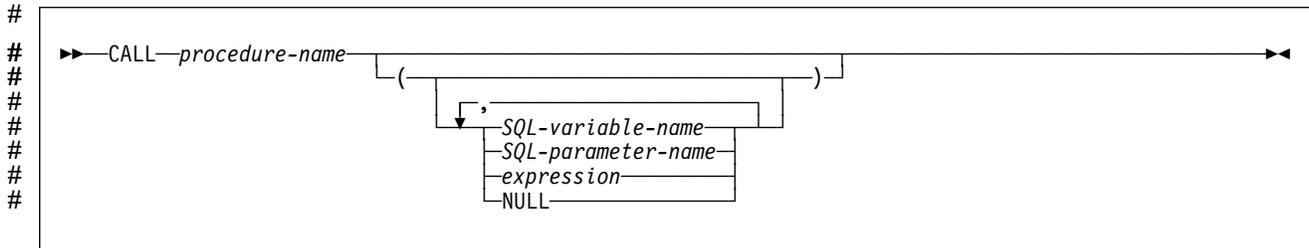
Examples

```
# 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



Description

procedure-name

Identifies the stored procedure to call. The procedure name must identify a

stored procedure that exists at the current server.

Parameters (*SQL-variable-name*, *SQL-parameter-name*, *expression*, **NULL**)

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 428 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 or *SQL-parameter-name*

Identifies a parameter to pass to or from the stored procedure. The data

type must be compatible with the data type of the corresponding parameter

in the stored procedure.

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 131
- # for *expression*, *expression* cannot include a column name or column
- # function or a user-defined function that is sourced on a column function.

CALL (SQL procedure)

• 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

If a CALL statement is the only statement in the procedure body, the statement
cannot end with a semicolon. Otherwise, the statement must end with a semicolon.

See “CALL” on page 428 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 (SQL procedure)

```
#          search-condition
#          Specifies a condition that is true, false, or unknown about a row or group of
#          table data. The search condition cannot contain a subselect.
#
#          END CASE
#          Ends a case-statement.
```

Notes

```
#          If none of the conditions specified in the WHEN 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.
```

```
#          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.
```

```
#          If a CASE statement is the only statement in the procedure body, the statement
#          cannot end with a semicolon. Otherwise, the statement must end with a semicolon.
```

```
#          Ensure that your CASE statement covers all possible execution conditions.
```

Examples

```
#          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
```

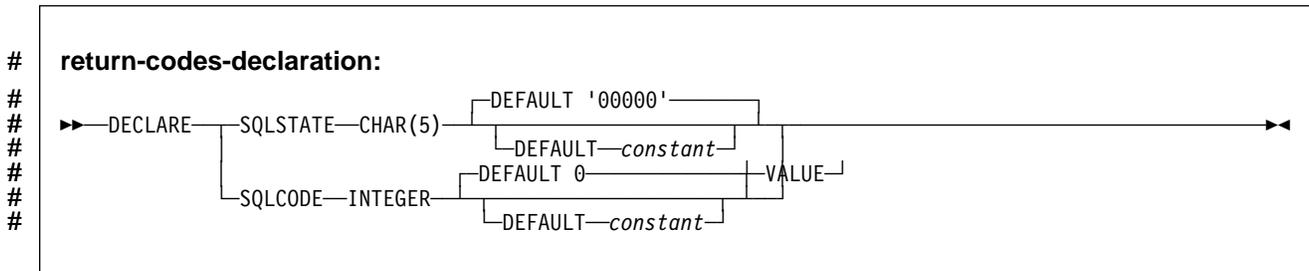
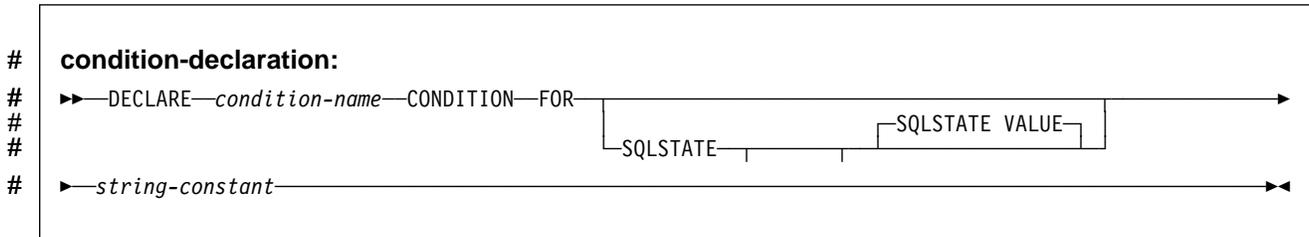
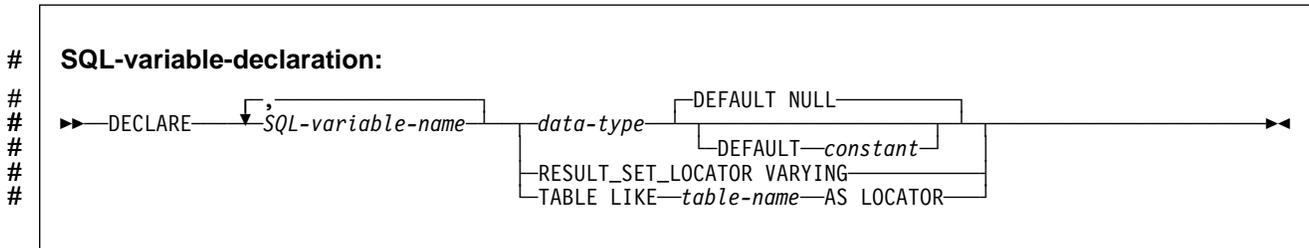
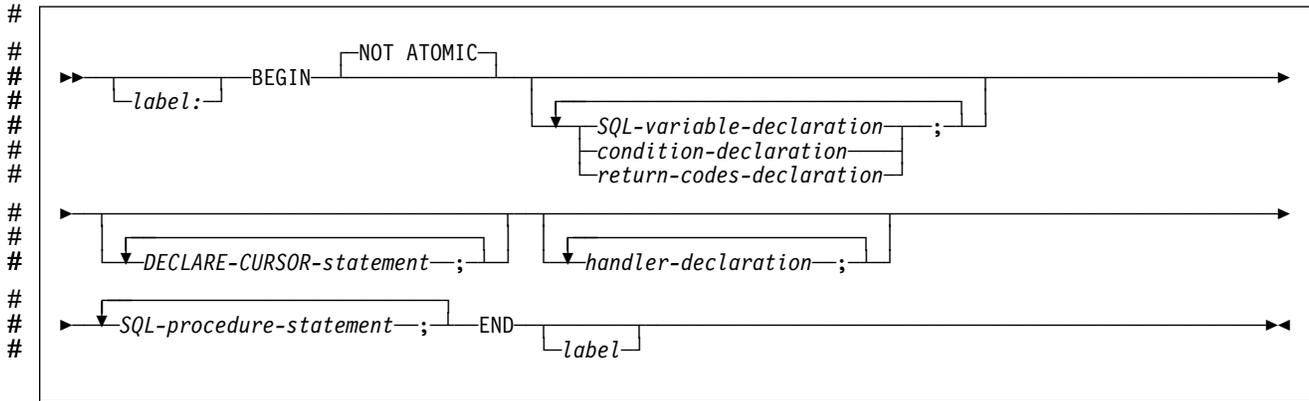
```
#          Use a searched case statement when clause to update column DEPTNAME in
#          table DEPT, depending on the value of SQL variable v_workdept.
```

```
#          CASE
#             WHEN v_workdept = 'A00'
#             THEN UPDATE department SET
#                deptname = 'DATA ACCESS 1';
#             WHEN v_workdept = 'B01'
#             THEN UPDATE department SET
#                deptname = 'DATA ACCESS 2';
#             ELSE UPDATE department 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



Compound Statement (SQL procedure)

```
# handler-declaration:
# ▶ DECLARE CONTINUE HANDLER FOR repeatable-handler-condition
#           EXIT      └─┬─┘
#                       └─┬─┘ nonrepeatable-handler-condition
# ▶ SQL-procedure-statement
```

```
# specific-handler-value:
# ▶ SQLSTATE VALUE string
#           └─┬─┘
#             condition-name
```

```
# general-condition-value:
# ▶ SQL EXCEPTION
#   SQL WARNING
#   NOT FOUND
```

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.

A label name cannot be the same as the name of the SQL procedure in which the label is used.

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 an SQL variable name is an SQL identifier of 1 to 18 bytes. If the SQL variable is a delimited identifier, the contents of the delimited identifier must conform to the rules for ordinary identifiers. The qualified form is an SQL procedure statement label followed by a period (.) and an SQL identifier.

DB2 folds all SQL variable names to uppercase. If an SQL reserved word is used as an SQL variable, the SQL variable must be delimited. 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)" on page 555 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.

#           TABLE LIKE table-name AS LOCATOR
#           Specifies the data type for a table locator. table-name identifies the table that
#           the locator is defined for.

#           condition-declaration
#           Declares a condition name and corresponding SQLSTATE value.

#           condition-name
#           Specifies the name of the condition. The condition name is a long SQL
#           identifier that 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 634 .

#           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 "Chapter
#           7. SQL procedure statements" on page 847 for information on
#           SQL-procedure-statement.

```

Compound Statement (SQL procedure)

A handler is active only within the compound statement in which it is declared.
The actions that a handler can perform are:

CONTINUE
After the handler is invoked successfully, control is returned to the SQL
statement that follows the statement that raised the exception. If the error
that raised the exception is an IF, CASE, WHILE, or REPEAT statement,
control returns to the statement that follows END IF, END CASE, END
WHILE, or END REPEAT.

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*
Specifies an SQLSTATE for which the handler is invoked. The SQLSTATE
cannot be '00000'.

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
Appendix C of *DB2 Messages and 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 and condition 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 an error occurs for which there is no handler, execution of the compound statement is terminated.
- 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 special 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.

If a compound statement is the only statement in the procedure body, the statement cannot end with a semicolon. Otherwise, the statement must end with a semicolon.

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

Compound Statement (SQL procedure)

```
#          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;
#          CLOSE c1;
#          END
```

GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement obtains information about the previous SQL
statement that was executed.

Syntax

```
#
# ──▶ GET DIAGNOSTICS ── SQL-variable-name ── = ── ROW_COUNT ── ◀─▶
```

Description

SQL-variable-name
Identifies the SQL variable that is the assignment target. The SQL variable
must be declared as an integer variable. For information on declaring SQL
variables, see “Compound statement” on page 855.

ROW_COUNT
Identifies the number of rows that are associated with the previous SQL
statement that was executed. If the previous SQL statement is a DELETE,
INSERT, or UPDATE statement, ROW_COUNT identifies the number of rows
that were deleted, inserted, or updated by the SQL statement. That number
does not include rows that were deleted, inserted, or updated because of
referential constraints or triggered actions. If the previous statement is another
SQL statement, the value that is returned has no meaning.

Notes

The GET DIAGNOSTICS statement does not change the contents of the SQLCA. If
SQLCODE and SQLSTATE variables are declared in the SQL procedure, those
variables contain the SQLCODE and SQLSTATE from the previous SQL statement.

Examples

Use a GET DIAGNOSTICS statement to determine how many rows were updated
by the previous SQL statement.

```
# BEGIN
# DECLARE rcount INTEGER;
# UPDATE PROJ
# SET PRSTAFF = PRSTAFF + 1.5
# WHERE DEPTNO = deptnbr;
# GET DIAGNOSTICS rcount = ROW_COUNT;
# END
```

GOTO (SQL procedure)

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 labelled statement at which processing is to continue.
#                         The labelled 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. label cannot be in a nested
#                         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 EXIT1;
# 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;
# EXIT1: SET return_parm = service;
# END
```

LEAVE statement

The LEAVE statement transfers program control out of a loop or a block of code.

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.

If a LEAVE statement is the only statement in the procedure body, the statement
cannot end with a semicolon. Otherwise, the statement must end with a semicolon.

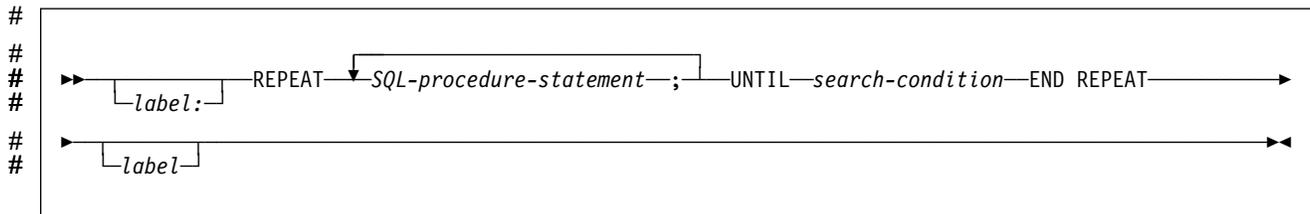
Examples

Use a LEAVE statement to transfer control out of a LOOP statement when a
negative SQLCODE occurs.

```
# ftch_loop: LOOP
#   FETCH c1 INTO
#     v_firstnme, v_midinit,
#     v_lastname, v_edlevel, v_salary;
#   IF SQLCODE=100 THEN LEAVE ftch_loop;
#   END IF;
# END LOOP
```


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 848.

search-condition

Specifies a condition that is evaluated after each execution of the SQL procedure statement. If the condition is true, the SQL procedure statement is not executed again.

Notes

If a REPEAT statement is the only statement in the procedure body, the statement cannot end with a semicolon. Otherwise, the statement must end with a semicolon.

Examples

Use a REPEAT statement to fetch rows from a table.

```
#
# fetch_loop:
# REPEAT
#   FETCH c1 INTO
#     v_firstnme, v_midinit, v_lastname;
# UNTIL
#   SQLCODE <> 0
# END REPEAT fetch_loop
```


Appendix A. Limits in DB2 for OS/390

System storage limits might preclude the limits specified here. The limit for items not specified below is system storage.

Table 49. Identifier length limits

Item	Limit
Longest collection ID, correlation name, statement name, or name of an alias, column, cursor, index, table, table check constraint, stored procedure, synonym, user-defined function, view name	18 bytes
Longest authorization name, or name of a database, package, plan, referential constraint, schema, storage group, tablespace name, or trigger	8 bytes
Longest host identifier	64 bytes
Longest server name or location identifier	16 bytes

Table 50. Numeric limits

Item	Limit
Smallest SMALLINT value	-32768
Largest SMALLINT value	32767
Smallest INTEGER value	-2147483648
Largest INTEGER value	2147483647
Smallest REAL value	About -7.2×10^{75}
Largest REAL value	About 7.2×10^{75}
Smallest positive REAL value	About 5.4×10^{-79}
Largest negative REAL value	About -5.4×10^{-79}
Smallest FLOAT value	About -7.2×10^{75}
Largest FLOAT value	About 7.2×10^{75}
Smallest positive FLOAT value	About 5.4×10^{-79}
Largest negative FLOAT value	About -5.4×10^{-79}
Smallest DECIMAL value	$1 - 10^{31}$
Largest DECIMAL value	$10^{31} - 1$
Largest decimal precision	31

Table 51 (Page 1 of 2). String length limits

Item	Limit
Maximum length of CHAR	255 bytes
Maximum length of GRAPHIC	127 DBCS characters
Maximum length of VARCHAR ³⁹	4046 bytes for 4-KB pages 8128 bytes for 8-KB pages 16320 bytes for 16-KB pages 32704 bytes for 32-KB pages

Limits in DB2 for OS/390

Table 51 (Page 2 of 2). String length limits

Item	Limit
Maximum length of VARGRAPHIC ³⁹	4046 bytes (2023 DBCS characters) for 4-KB pages 8128 bytes (4064 DBCS characters) for 8-KB pages 16320 bytes (8160 DBCS characters) for 16-KB pages 32704 bytes (16352 DBCS characters) for 32-KB pages
Maximum length of CLOB	2 147 483 647 bytes (2 gigabytes - 1 byte)
Maximum length of DBCLOB	1 073 741 824 DBCS characters
Maximum length of BLOB	2 147 483 647 bytes (2 gigabytes - 1 byte)
Maximum length of a character constant	255 bytes
Maximum length of a hexadecimal constant	254 digits
Maximum length of a graphic string constant	124 DBCS characters
Maximum length of a concatenated character string	2 147 483 647 bytes (2 gigabytes - 1 byte)
Maximum length of a concatenated graphic string	1 073 741 824 DBCS characters
Maximum length of a concatenated binary string	2 147 483 647 bytes (2 gigabytes - 1 byte)

Table 52. 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

Table 53 (Page 1 of 2). DB2 limits on SQL statements

Item	Limit
Maximum number of columns 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 749 if the table is a dependent
Maximum number of base tables in a view, SELECT, UPDATE, INSERT, or DELETE	225
Maximum row and record sizes for a table	See Maximum record size on page 592 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 2G 254 for table spaces that are defined with LARGE or a DSSIZE greater than 2G

³⁹ 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.

Table 53 (Page 2 of 2). DB2 limits on SQL statements

Item	Limit
Maximum size of a partition (table space or index)	For table spaces that are not defined with LARGE or a DSSIZE greater than 2G: 4 gigabytes, for 1 to 16 partitions 2 gigabytes, for 17 to 32 partitions 1 gigabyte, for 33 to 64 partitions For table spaces that are defined with LARGE: 4 gigabytes, for 1 to 254 partitions For table spaces that are defined with a DSSIZE greater than 2G: 64 gigabytes, for 1 to 254 partitions
Maximum size of a DBRM entry	131072 bytes
Longest index key	255 bytes less the number of key columns that allow nulls.
Maximum number of bytes used in the partitioning of a partitioned index ⁴⁰	255
Maximum number of columns in an index key	64
Maximum number of tables in a FROM clause	15
Maximum number of subqueries in a statement	14
Maximum total length of host and indicator variables pointed to in an SQLDA	32767 bytes 2 147 483 647 bytes (2 gigabytes - 1 byte) for a LOB, subject to the limitations imposed by the application environment and host language
Longest host variable used for insert or update	32704 bytes for a non-LOB 2 147 483 647 bytes (2 gigabytes - 1 byte) for a LOB, subject to the limitations imposed by the application environment and host language
Longest SQL statement	32765 bytes
Maximum number of elements in a select list	750
Maximum number of predicates in a WHERE or HAVING clause	750
Maximum total length of columns of a query operation requiring a sort key (SELECT DISTINCT, ORDER BY, GROUP BY, UNION without the ALL keyword, and the DISTINCT column function)	4000 bytes
Maximum length of a table 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 gigabytes - 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	254 bytes

Limits in DB2 for OS/390

Table 54. 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 used by each DB2 or application agent
Largest table or table space	16 terabytes
Largest log space	248
Largest active log data set	2 gigabytes
Largest archive log data set	2 gigabytes
Maximum number of active log copies	2
Maximum number of archive log copies	2
Maximum number of active log data sets (each copy)	31
Maximum number of archive log volumes (each copy)	1000
Maximum number of databases accessible to an application or end user	Limited by system storage and EDM pool size
Largest EDM pool	The installation parameter maximum depends on available space
Maximum number of databases	65279
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 gigabytes
Maximum partitioned data set size	See item "maximum size of a partition" in Table 53 on page 870
Maximum LOB data set size	64 gigabytes

⁴⁰ The maximum length of the key for a partitioning index is 255 bytes; all 255 bytes can be used to determine the partition.

Appendix B. Characteristics of SQL statements in DB2 for OS/390

#

This appendix provides a summary of the actions that are allowed on SQL statements in DB2 for OS/390. It also contains a list of the SQL statements that can be executed in external user-defined functions and stored procedures and in SQL procedures.

Actions allowed on SQL statements

Table 55 shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the application requester, the application server, or the precompiler. The letter Y means yes.

Table 55 (Page 1 of 4). Actions allowed on SQL statements in DB2 for OS/390

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 ON	Y	Y		Y	
COMMIT	Y	Y		Y	
CONNECT (Type 1 and Type 2)	Y		Y		
CREATE ²	Y	Y		Y	
DECLARE CURSOR					Y
# DECLARE GLOBAL # TEMPORARY TABLE	Y	Y		Y	
DECLARE STATEMENT					Y
DECLARE TABLE					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	

Characteristics of SQL statements in DB2 for OS/390

Table 55 (Page 2 of 4). Actions allowed on SQL statements in DB2 for OS/390

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
EXPLAIN	Y	Y		Y	
FETCH	Y			Y	
FREE LOCATOR ¹	Y	Y		Y	
GRANT ²	Y	Y		Y	
HOLD LOCATOR ¹	Y	Y		Y	
INCLUDE					Y
INSERT	Y	Y		Y	
LABEL ON	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
OPEN	Y			Y	
PREPARE	Y			Y ⁴	
RELEASE connection	Y		Y		
# RELEASE SAVEPOINT	Y	Y		Y	
RENAME ²	Y	Y		Y	
REVOKE ²	Y	Y		Y	
ROLLBACK	Y	Y		Y	
# SAVEPOINT	Y	Y		Y	
SELECT INTO	Y			Y	
SET CONNECTION	Y		Y		
SET CURRENT DEGREE	Y	Y		Y	
SET CURRENT LC_CTYPE	Y	Y		Y	
SET CURRENT OPTIMIZATION HINT	Y	Y		Y	
SET CURRENT PACKAGESET	Y		Y		
SET CURRENT PATH	Y	Y		Y	
SET CURRENT PRECISION	Y	Y		Y	
SET CURRENT RULES	Y	Y		Y	
SET CURRENT SQLID ⁵	Y	Y		Y	
SET <i>host-variable</i> = CURRENT DATE	Y			Y	
SET <i>host-variable</i> = CURRENT DEGREE	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	

Table 55 (Page 3 of 4). Actions allowed on SQL statements in DB2 for OS/390

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
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 <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 SQLSTATE ⁶	Y			Y	
UPDATE	Y	Y		Y	
VALUES ⁶	Y			Y	
VALUES INTO ⁷	Y			Y	
WHENEVER					Y

Characteristics of SQL statements in DB2 for OS/390

Table 55 (Page 4 of 4). Actions allowed on SQL statements in DB2 for OS/390

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler

Notes:

- | 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 and the implicit qualifier 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 only be used 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.

SQL statements allowed in external functions and stored procedures

Table 56 shows which SQL statements a external stored procedure or external user-defined function can execute. The statements that can be executed depend on the level of SQL data access with which the stored procedure or external function is defined (NO SQL, CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA). 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 56 (Page 1 of 3). 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 ON				Y
COMMIT				
CONNECT (Type 1 and Type 2)		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
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 ³	Y ³	Y
EXECUTE IMMEDIATE		Y ³	Y ³	Y
EXPLAIN				Y

Table 56 (Page 2 of 3). 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
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GRANT				Y
HOLD LOCATOR		Y	Y	Y
INCLUDE	Y ¹	Y	Y	Y
INSERT				Y
LABEL ON				Y
LOCK TABLE		Y	Y	Y
OPEN			Y	Y
PREPARE		Y	Y	Y
RELEASE connection		Y	Y	Y
RELEASE SAVEPOINT ⁵				Y
REVOKE				Y
ROLLBACK ^{5, 6}				
SAVEPOINT ⁵				Y
SELECT			Y	Y
SELECT INTO			Y	Y
SET CONNECTION		Y	Y	Y
SET host-variable Assignment		Y ⁴	Y	Y
SET special register		Y	Y	Y
SET transition-variable Assignment		Y ⁴	Y	Y
SIGNAL SQLSTATE		Y	Y	Y
UPDATE				Y
VALUES			Y	Y
VALUES INTO		Y ⁴	Y	Y
WHENEVER	Y ¹	Y	Y	Y

Table 56 (Page 3 of 3). SQL statements in external user-defined functions and stored procedures

SQL statement	Level of SQL access		
	NO SQL	CONTAINS SQL	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, or CONTAINS SQL. A routine defined as CONTAINS SQL can only call a procedure defined as CONTAINS SQL.
3. 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, or DELETE.
4. The statement is supported only if it does not contain a subquery or query-expression.
5. RELEASE SAVEPOINT, SAVEPOINT, and ROLLBACK (with the TO SAVEPOINT clause) cannot be executed from a user-defined function.
6. The ROLLBACK statement without the TO SAVEPOINT clause can be executed in a stored procedure. However, an error is returned to the calling program and the application is placed in a "must rollback" state.

SQL statements allowed in SQL procedures

Table 57 lists the SQL statements that are valid in an SQL procedure body, in addition to SQL procedure statements. The table lists the SQL statements that can be used as the only statement in the SQL procedure and the statements that can be nested in a compound statement. Whether an SQL statement can be executed in an SQL procedure also depends on whether MODIFIES SQL DATA, CONTAINS SQL, or READS SQL DATA is specified in the stored procedure definition. See Table 56 on page 877 for a list of SQL statements that can be executed for each of these parameter values.

Table 57 (Page 1 of 3). 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		
ALTER INDEX	Y	Y
ALTER PROCEDURE		
ALTER STOGROUP	Y	Y
ALTER TABLE	Y	Y
ALTER TABLESPACE	Y	Y
ASSOCIATE LOCATORS		Y

Table 57 (Page 2 of 3). Valid SQL statements in an SQL procedure body

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
BEGIN DECLARE SECTION		
CALL		Y
CLOSE		Y
COMMENT ON	Y	Y
COMMIT		
CONNECT (Type 1 and Type 2)	Y	Y
CREATE ALIAS	Y	Y
CREATE DATABASE	Y	Y
CREATE DISTINCT TYPE		
CREATE FUNCTION		
CREATE GLOBAL TEMPORARY TABLE	Y	Y
CREATE INDEX	Y	Y
CREATE PROCEDURE		
CREATE STOGROUP	Y	Y
CREATE SYNONYM	Y	Y
CREATE TABLE	Y	Y
CREATE TABLESPACE	Y	Y
CREATE TRIGGER		
CREATE VIEW	Y	Y
DECLARE CURSOR		Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y
DECLARE STATEMENT		
DECLARE TABLE		
DELETE	Y	Y
DESCRIBE (prepared statement or table)		
DESCRIBE CURSOR		
DESCRIBE INPUT		
DESCRIBE PROCEDURE		
DROP	Y	Y
END DECLARE SECTION		
EXECUTE		Y
EXECUTE IMMEDIATE	Y	Y
EXPLAIN		
FETCH		Y
FREE LOCATOR		
GRANT	Y	Y

Table 57 (Page 3 of 3). Valid SQL statements in an SQL procedure body

SQL statement	SQL statement is...	
	The only statement in the procedure	Nested in a compound statement
HOLD LOCATOR		
INCLUDE		
INSERT	Y	Y
LABEL ON	Y	Y
LOCK TABLE	Y	Y
OPEN		Y
PREPARE FROM		Y
RELEASE connection	Y	Y
RELEASE SAVEPOINT	Y	Y
RENAME	Y	Y
REVOKE	Y	Y
ROLLBACK		
ROLLBACK TO SAVEPOINT	Y	Y
SAVEPOINT	Y	Y
SELECT		
SELECT INTO	Y	Y
SET CONNECTION	Y	Y
SET host-variable Assignment ¹		
SET special register ¹	Y	Y
SET transition-variable Assignment ¹		
SIGNAL SQLSTATE		
UPDATE	Y	Y
VALUES		
VALUES INTO	Y	Y
WHENEVER		

Notes:

1. SET host-variable Assignment, SET transition-variable Assignment, and SET special register are the SQL SET statements that are described in "Chapter 6. Statements" on page 337, not the SQL procedure assignment statement that is described in "Assignment statement" on page 849.

Appendix C. SQLCA and SQLDA

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. There are two exceptions:

- A program that is precompiled with the STDSQL(YES) option must not provide an SQLCA
- In some cases (as discussed below in In Fortran), a Fortran program must provide more than one SQLCA.

In all host languages except REXX, the SQL INCLUDE statement can be used to provide the declaration of the SQLCA.

In COBOL and Assembler: The name of the storage area must be SQLCA.

In PL/I, and C: The name of the structure must be SQLCA. Every executable SQL statement must be within the scope of its declaration.

Unless noted otherwise, C is used to represent C/370™ and C/C++ for MVS/ESA 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 888 for more information on the REXX SQLCA.

Description of fields

The names in the following table are those provided by the SQL INCLUDE statement. For the most part, COBOL, C, PL/I, and Assembler use the same names, and Fortran names are different. However, there is one instance where C, PL/I, and Assembler names differ from COBOL.

Table 58 (Page 1 of 3). 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'.
SQLCABC	sqlcab	Not used.	INTEGER	Contains the length of the SQLCA: 136.

SQLCA

Table 58 (Page 2 of 3). Fields of SQLCA

Assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose								
SQLCODE (See note 1)	SQLCODE	SQLCOD	INTEGER	Contains the SQL return code. (See note 2) <table border="0"> <tr> <td>Code</td> <td>Means</td> </tr> <tr> <td>0</td> <td>Successful execution (though there might have been warning messages).</td> </tr> <tr> <td>positive</td> <td>Successful execution, but with a warning condition or other information</td> </tr> <tr> <td>negative</td> <td>Error condition.</td> </tr> </table>	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.
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.								
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 OS/390.								
SQLERRD(1)	sqlerrd[0]	SQLERR(1)	INTEGER	Contains an internal error code.								
SQLERRD(2)	sqlerrd[1]	SQLERR(2)	INTEGER	Contains an internal error code.								
SQLERRD(3)	sqlerrd[2]	SQLERR(3)	INTEGER	Contains the number of rows affected after INSERT, UPDATE, and DELETE (but not rows deleted as a result of CASCADE delete). 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 or from a table created with CREATE GLOBAL TEMPORARY TABLE.. For SQLCODES -911 and -913, SQLERRD(3) will contain the reason code for the deadlock and timeout.								
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 OS/390.								
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 W if at least one other indicator also contains a W; otherwise, contains a blank.								
SQLWARN1	SQLWARN1	SQLWRN(1)	CHAR(1)	Contains a W if the value of a string column was truncated when assigned to a host variable.								
SQLWARN2	SQLWARN2	SQLWRN(2)	CHAR(1)	Contains a W if null values were eliminated from the argument of a column 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.								

Table 58 (Page 3 of 3). Fields of SQLCA

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 OS/390.
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.
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).

Note:

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 Section 2 of *DB2 Messages and 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 885.
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 Appendix C of *DB2 Messages and 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. Table 59 lists the
 # variables in a REXX SQLCA.

Table 59 (Page 1 of 2). Variables in a REXX SQLCA

# Variable	Contents
# SQLCODE	The SQL return code.
# SQLERRMC	One or more tokens, separated by X'FF', that are substituted for variables in the descriptions of error conditions.
# SQLERRP	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 OS/390, the product signature is 'DSN'.
# SQLERRD.1	An internal error code.
# SQLERRD.2	An internal error code.
# SQLERRD.3	The number of rows affected after INSERT, UPDATE, and DELETE (but not rows deleted as a result of CASCADE delete). 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 SQLCODE -911 or -913, SQLERRD.3 can also contain the reason code for a timeout or deadlock.

Table 59 (Page 2 of 2). Variables in a REXX SQLCA

# Variable	Contents
# 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 OS/390.
# SQLERRD.5 #	The position or column of a syntax error for a PREPARE or EXECUTE IMMEDIATE statement.
# SQLERRD.6	An internal error code.
# SQLWARN.0 #	Blank if all other indicators are blank; W if at least one other indicator also contains a W.
# SQLWARN.1	W if the value of a string column was truncated when assigned to a host variable.
# SQLWARN.2 # #	W if null values were eliminated from the argument of a column function; not necessarily set to W for the MIN function because its results are not dependent on the elimination of null values.
# SQLWARN.3 # #	W if the number of result columns is larger than the number of host variables. Z if the ASSOCIATE LOCATORS statement contains fewer locators than the stored procedure returned.
# SQLWARN.4	W if a prepared UPDATE or DELETE statement does not include a WHERE clause.
# SQLWARN.5 #	W if the SQL statement was not executed because it is not a valid SQL statement in DB2 for OS/390.
# SQLWARN.6 # #	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 #	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	W if a character that could not be converted was replaced with a substitute character.
# SQLWARN.9 #	W if arithmetic exceptions were ignored during COUNT or COUNT_BIG processing. Z if the stored procedure returned multiple result sets.
# SQLWARN.10 #	W if at least one character field of the SQLCA is invalid due to a character conversion error.
# SQLSTATE	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*.

The following sections discuss the fields of the SQLDA and the format of the
 # SQLDA for each language. Because the fields and format of the SQLDA for REXX
 # is somewhat different from the SQLDAs for other languages, the REXX SQLDA is
 # discussed separately.

Field descriptions

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 next section describes the SQLDA header, and “SQLVAR entries” on page 893 describes the SQLVAR and how to determine how many SQLVAR entries to allocate in an SQLDA.

The SQLDA Header

Table 60 describes the fields in the SQLDA header.

Table 60 (Page 1 of 2). 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, OPEN, EXECUTE, and CALL
sqldaid SQLDAID	CHAR(8)	An “eye catcher” for storage dumps, containing the text 'SQLDA '. The 7th byte of the field is a flag that can be used to determine if more than one SQLVAR entry is needed for each column. For details, see “Determining how many SQLVAR occurrences are needed” on page 894. For DESCRIBE CURSOR, the field is set to 'SQLRS'. If the cursor is declared WITH HOLD in a stored procedure, the high-order bit of the 8th byte is set to 1. For DESCRIBE PROCEDURE, it is set to 'SQLPR'.	A plus sign (+) in the 6th byte indicates that SQLNAME contains an overriding CCSID. A '2' in the 7th byte indicates the two SQLVAR entries were allocated for each column or parameter. A '3' in the 7th byte indicates that three SQLVAR entries were allocated for each column or parameter. Although three entries are never needed on input to DB2, an SQLDA with three entries might be used when the SQLDA was initialized by a DESCRIBE or PREPARE INTO with a USING BOTH clause. Otherwise, SQLDAID is not used.
sqldabc SQLDABC	INTEGER	Length of the SQLDA, equal to SQLN×44+16.	Length of the SQLDA, greater than or equal to SQLN×44+16.

Table 60 (Page 2 of 2). 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, 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^s 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. 	<p>Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero.</p>
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>	<p>The number of host variables described by occurrences of SQLVAR.</p>

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.

SQLVAR entries

For each column or host variable described by the SQLDA, there are two types of 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 type⁴¹ 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 894.

Table 61 on page 894 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 SQLD⁴² even though many of the extended SQLVAR entries might be unused.

⁴¹ DESCRIBE INPUT does not return information about distinct types.

⁴² 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 61. Contents of SQLVAR arrays

LOBs	Distinct types ¹	7th byte of SQLDAID	SQLD	Minimum for SQLN ²	Set of SQLVAR entries		
					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 or labels	Not Used	Not Used
Yes ³	Yes ³	2	<i>n</i>	<i>2n</i>	Column names or 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

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. See the “*Minimum Number of SQLVARs Needed*” column in Table 61.

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: Table 62 describes the contents of the fields of a base SQLVAR.

Table 62 (Page 1 of 2). 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	<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 Table 64 on page 900.</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>	<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 899.</p>
sqlen SQLLEN	SMALLINT	<p>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 899 for a description of allowable values.</p> <p>For LOBs, the value is 0 regardless of the length attribute of the LOB. Field SQLLONGLEN in the extended SQLVAR contains the length attribute.</p>	<p>The length attribute of the host variable. See "SQLTYPE and SQLLEN" on page 899 for a description of allowable values.</p> <p>For LOBs, the value is 0 regardless of the length attribute of the LOB. Field SQLLONGLEN in the extended SQLVAR contains the length attribute.</p>
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 DESCRIBE PROCEDURE, the result set locator value associated with the result set.</p>	<p>Contains the address of the host variable.</p>
sqlind SQLIND	pointer	<p>Reserved</p> <p>For DESCRIBE PROCEDURE, it is set to -1.</p>	<p>Contains the address of an associated indicator variable, if SQLTYPE is odd. Otherwise, the field is not used.</p>

Table 62 (Page 2 of 2). 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
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 prepared statement includes a UNION or UNION ALL clause, SQLNAME contains the name or label, if any, of the corresponding column of the first operand of the UNION.</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 a CCSID. DB2 interprets the third and fourth byte of SQLNAME as the CCSID of the host variable if all of the following are true:</p> <ul style="list-style-type: none"> • The 6th byte of SQLDAID is '+' • SQLTYPE indicates the host variable is a string variable • The length of SQLNAME is 8 • The first two bytes of SQLNAME are X'0000'.

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: Table 63 describes the contents of the fields of an extended SQLVAR entry.

Table 63 (Page 1 of 2). Fields in an occurrence of an extended SQLVAR

C name Assembler, COBOL, or PL/I name	Data type	Usage in DESCRIBE¹ and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
len.sqllonglen SQLLONGL SQLLONGLEN	INTEGER	The length attribute of a LOB (BLOB, CLOB, or DBCLOB) column.	The length attribute of a LOB (BLOB, CLOB, or DBCLOB) host variable. DB2 ignores the SQLLEN field in the base SQLVAR for these data types. The length attribute indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB.
*	INTEGER	Reserved.	Reserved.
sqldatalen SQLDATAL SQLDATALEN	pointer	Not used.	Used only for LOB (BLOB, CLOB, and DBCLOB) host variables. If the value of the field is null, the actual length of the LOB is stored in the 4 bytes immediately before the start of the data, and SQLDATA points to the first byte of the field length. The actual length indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB. If the value of the field is not null, the field contains a pointer to a 4-byte long buffer that contains the actual length <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 63 (Page 2 of 2). 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
sqldatatype_name SQLTNAME SQLDATATYPE-NAME	VARCHAR(30)	<p>A SQLTNAME field of the appropriate extended SQLVAR is set to one of the following as per Table 61 on page 894.</p> <ul style="list-style-type: none"> For a distinct type column, DB2 sets this to the fully qualified distinct type name. The first 8 bytes contain the schema name of the type, (extended to the right with spaces, if necessary). Byte 9 contains a period (.). Bytes 10 to 27 contain the low order portion of the type name, which is not extended to the right with spaces. <p>Otherwise, schema name is SYSIBM and the low order portion of the name is the name of the type from the catalog.</p> <p>DESCRIBE INPUT does not return information about distinct type types.</p> <ul style="list-style-type: none"> For a label, this field is set to the contents of the 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.

Notes:

- 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 following table shows the values that may 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 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.

SQLDA

Table 64 (Page 1 of 2). *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
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	Not used. ³
408/409	CLOB	0 ³	CLOB	Not used. ³
412/413	DBCLOB	0 ³	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
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 varying-length graphic string	length attribute of the column	long graphic string	length attribute of the host variable
480/481	floating point	4 for single precision, 8 for double precision	floating point	4 for single precision, 8 for double precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
504/505	N/A	N/A	DISPLAY SIGN LEADING SEPARATE	precision in byte 1; scale in byte 2
904/905	N/A	N/A	ROWID	40
960/961	N/A	N/A	BLOB locator	4

Table 64 (Page 2 of 2). *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
964/965	N/A	N/A	CLOB locator	4
968/969	N/A	N/A	DBCLOB locator	4
972/973	result set locator	4	result set locator	4
976/977	table locator	4	table locator	4

Notes:

1. Might be different if a date installation exit is specified.
2. Might be different if a time installation exit is specified.
3. Field *SQLLONGLEN* in the extended *SQLVAR* contains the length attribute of the column.

SQLDATA: The following table identifies the *CCSID* values that appear in the *SQLDATA* field when the *SQLVAR* element describes a string column.

Table 65. *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

The included SQLDA

The description of the SQLDA that is given by INCLUDE SQLDA is shown below. Only Assembler, C, C++, COBOL⁴³, and PL/I C are supported.

Assembler:

```

SQLTRIPL EQU    C'3'
SQLDOUBL EQU    C'2'
SQLSINGL EQU    C' '
*
        SQLSECT SAVE
*
SQLDA    DSECT
SQLDAID DS    CL8      ID
SQLDABC DS    F        BYTE COUNT
SQLN     DS    H        COUNT SQLVAR/SQLVAR2 ENTRIES
SQLD     DS    H        COUNT VARS (TWICE IF USING BOTH)
*
SQLVAR   DS    0F      BEGIN VARS
SQLVARN  DSECT ,      NTH VARIABLE
SQLTYPE  DS    H        DATA TYPE CODE
SQLLEN   DS    0H      LENGTH
SQLPRCSN DS    X        DEC PRECISION
SQLSCALE DS    X        DEC SCALE
SQLDATA  DS    A        ADDR OF VAR
SQLIND   DS    A        ADDR OF IND
SQLNAME  DS    H,CL30  DESCRIBE NAME
SQLVSIZ  EQU    *-SQLDATA
SQLSIZV  EQU    *-SQLVARN
*
SQLDA    DSECT
SQLVAR2  DS    0F      BEGIN EXTENDED FIELDS OF VARS
SQLVAR2N DSECT ,      EXTENDED FIELDS OF NTH VARIABLE
SQLLONGL DS    F        LENGTH
SQLRSVDL DS    F        RESERVED
SQLDATAL DS    A        ADDR OF LENGTH IN BYTES
SQLTNAME DS    H,CL30  DESCRIBE NAME
*
        SQLSECT RESTORE

```

In the above declaration, SQLSECT SAVE is a macro invocation that remembers the current CSECT name. SQLSECT RESTORE is a macro invocation that continues that CSECT.

⁴³ Excluding OS/VS COBOL

C and C++:

```

#ifndef SQLDASIZE                /* Permit duplicate Includes */
/**/
struct sqlvar
{ short  sqltype;
  short  sqlllen;
  char   *sqldata;
  short  *sqlind;
  struct sqlname
    { short  length;
      char   data[30];
    } sqlname;
};
/**/
struct sqlvar2
{ struct
    { long  sqllonglen;
      unsigned long  reserved;
    } len;
  char *sqldataalen;
  struct sqldistinct_type
    { short  length;
      char   data[30];
    } sqldatatype_name;
};
/**/
struct sqlda
{ char  sqldaid[8];
  long  sqldabc;
  short sqln;
  short sqld;
  struct sqlvar sqlvar[1];
};
/**/
/*****
/* Macros for using the sqlvar2 fields. */
/*****
/**/
/*****
/* '2' in the 7th byte of sqldaid indicates a doubled number of */
/*   sqlvar entries. */
/* '3' in the 7th byte of sqldaid indicates a tripled number of */
/*   sqlvar entries. */
/*****
#define SQLDOUBLED '2'
#define SQLTRIPLED '3'
#define SQLSINGLED ' '
/**/

```

```

/*****
/* GETSQLDOUBLED(daptr) returns 1 if the SQLDA pointed to by      */
/* daptr has been doubled, or 0 if it has not been doubled.      */
/*****
#define GETSQLDOUBLED(daptr) \
    (((daptr)->sqlda[6] == ( char) SQLDOUBLED) ? \
     (1)      : \
     (0)      )
/**/
/*****
/* GETSQLTRIPLED(daptr) returns 1 if the SQLDA pointed to by      */
/* daptr has been tripled, or 0 if it has not been tripled.      */
/*****
#define GETSQLTRIPLED(daptr) \
    (((daptr)->sqlda[6] == ( char) SQLTRIPLED) ? \
     (1)      : \
     (0)      )
/**/
/*****
/* SETSQLDOUBLED(daptr, SQLDOUBLED) sets the 7th byte of sqlda   */
/* to '2'.                                                         */
/* SETSQLDOUBLED(daptr, SQLSINGLED) sets the 7th byte of sqlda   */
/* to be a ' '.                                                    */
/*****
#define SETSQLDOUBLED(daptr, newvalue) \
    (((daptr)->sqlda[6] = (newvalue)))
/**/
/*****
/* SETSQLTRIPLED(daptr) sets the 7th byte of sqlda               */
/* to '3'.                                                         */
/*****
#define SETSQLTRIPLED(daptr) \
    (((daptr)->sqlda[6] = (SQLTRIPLED)))
/**/
/*****
/* GETSQLDALONGLEN(daptr,n) returns the data length of the nth   */
/* entry in the sqlda pointed to by daptr. Use this only if the   */
/* sqlda was doubled or tripled and the nth SQLVAR entry has a   */
/* LOB datatype.                                                  */
/*****
#define GETSQLDALONGLEN(daptr,n) ( \
    (long) (((struct sqlvar2 *) &((daptr);->sqlvar[(n) + \
    ((daptr)->sqlda])) \
    ->len.sqllonglen))
/**/

```

```

/*****
/* SETSQLDALONGLEN(daptr,n,len) sets the sqllonglen field of the */
/* sqllda pointed to by daptr to len for the nth entry. Use this only */
/* if the sqllda was doubled or tripled and the nth SQLVAR entry has */
/* a LOB datatype. */
/*****
#define SETSQLDALONGLEN(daptr,n,length) { \
    struct sqlvar2 *var2ptr; \
    var2ptr = (struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + ((daptr)->sqld)]); \
    var2ptr->len.sqllonglen = (long) (length); \
}
/**/
/*****
/* GETSQLDALENPTR(daptr,n) returns a pointer to the data length for */
/* the nth entry in the sqllda pointed to by daptr. Unlike the inline */
/* value (union sql8bytelen len), which is 8 bytes, the sqldataalen */
/* 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 sqllda has been doubled or tripled. */
/*****
#define GETSQLDALENPTR(daptr,n) ( \
    ((struct sqlvar2 *) &((daptr);->sqlvar[(n) + (daptr)->sqld]) \
        ->sqldataalen == NULL) ? \
    ((long *) NULL) : \
    ((long *) ((struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + (daptr)->sqld]) \
        ->sqldataalen) ) )
/**/
/*****
/* SETSQLDALENPTR(daptr,n,ptr) sets a pointer to the data length for */
/* the nth entry in the sqllda pointed to by daptr. */
/* Use this only if the sqllda has been doubled or tripled. */
/*****
#define SETSQLDALENPTR(daptr,n,ptr) { \
    struct sqlvar2 *var2ptr; \
    var2ptr = (struct sqlvar2 *) \
        &((daptr);->sqlvar[(n) + ((daptr)->sqld)]); \
    var2ptr->sqldataalen = (char *) ptr; \
}
/**/
#define SQLDASIZE(n) \
    ( sizeof(struct sqllda) + ((n)-1) * sizeof(struct sqlvar) )
#endif /* SQLDASIZE */

```

COBOL (IBM COBOL and VS COBOL II 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.
```

```
#
# Table 66 on page 908 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
```

SQLDA

column of the table are values that the application must put in the SQLDA before
the statement executes.

Table 66 (Page 1 of 2). Fields of a REXX SQLDA

# Variable name	Usage in DESCRIBE and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
# <i>stem</i> .SQLD	The number of columns that are described in the SQLDA. Double that number if USING BOTH appears in the DESCRIBE or PREPARE INTO statement. Contains a 0 if the statement string is not a query. For DESCRIBE PROCEDURE, the number of result sets returned by the stored procedure. Contains a 0 if no result sets are returned.	The number of host variables that are used by the SQL statement.
# Each SQLDA contains <i>stem</i> .SQLD of the following variables. Therefore, $1 \leq n \leq \textit{stem}.\textit{SQLD}$. 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 SQLDAs for other languages.		
# <i>stem.n</i> .SQLTYPE	Indicates the data type of the column or parameter and whether it can contain null values. For a description of the type codes, see "SQLTYPE and SQLLEN" on page 899. 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 899.
# <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 899 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 899 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.
# <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> .SQLLOCATOR	For DESCRIBE PROCEDURE, the value of a result set locator.	Not used.

Table 66 (Page 2 of 2). Fields of a REXX SQLDA

# Variable name	Usage in DESCRIBE and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
# <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.

Appendix D. DB2 catalog tables

DB2 for OS/390 maintains a set of tables (in database DSNDB06) called the DB2 catalog. This appendix describes 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.

Use as a programming interface

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 column's explanation.

Table spaces and indexes

The table below shows to what table spaces the catalog tables are assigned, and what indexes they have. The pages that follow describe the columns in each table arranged alphabetically by table name. The indexes are in ascending order, except where noted.

DB2 Catalog Tables

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	Page	INDEX SYSIBM. ...	INDEX FIELDS
SYSCOPY	SYSCOPY	941	DSNUCH01	DBNAME.TSNAME.START_RBA.1 TIMESTAMP1
			DSNUCX01	DSNAME
SYSDBASE	SYSCOLAUTH	930		
	SYSCOLUMNS	934	DSNDCX01	TBCREATOR.TBNAME.NAME
			DSNDCX02	TYPESCHEMA.TYPENAME
	SYSFIELDS	953		
	SYSFOREIGNKEYS	954		
	SYSINDEXES	955	DSNDXX01	CREATOR.NAME
			DSNDXX02	DBNAME.INDEXSPACE
			DSNDXX03	TBCREATOR.TBNAME.CREATOR. NAME
	SYSINDEXPART	958	DSNDRX01	IXCREATOR.IXNAME.PARTITION
			DSNDRX02	STORNAME
SYSKEYS	961	DSNDKX01	IXCREATOR.IXNAME.COLNAME	
SYSRELS	985	DSNDLX01	REFTBCREATOR.REFTBNAME	
SYSSYNONYMS	1003	DSNDYX01	CREATOR.NAME	
SYSTABAUTH	1004	DSNATX01	GRANTOR	
		DSNATX02	GRANTEE.TCREATOR.TTNAME. GRANTEETYPE.UPDATECOLS. ALTERAUTH.DELETEAUTH. INDEXAUTH.INSERTAUTH. SELECTAUTH.UPDATEAUTH. CAPTUREAUTH.REFERENCESAUTH. REFCOLS.TRIGGERAUTH	
		DSNATX03	GRANTEE.GRANTEETYPE.COLLID CONTOKEN	
SYSTABLEPART	1007	DSNDPX01	DBNAME.TSNAME.PARTITION	
		DSNDPX02	STORNAME	
SYSTABLES	1010	DSNDTX01	CREATOR.NAME	
		DSNDTX02	DBID.OBID.CREATOR.NAME	
SYSTABLESPACE	1014	DSNDSX01	DBNAME.NAME	
SYSDBAUT	SYSDATABASE	944	DSNDDH01	NAME
			DSNDDX02	GROUP_MEMBER
SYSDBAUTH	947	DSNADH01	GRANTEE.NAME	
		DSNADX01	GRANTOR.NAME	
SYSDDF	IPNAMES	920	DSNFPX01	LINKNAME
	LOCATIONS	921	DSNFCX01	LOCATION
	LULIST	922	DSNFLX01	LINKNAME.LUNAME
			DSNFLX02	LUNAME
	LUMODES	923	DSNFMX01	LUNAME.MODENAME
	LUNAMES	924	DSNFNX01	LUNAME
	MODESELECT	926	DSNFDX01	LUNAME.AUTHID1.PLANNAME1

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	Page	INDEX SYSIBM. ...	INDEX FIELDS
	USERNAMES	1025	DSNFEX01	TYPE.AUTHID ¹ .LINKNAME ¹
SYSGPAUT	SYSRESAUTH	986	DSNAGH01	GRANTEE.QUALIFIER. NAME.OBTYPE
			DSNAGX01	GRANTOR.QUALIFIER. NAME.OBTYPE
SYSGROUP	SYSSTOGROUP	1000	DSNSSH01	NAME
	SYSVOLUMES	1024		
SYSOBJ	SYSAUXRELS	927	DSNOXX01	TBOWNER.TBNAME
			DSNOXX02	AUXTBOWNER.AUXTBNAME
	SYSCONSTDEP	940	DSNCCX01	BSHEMA.BNAME.BTYPE
			DSNCCX02	DTBCREATOR.DTBNAME
	SYSDATATYPES	946	DSNODX01	SCHEMA.NAME
			DSNODX02	DATATYPEID ¹
	SYSPARMS	973	DSNOPX01	SCHEMA.SPECIFICNAME. ROUTINETYPE.ROWTYPE ORDINAL
			DSNOPX02	TYPESHEMA.TYPENAME. ROUTINETYPE.CAST_FUNCTION. OWNER.SCHEMA.SPECIFICNAME
			DSNOPX03	TYPESHEMA.TYPENAME
	SYSROUTINEAUTH	988	DSNOAX01	GRANTOR.SCHEMA. SPECIFICNAME.ROUTINETYPE. GRANTEETYPE.EXECUTEAUTH
			DSNOAX02	GRANTEE.SCHEMA.SPECIFICNAME. ROUTINETYPE.GRANTEETYPE. EXECUTEAUTH.GRANTEDTS
			DSNOAX03	SCHEMA.SPECIFICNAME ROUTINETYPE
SYSROUTINES	989	DSNOFX01	NAME.PARM_COUNT. PARM_SIGNATURE.ROUTINETYPE. SCHEMA.PARM1.PARM2.PARM3. PARM4.PARM5.PARM6.PARM7. PARM8.PARM9.PARM10.PARM11. PARM12.PARM13.PARM14.PARM15. PARM16.PARM17.PARM18.PARM19. PARM20.PARM21.PARM22.PARM23. PARM24.PARM25.PARM26.PARM27. PARM28.PARM29.PARM30	
		DSNOFX02	SCHEMA.SPECIFICNAME. ROUTINETYPE	
		DSNOFX03	NAME.SCHEMA.CAST_FUNCTION. PARM_COUNT.PARM_SIGNATURE. PARM1	
		DSNOFX04	ROUTINE_ID ¹	
		DSNOFX05	SOURCESHEMA.SOURCESPECIFIC. ROUTINETYPE	
		DSNOFX06	SCHEMA.NAME.ROUTINETYPE. PARM_COUNT	
SYSSCHEMAAUTH	995	DSNSKX01	GRANTEE.SCHEMANAME	
		DSNSKX02	GRANTOR	

DB2 Catalog Tables

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	Page	INDEX SYSIBM. ...	INDEX FIELDS
	SYSTRIGGERS	1018	DSNOTX01	SCHEMA.NAME.SEQNO
			DSNOTX02	TBOWNER.TBNAME
SYSPKAGE	SYSPACKAGE	963	DSNKX01	LOCATION.COLLID.NAME. VERSION
			DSNKX02	LOCATION.COLLID.NAME. CONTOKEN
	SYSPACKAUTH	968	DSNKAX01	GRANTOR.LOCATION.COLLID.NAME
			DSNKAX02	GRANTEE.LOCATION.COLLID. NAME.BINDAUTH.COPYAUTH. EXECUTEAUTH
			DSNKAX03	LOCATION.COLLID.NAME
	SYSPACKDEP	969	DSNKDX01	DLOCATION.DCOLLID.DNAME. DCONTOKEN
			DSNKDX02	BQUALIFIER.BNAME.BTYPE
			DSNKDX03	BQUALIFIER.BNAME.BTYPE. DTYPE
	SYSPROCEDURES	982	DSNKCX01	PROCEDURE.AUTHID ¹ .LUNAME ¹
	SYSPACKLIST	970	DSNKLX01	LOCATION.COLLID.NAME
			DSNKLX02	PLANNAME.SEQNO.LOCATION. COLLID.NAME
	SYSPACKSTMT	971	DSNKSX01	LOCATION.COLLID.NAME. CONTOKEN.SEQNO
	SYSPKSYSTEM	975	DSNKYX01	LOCATION.COLLID.NAME. CONTOKEN.SYSTEM.ENABLE
	SYSPLSYSTEM	981	DSNKPX01	NAME.SYSTEM.ENABLE
SYSPLAN	SYSDBRM	950		
	SYSPLAN	976	DSNPPH01	NAME
	SYSPLANAUTH	979	DSNAPH01	GRANTEE.NAME.EXECUTEAUTH
			DSNAPX01	GRANTOR
	SYSPLANDEP	980	DSNGGX01	BCREATOR.BNAME.BTYPE
	SYSSTMT	998		
#	SYSSEQ	996	DSNSQX01	SCHEMA.NAME
#			DSNSQX02	SEQUENCEID ¹
#	SYSSEQ2	997	DSNSRX01	DCREATOR.DNAME.DCOLNAME
	SYSSTATS	931	DSNTNX01	TBOWNER.TBNAME.NAME
	SYSCOLDISTSTATS	932	DSNTPX01	TBOWNER.TBNAME.NAME PARTITION
	SYSCOLSTATS	933	DSNTCX01	TBOWNER.TBNAME.NAME PARTITION
	SYSINDEXSTATS	960	DSNTXX01	OWNER.NAME.PARTITION
	SYSLOBSTATS	962	DSNLNX01	DBNAME.NAME
	SYSTABSTATS	1017	DSNTTX01	OWNER.NAME.PARTITION
SYSSTR	SYSSTRINGS	1001	DSNSSX01	OUTCCSID.INCCSID.IBMREQD

TABLE SPACE DSNDB06. ...	TABLE SYSIBM. ...	Page	INDEX SYSIBM. ...	INDEX FIELDS
	SYSCHECKS	929	DSNSCX01	TBOWNER.TBNAME.CHECKNAME
	SYSCHECKDEP	928	DSNSDX01	TBOWNER.TBNAME.CHECKNAME COLNAME
SYSUSER	SYSUSERAUTH	1019	DSNAUH01	GRANTEE GRANTEDTS
			DSNAUX02	GRANTOR
SYSVIEWS	SYSVIEWDEP	1022	DSNGGX02	BCREATOR.BNAME.BTYPE
			DSNGGX03	BSHEMA.BNAME.BTYPE
	SYSVIEWS	1023	DSNVVX01	CREATOR.NAME.SQNO

Note:

1. Index field is in descending order

SQL statements allowed on the catalog

The following SQL statements can be used to change the value of certain options for existing catalog indexes and table spaces, and 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 FREEPAGE GBPCACHE PCTFREE PIECESIZE COPY
		The GBPCACHE attribute of some indexes cannot be altered. For details, see the description of the GBPCACHE clause in "ALTER INDEX" on page 364.
ALTER TABLE		The only clause allowed is DATA CAPTURE CHANGES.
ALTER TABLESPACE		Only these clauses are allowed: CLOSE FREEPAGE GBPCACHE LOCKMAX MAXROWS PCTFREE TRACKMOD
		The GBPCACHE and MAXROWS attribute of some catalog table spaces cannot be altered. For details, see the description of these clauses in "ALTER TABLESPACE" on page 412.

DB2 Catalog Tables

SQL statement	Index	Allowable clauses and usage notes
CREATE INDEX	User-created	All clauses are allowed, except for: CLOSE YES CLUSTER UNIQUE DEFER YES (only on tables SYSINDEXES, SYSINDEXPART, and SYSKEYS) The only value allowed for BUFFERPOOL is BP0. You can create up to 100 indexes on the catalog.
ALTER INDEX	User-created	All clauses are allowed, except for BUFFERPOOL.
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.SYSOBJ
- DSNDB06.SYSPKAGE
- 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

Descriptions of the following catalog tables have been added:

- SYSIBM.SYSAUXRELS
- SYSIBM.SYSCONSTDEP
- SYSIBM.SYSDATATYPES
- SYSIBM.SYSLOBSTATS
- SYSIBM.SYSPARMS
- SYSIBM.SYSROUTINEAUTH
- SYSIBM.SYSROUTINES
- SYSIBM.SYSSCHEMAAUTH
- SYSIBM.SYSSEQUENCES
- SYSIBM.SYSSEQUENCESDEP
- SYSIBM.SYSTRIGGERS

#

The following tables have new or revised columns, column values, or column descriptions to support the new function in DB2 Version 6:

Table name	New column	Revised column
SYSCOLDIST		COLVALUE FREQUENCYF
SYSCOLDISTSTATS		COLVALUE FREQUENCYF
SYSCOLSTATS		HIGHKEY HIGH2KEY LOWKEY LOW2KEY STATSTIME In addition, all columns can be updated
SYSCOLUMNS	COLSTATUS LENGTH2 DATATYPEID SOURCETYPEID TYPESCHEMA TYPENAME CREATEDTS	COLTYPE LENGTH HIGH2KEY LOW2KEY UPDATES DEFAULT STATSTIME DEFAULTVALUE COLCARDF
SYSCOPY	OTYPE LOWDSNUM HIGHDSNUM	TSNAME DSNUM ICTYPE DSNAME STYPE
SYSDATABASE	INDEXBP	BPOOL DBID ROSHARE TIMESTAMP TYPE ENCODING_SCHEME SBCS_CCSID DBCS_CCSID MIXED_CCSID
SYSINDEXES	COPY COPYLRN CLUSTERRATIOF	UNIQUERULE CLUSTERED DSETPASS IBMREQD CLUSTERRATIO INDEXTYPE

#

DB2 Catalog Tables

#

Table name	New column	Revised column
SYSINDEXPART	SECQTYI IPREFIX ALTEREDTS	SQTY SPACE GBPCACHE FAROFFPOSF NEAROFFPOSF CARDF
SYSINDEXSTATS	FIRSTKEYCARDF FULLKEYCARDF KEYCOUNTF CLUSTERRATIOF	All columns can be updated
SYSPACKAGE	PATHSCHEMAS TYPE DBPROTOCOL FUNCTIONTS OPTHINT	COLLID HOSTLANG IBMREQD VERSION DYNAMICRULES VALID
SYSPACKDEP	DOWNER DTYPE	BTYPE
SYSPACKSTMT	ACCESSPATH STMTNOI SECTNOI	STMTNO SECTNO
SYSPLAN	PATHSCHEMAS DBPROTOCOL FUNCTIONTS OPTHINT	IBMREQD VALID
SYSPLANDEP		BTYPE
SYSRESAUTH		QUALIFIER NAME OBTYP IBMREQD
SYSSTMT	ACCESSPATH STMTNOI SECTNOI	STMTNO SECTNO
SYSSTOGROUP		VPASSWORD
SYSTABAUTH	TRIGGERAUTH	IBMREQD
SYSTABLEPART	TRACKMOD EPOCH SECQTYI CARDF IPREFIX ALTEREDTS	SQTY CARD FARINDREF NEARINDREF PERCACTIVE PERCDROP SPACE GBPCACHE
SYSTABLES	TABLESTATUS	TYPE NPAGES PCTPAGES IBMREQD RECLENGTH STATUS PCTROWCOMP CARDF RBAI RBA2

#

Table name	New column	Revised column
SYSTABLESPACE	LOG NACTIVEF DSSIZE	LOCKRULE DSETPASS IBMREQD LOCKMAX TYPE ENCODING_SCHEME SBCS_CCSID DBCS_CCSID MIXED_CCSID
SYSTABSTATS	CARDF	All columns can be inserted, updated, and deleted
SYSVIEWDEP	BSHEMA	BNAME BCREATOR BTYPE
SYSVIEWS	PATHSCHEMAS	IBMREQD

#

|

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	CHAR(8) 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>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>P The option is "password." Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table.</p> <p> The USERNAMES column must specify "O."</p>	G
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>blank No translation occurs.</p>	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	<p>Whether the row came from the basic machine-readable material (MRM) tape:</p> <p>N No Y Yes</p>	G
IPADDR	VARCHAR(254) NOT NULL WITH DEFAULT	<p>This column contains the IP 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"> • If the IPADDR contains a left justified character string containing four numeric values delimited by decimal points, DB2 assumes the value is an IP address in dotted decimal format. For example, '123.456.78.91' would be interpreted as a dotted decimal IP address. • All other values are interpreted as a TCP/IP domain name, which can be resolved by the TCP/IP gethostbyname socket call. TCP/IP domain names are not case sensitive. 	G

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	CHAR(16) 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	CHAR(8) 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'	Whether the row came from the basic machine-readable material (MRM) tape: <p>N No Y Yes</p>	G
PORT	CHAR(32) 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
TPN	VARCHAR(64) 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. <p>For an SQL/DS™ server, TPN should contain the resource ID of the SQL/DS machine.</p>	G

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	CHAR(8) 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	CHAR(8) 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'	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.LUMODES table

Each row of the 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	CHAR(8) NOT NULL	LU name of the server involved in the CNOS processing.	G
MODENAME	CHAR(8) 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'	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.LUNAMES table

The 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	CHAR(8) 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	CHAR(8) NOT NULL WITH DEFAULT	Mode used to establish inter-system conversations. A blank indicates the default mode IBMDB2LM (DB2 private protocol access).	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 DCE 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 DCE security ticket. If the USERNAMES column contains 'I' or 'B', RACF is not invoked to validate incoming connection requests that contain only a userid.	G
SECURITY_OUT	CHAR(1) NOT NULL WITH DEFAULT 'A'	This column defines the security option that is used when local DB2 SQL applications connect to any remote server associated with this LUNAME: A The option is "already verified." Outbound connection requests contain an authorization ID and no password. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column. R The option is "RACF PassTicket." Outbound connection requests contain a userid and a RACF PassTicket. The server's LU name is used as the RACF PassTicket application name. The authorization ID used for an outbound request is either the DB2 user's authorization ID or a translated ID, depending upon the value of the USERNAMES column. P The option is "password." Outbound connection requests contain an authorization ID and a password. The password is obtained from the SYSIBM.USERNAMES table or RACF, depending upon the value specified in the ENCRYPTPWDS column. The USERNAMES column must specify 'B' or 'O'.	G

Column name	Data type	Description	Use
ENCRYPTPSWDS	CHAR(1) NOT NULL WITH DEFAULT 'N'	<p>This column only applies to DB2 for OS/390 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, we recommend using 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
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'	<p>Whether the row came from the basic machine-readable material (MRM) tape:</p> <p>N No</p> <p>Y Yes</p>	G

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	CHAR(8) 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	CHAR(8) 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	CHAR(8) NOT NULL	LU name associated with the SQL request.	G
MODENAME	CHAR(8) 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'	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

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	CHAR(8) NOT NULL	Authorization ID of the owner of the base table.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the base table.	G
COLNAME	VARCHAR(18) NOT NULL	Name of the LOB column in the base table.	G
PARTITION	SMALLINT NOT NULL	Partition number if the base table space is partitioned. Otherwise, the value is 0.	G
AUXTBOWNER	CHAR(8) NOT NULL	Authorization ID of the owner of the auxiliary table.	G
AUXTBNAME	VARCHAR(18) 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSCHECKDEP table

Contains one row for each reference to a column in a table check constraint.

Column name	Data type	Description	Use
TBOWNER	CHAR(8) NOT NULL	Authorization ID of the owner of the table on which the table check constraint is defined.	G
TBNAME	VARCHAR(18) 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(18) NOT NULL	Name of the column that the table check constraint refers to.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSCHECKS table

Contains one row for each table check constraint.

Column name	Data type	Description	Use
TBOWNER	CHAR(8) NOT NULL	Authorization ID of the owner of the table on which the table check constraint is defined.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the creator of the table check constraint.	G
DBID	SMALLINT NOT NULL	Internal identifier of the database for the table check constraint.	S
OBID	SMALLINT NOT NULL	Internal identifier of the table check constraint.	S
TIMESTAMP	TIMESTAMP NOT NULL	Time when the table check constraint was created.	G
RBA	CHAR(6) FOR BIT DATA NOT NULL	The log RBA when the table check constraint was created.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table on which the check constraint is defined.	G
CHECKNAME	VARCHAR(128) NOT NULL	Table check constraint name.	G
CHECKCONDITION	VARCHAR(3800) NOT NULL	Text of the table check constraint.	G

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	CHAR(8) NOT NULL	Authorization ID of the user who granted the privileges. Could also be PUBLIC or PUBLIC followed by an asterisk ⁴⁴ .	G
GRANTEE	CHAR(8) NOT NULL	Authorization ID 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 P An application plan or a package. The grantee is a package if COLLID is not blank.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the table or view on which the update privilege is held.	G
TNAME	VARCHAR(18) NOT NULL	Name of the table or view.	G
	CHAR(12) NOT NULL	Internal use only	I
DATEGRANTED	CHAR(6) NOT NULL	Date the privilege was granted, in the form <i>yymmdd</i> .	G
TIMEGRANTED	CHAR(8) NOT NULL	Time the privilege was granted, in the form <i>hhmmssst</i> .	G
COLNAME	VARCHAR(18) NOT NULL	Name of the column to which the UPDATE privilege applies.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
	CHAR(16) NOT NULL WITH DEFAULT	Not used	N
COLLID	CHAR(18) NOT NULL WITH DEFAULT	If GRANTEE is a package, its collection name. Otherwise, the value is blank.	G
CONTOKEN	CHAR(8) NOT NULL WITH DEFAULT	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

⁴⁴ PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see Section 3 (Volume 1) of *DB2 Administration Guide*.

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
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
TBOWNER	CHAR(8) NOT NULL	Authorization ID of the owner of the table that contains the column.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(18) 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(254) NOT NULL FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data might not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: C Cardinality F Frequent value	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values for the column group. This number is valid only for TYPE C statistics.	S
COLGROUPCOLNO	VARCHAR(254) NOT NULL WITH DEFAULT	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS. This is an updatable column.	S
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of 1 indicates 100%. A value of .153 indicates 15.3%. Statistics are not collected for an index on a ROWID column.	G

SYSIBM.SYSCOLDISTSTATS table

Contains zero or more rows per partition for the first key column of a partitioning index. Rows are inserted when RUNSTATS scans index partitions of the partitioning index. No row is inserted if the index is a nonpartitioning index. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
	SMALLINT NOT NULL	Not used	N
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	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	CHAR(8) NOT NULL	Authorization ID of the owner of the table that contains the column.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(18) 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(254) NOT NULL FOR BIT DATA	Contains the data of a frequently occurring value. Statistics are not collected for an index on a ROWID column. If the value has a non-character data type, the data may not be printable.	S
TYPE	CHAR(1) NOT NULL WITH DEFAULT 'F'	The type of statistics gathered: C Cardinality F Frequent value	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values for the column group. This number is valid only for TYPE C statistics.	S
COLGROUPCOLNO	VARCHAR(254) NOT NULL WITH DEFAULT	Identifies the set of columns associated with the statistics. If the statistics are only associated with a single column, the field contains a zero length. Otherwise, the field is an array of SMALLINT column numbers with a dimension equal to the value in NUMCOLUMNS. This is an updatable column.	S
NUMCOLUMNS	SMALLINT NOT NULL WITH DEFAULT 1	Identifies the number of columns associated with the statistics.	G
FREQUENCYF	FLOAT NOT NULL WITH DEFAULT -1	Gives the percentage of rows in the table with the value specified in COLVALUE when the number is multiplied by 100. For example, a value of 1 indicates 100%. A value of .153 indicates 15.3%. Statistics are not collected for an index on a ROWID column.	G

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	CHAR(8) NOT NULL FOR BIT DATA	Highest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column. If the column has a non-character data type, the data might not be printable.	S
HIGH2KEY	CHAR(8) NOT NULL FOR BIT DATA	Second highest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column. If the column has a non-character data type, the data might not be printable.	S
LOWKEY	CHAR(8) NOT NULL FOR BIT DATA	Lowest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column. If the column has a non-character data type, the data might not be printable.	S
LOW2KEY	CHAR(8) NOT NULL FOR BIT DATA	Second lowest value of the column within the partition. Blank if statistics have not been gathered or the column is an indicator column. If the column has a non-character data type, the data might not be printable.	S
	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	CHAR(8) NOT NULL	Authorization ID of the owner of the table that contains the column.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table that contains the column.	G
NAME	VARCHAR(18) NOT NULL	Name of the column.	G
COLCARDATA	VARCHAR(1000) NOT NULL FOR BIT DATA	Internal use only	I

SYSIBM.SYSCOLUMNS table

Contains one row for every column of each table and view.

Column name	Data type	Description	Use																																				
NAME	VARCHAR(18) NOT NULL	Name of the column.	G																																				
TBNAME	VARCHAR(18) NOT NULL	Name of the table or view which contains the column.	G																																				
TBCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner 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). An additional row with column number 0 is inserted into SYSCOLUMNS if the definition of the table is incomplete (all required unique indexes have not been created).	G																																				
COLTYPE	CHAR(8) NOT NULL	<p>The type of the column specified in the definition of the column:</p> <table border="0"> <tr><td>INTEGER</td><td>Large integer</td></tr> <tr><td>SMALLINT</td><td>Small integer</td></tr> <tr><td>FLOAT</td><td>Floating-point</td></tr> <tr><td>CHAR</td><td>Fixed-length character string</td></tr> <tr><td>VARCHAR</td><td>Varying-length character string</td></tr> <tr><td>LONGVAR</td><td>Varying-length character string</td></tr> <tr><td>DECIMAL</td><td>Decimal</td></tr> <tr><td>GRAPHIC</td><td>Fixed-length graphic string</td></tr> <tr><td>VARG</td><td>Varying-length graphic string</td></tr> <tr><td>LONGVARG</td><td>Varying-length graphic string</td></tr> <tr><td>DATE</td><td>Date</td></tr> <tr><td>TIME</td><td>Time</td></tr> <tr><td>TIMESTAMP</td><td>Timestamp</td></tr> <tr><td>BLOB</td><td>Binary large object</td></tr> <tr><td>CLOB</td><td>Character large object</td></tr> <tr><td>DBCLOB</td><td>Double-byte character large object</td></tr> <tr><td>ROWID</td><td>Row ID data type</td></tr> <tr><td>DISTINCT</td><td>Distinct type</td></tr> </table> <p>Whether a column described as VARCHAR, LONGVAR, VARG, or LONGVARG is a long string column depends on its length attribute. A column described as BLOB, CLOB, or DBCLOB is always a long string column.</p>	INTEGER	Large integer	SMALLINT	Small integer	FLOAT	Floating-point	CHAR	Fixed-length character string	VARCHAR	Varying-length character string	LONGVAR	Varying-length character string	DECIMAL	Decimal	GRAPHIC	Fixed-length graphic string	VARG	Varying-length graphic string	LONGVARG	Varying-length graphic string	DATE	Date	TIME	Time	TIMESTAMP	Timestamp	BLOB	Binary large object	CLOB	Character large object	DBCLOB	Double-byte character large object	ROWID	Row ID data type	DISTINCT	Distinct type	G
INTEGER	Large integer																																						
SMALLINT	Small integer																																						
FLOAT	Floating-point																																						
CHAR	Fixed-length character string																																						
VARCHAR	Varying-length character string																																						
LONGVAR	Varying-length character string																																						
DECIMAL	Decimal																																						
GRAPHIC	Fixed-length graphic string																																						
VARG	Varying-length graphic string																																						
LONGVARG	Varying-length graphic string																																						
DATE	Date																																						
TIME	Time																																						
TIMESTAMP	Timestamp																																						
BLOB	Binary large object																																						
CLOB	Character large object																																						
DBCLOB	Double-byte character large object																																						
ROWID	Row ID data type																																						
DISTINCT	Distinct type																																						

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.	G
		INTEGER 4	
		SMALLINT 2	
		FLOAT 4 or 8	
		CHAR Length of string	
		VARCHAR Maximum length of string	
		LONGVAR Maximum length of string	
		DECIMAL Precision of number	
		GRAPHIC Number of DBCS characters	
		VARG Maximum number of DBCS characters	
		LONGVARG Maximum number of DBCS characters	
		DATE 4	
		TIME 3	
		TIMESTAMP 10	
BLOB 4 - The length of the field that is stored in the base table. The maximum length of the LOB column is found in LENGTH2.			
CLOB 4 - The length of the field that is stored in the base table. The maximum length of the CLOB column is found in LENGTH2.			
DBCLOB 4 - The length of the field that is stored in the base table. The maximum length of the DBCLOB column is found in LENGTH2.			
ROWID 17 - The maximum length of the stored portion of the identifier.			
DISTINCT The length of the source data type.			
SCALE	SMALLINT NOT NULL	Scale of decimal data. Zero if not a decimal column.	G
NULLS	CHAR(1) NOT NULL	Whether the column can contain null values: N No Y Yes The value can be N for a view column that is derived from an expression or a function. Nevertheless, such a column allows nulls when an outer select list refers to it.	G
		INTEGER NOT NULL	
HIGH2KEY	CHAR(8) NOT NULL FOR BIT DATA	Second highest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable. This is an updatable column.	S
LOW2KEY	CHAR(8) NOT NULL FOR BIT DATA	Second lowest value of the column. Blank if statistics have not been gathered, or the column is an indicator column or a column of an auxiliary table. If the column has a non-character data type, the data might not be printable. 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:	G
		<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) 	
		The value can be Y for columns of a read-only view or columns defined with the AS IDENTITY and GENERATED ALWAYS attributes..	
#			
#			
#			

SYSIBM.SYSCOLUMNS

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
REMARKS	VARCHAR(254) NOT NULL	A character string provided by the user with the COMMENT ON statement.	G

Column name	Data type	Description	Use																												
DEFAULT	CHAR(1) NOT NULL	The contents of this column are meaningful only if the TYPE column for the associated SYSTABLES row indicates that this is for a table (T) or a created temporary table (G). Default indicator: A The column has a row ID data type (COLTYPE='ROWID') and the GENERATED ALWAYS attribute. B The column has a default value that depends on the data type of the column. <table border="1"> <thead> <tr> <th>Data type</th> <th>Default Value</th> </tr> </thead> <tbody> <tr> <td>Numeric</td> <td>0</td> </tr> <tr> <td>Fixed-length string</td> <td>Blanks</td> </tr> <tr> <td>Varying-length string</td> <td>A string length of 0</td> </tr> <tr> <td>Date</td> <td>The current date</td> </tr> <tr> <td>Time</td> <td>The current time</td> </tr> <tr> <td>Timestamp</td> <td>The current timestamp</td> </tr> </tbody> </table> D The column has a row ID data type (COLTYPE='ROWID') and the GENERATED BY DEFAULT attribute. I The column is defined with the AS IDENTITY and GENERATED ALWAYS attributes. J The column is defined with the AS IDENTITY and GENERATED BY DEFAULT attributes. N The column has no default value. 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 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. <table border="1"> <thead> <tr> <th>Data type</th> <th>Default Value</th> </tr> </thead> <tbody> <tr> <td>Numeric</td> <td>0</td> </tr> <tr> <td>Fixed-length string</td> <td>Blanks</td> </tr> <tr> <td>Varying-length string</td> <td>A string length of 0</td> </tr> <tr> <td>Date</td> <td>The current date</td> </tr> <tr> <td>Time</td> <td>The current time</td> </tr> <tr> <td>Timestamp</td> <td>The current timestamp</td> </tr> </tbody> </table> 1 The column has a default value that is the string constant 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 hex string found in the DEFAULTVALUE column of this table row.	Data type	Default Value	Numeric	0	Fixed-length string	Blanks	Varying-length string	A string length of 0	Date	The current date	Time	The current time	Timestamp	The current timestamp	Data type	Default Value	Numeric	0	Fixed-length string	Blanks	Varying-length string	A string length of 0	Date	The current date	Time	The current time	Timestamp	The current timestamp	G
Data type	Default Value																														
Numeric	0																														
Fixed-length string	Blanks																														
Varying-length string	A string length of 0																														
Date	The current date																														
Time	The current time																														
Timestamp	The current timestamp																														
Data type	Default Value																														
Numeric	0																														
Fixed-length string	Blanks																														
Varying-length string	A string length of 0																														
Date	The current date																														
Time	The current time																														
Timestamp	The current timestamp																														

#

SYSIBM.SYSCOLUMNS

Column name	Data type	Description	Use
KEYSEQ	SMALLINT NOT NULL	The column's numeric position within the table's primary key. The value is 0 if it is not part of a primary key.	G
FOREIGNKEY	CHAR(1) NOT NULL	Applies to character columns only, where it indicates the subtype of the data. A value of B indicates BIT data, and if value of the field MIXED DATA on installation panel DSNTIPF is: <ul style="list-style-type: none"> • NO, any other value indicates SBCS data • YES, an S indicates SBCS and any other value indicates MIXED. This is an updatable column.	G
FLDPROC	CHAR(1) NOT NULL	Whether the column has a field procedure: N No Y Yes blank The column is for views.	G
LABEL	VARCHAR(30) NOT NULL	The column label provided by the user with a LABEL ON statement; otherwise it is an empty string.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. If the value is '0001-01-02.00.00.00.000000', which indicates that an ALTER TABLE statement was executed to change the length of a VARCHAR column, RUNSTATS should be run to update the statistics before they are used. This is an updatable column.	G
DEFAULTVALUE	VARCHAR(512) NOT NULL WITH DEFAULT	This field is meaningful only if the column being described is for a table (the TYPE column of the associated SYSTABLES row is T for table or G for created temporary table). When the DEFAULT column is 1, 2, 3, 4, or 5, this field contains the default value of the column. If the default value is a string constant or a hexadecimal constant (DEFAULT is 1 or 5, respectively), the value is stored without delimiters, except for a graphic string constant which is enclosed by the shift-out and shift-in characters. If the default value is a numeric constant (DEFAULT is 2, 3, or 4), the value is stored as specified by the user, including sign and decimal point representation, as appropriate for the constant. When the default column is S or U and the default value was specified with the definition of a new column on an ALTER TABLE statement, this field contains the value of the CURRENT SQLID or USER special register at the time of the ALTER statement.	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	Indicates the status of the definition of a column: I The definition is incomplete because a LOB table space, auxiliary table, or index on an auxiliary table has not been created for the column. blank The definition is complete.	G
LENGTH2	INTEGER NOT NULL WITH DEFAULT	Maximum length of the data retrieved from the column. Possible values are: 0 Not a LOB or ROWID column 40 For a ROWID column, the length of the returned value 1 to 2 147 483 647 bytes For a LOB column, the maximum length	G

Column name	Data type	Description	Use
DATATYPEID	INTEGER NOT NULL WITH DEFAULT	For a built-in data type, the internal ID of the built-in type. For a distinct type, the internal ID of the distinct type.	S
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 sourced.	S
TYPESHEMA	CHAR(8) NOT NULL WITH DEFAULT 'SYSIBM'	If COLTYPE is 'DISTINCT', the schema of the distinct type. Otherwise, the value is 'SYSIBM'.	G
TYPENAME	VARCHAR(18) NOT NULL WITH DEFAULT	If COLTYPE is 'DISTINCT', the name of the distinct type. Otherwise, the value is the same as the value of the COLTYPE column. TYPENAME is set only for columns created in Version 6 or later. The value for columns created earlier is not filled in.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Timestamp when the column was created. The value is '0001-01-01.00.00.00.000000' if the column was created prior to migration to Version 6.	G

SYSIBM.SYSCONSTDEP table

Records dependencies on check constraints or user-defined defaults for a column.

Column name	Data type	Description	Use
BNAME	VARCHAR(18) NOT NULL	Name of the object on which the dependency exists.	G
BSHEMA	CHAR(8) 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(18) NOT NULL	Name of the table to which the dependency applies.	G
DTBCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

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 D CHECK DATA LOG(NO) (no log records for the range are available for RECOVER utility) F COPY FULL YES I COPY FULL NO P RECOVER TOCOPY or RECOVER TORBA (partial recovery point) Q QUIESCE R LOAD REPLACE LOG(YES) S LOAD REPLACE LOG(NO) W REORG LOG(NO) X REORG LOG(YES) Y LOAD LOG(NO) Z LOAD LOG(YES) T TERM UTILITY command (terminated utility)	G
ICDATE	CHAR(6) NOT NULL	Date of the entry in the form <i>yyymmdd</i> .	G
START_RBA	CHAR(6) NOT NULL 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
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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSCOPY

Column name	Data type	Description	Use
DSNAME	CHAR(44) NOT NULL	For ICTYPE='P' (RECOVER TOCOPY only), 'I', or 'F', DSNAME contains the data set name. Otherwise, DSNAME contains the name of the database and table space or index space in the form, <i>database-name.space-name</i> , or DSNAME is blank for any row migrated from a DB2 release prior to Version 4.	G
ICTIME	CHAR(6) NOT NULL	The time at which this row was inserted, in the form <i>hhmmss</i> . The insertion takes place after the completion of the operation that the row represents. ICTIME is blank for any row which was migrated from Version 1 Release 1 of DB2.	G
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 may not be supported in later DB2 releases.	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 'I' or 'F'.	G

Column name	Data type	Description	Use
# STYPE #	CHAR(1) NOT NULL WITH DEFAULT	<p>When ICTYPE=A (the length of an indexed VARCHAR column in a table was increased), the value is V.</p> <p>When ICTYPE=F, the values are:</p> <p>blank DB2 image copy C DFSMS concurrent copy R LOAD REPLACE(YES) S LOAD REPLACE(NO) W REORG LOG(NO) X REORG LOG(YES)</p> <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=P and the operation is RECOVER TORBA LOGONLY, the value is L.</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, S, W, or X and the operation is resetting REORG pending status, the value is A.</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 I COPY FULL NO</p> <p>For other values of ICTYPE, the value is blank.</p>	G
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>	G
GROUP_MEMBER	CHAR(8) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that performed the operation. This column is blank if the DB2 subsystem was not in a DB2 data sharing environment at the time the operation was performed.	G
OTYPE	CHAR(1) NOT NULL WITH DEFAULT 'T'	<p>Type of object that the recovery information is for:</p> <p>I Index space T Table space</p>	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.) The column is valid only for these uses.	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.	G

SYSIBM.SYSDATABASE table

Contains one row for each database, except for database DSNDB01.

Column name	Data type	Description	Use
NAME	CHAR(8) NOT NULL	Database name.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the database.	G
STGROUP	CHAR(8) 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes E V2R3 dependency indicator; not from MRM tape G V4 dependency indicator; not from MRM tape	G
CREATEDBY	CHAR(8) 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	TIMESTAMP NOT NULL WITH DEFAULT	The value is '0001-01-01-00.00.00.000000'.	G
TYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of database: blank Not a work file database or a TEMP database. T A TEMP database. The database was created with the AS TEMP clause, which indicates it is used for declared temporary tables. 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	CHAR(8) NOT NULL WITH DEFAULT	The DB2 data sharing member name of the DB2 subsystem that uses this work file database. This column is blank if the work file database was not created in a DB2 data sharing environment, or if the database is not a work file database as indicated by the TYPE column.	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the database. For DSNDB04 and DSNDB06, the value is '1985-04-01.00.00.00.000000'.	G
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 blank For DSNDB04, a work file database, and a TEMP database.	G

Column name	Data type	Description	Use
# SBCS_CCSID #	INTEGER NOT NULL WITH DEFAULT	Default SBCS CCSID for the database. For 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 database. For 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 database. For a TEMP database or 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

SYSIBM.SYSDATATYPES table

Contains one row for each distinct type defined to the system.

Column name	Data type	Description	Use
SCHEMA	CHAR(8) NOT NULL	Schema of the distinct type.	G
OWNER	CHAR(8) NOT NULL	Owner of the distinct type.	G
NAME	CHAR(18) NOT NULL	Name of the distinct type.	G
CREATEDBY	CHAR(8) NOT NULL	Authorization ID under which the distinct type was created.	G
SOURCESCHEMA	CHAR(8) NOT NULL	Schema of the source data type.	G
SOURCETYPE	CHAR(18) 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 upon which the distinct type is sourced.	S
LENGTH	INTEGER NOT NULL	Maximum length or precision of a distinct type that is sourced on the IBM-defined DECIMAL data type.	G
SCALE	SMALLINT NOT NULL	Scale for a distinct type that is sourced 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	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
REMARKS	VARCHAR(254) NOT NULL	A character string provided by the user with the COMMENT ON statement.	G

SYSIBM.SYSDBAUTH table

Records the privileges that are held by users over databases.

Column name	Data type	Description	Use	
GRANTOR	CHAR(8) NOT NULL	Authorization ID of the user who granted the privileges. Could also be PUBLIC or PUBLIC followed by an asterisk. ⁴⁵	G	
GRANTEE	CHAR(8) NOT NULL	Application ID of the user who holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G	
NAME	CHAR(8) NOT NULL	Database name.	G	
	CHAR(12) NOT NULL	Internal use only	I	
DATEGRANTED	CHAR(6) NOT NULL	Date the privileges were granted; in the form <i>yyymmdd</i> .	G	
TIMEGRANTED	CHAR(8) NOT NULL	Time the privileges were granted; in the form <i>hhmmssst</i> .	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.	G	
		blank		Not applicable
		C		DBCTL
		D		DBADM
		L		SYSCTRL
		M		DBMAINT
CREATETABAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create tables within the database:	G	
		blank		Privilege is not held
		G		Privilege held with the GRANT option
		Y		Privilege is held without the GRANT option
CREATETSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create table spaces within the database:	G	
		blank		Privilege is not held
		G		Privilege held with the GRANT option
		Y		Privilege is held without the GRANT option
DBADMAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBADM authority over the database:	G	
		blank		Privilege is not held
		G		Privilege held with the GRANT option
		Y		Privilege is held without the GRANT option
DBCTRLAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBCTRL authority over the database:	G	
		blank		Privilege is not held
		G		Privilege held with the GRANT option
		Y		Privilege is held without the GRANT option
DBMAINTAUTH	CHAR(1) NOT NULL	Whether the GRANTEE has DBMAINT authority over the database:	G	
		blank		Privilege is not held
		G		Privilege held with the GRANT option
		Y		Privilege is held without the GRANT option

⁴⁵ PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see Section 3 (Volume 1) of *DB2 Administration Guide*.

SYSIBM.SYSDBAUTH

Column name	Data type	Description	Use
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
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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

Column name	Data type	Description	Use
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G

SYSIBM.SYSDBRM table

Contains one row for each DBRM of each application plan.

Column name	Data type	Description	Use
NAME	CHAR(8) NOT NULL	Name of the DBRM.	G
TIMESTAMP	CHAR(8) NOT NULL FOR BIT DATA	Consistency token.	S
PDSNAME	CHAR(44) NOT NULL	Name of the partitioned data set of which the DBRM is a member.	G
PLNAME	CHAR(8) NOT NULL	Name of the application plan of which this DBRM is a part.	G
PLCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the application plan.	G
PRECOMPTIME	CHAR(8) NOT NULL	Time of precompilation in the form <i>hhmmssst</i> . If the LEVEL precompiler option is used, then this value does not represent the precompile time.	G
PRECOMPDATE	CHAR(6) NOT NULL	Date of precompilation in the form <i>yyymmdd</i> . If the LEVEL precompiler option is used, then this value does not represent the precompile date.	G
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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	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
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 67): N No Y Yes	G

Column name	Data type	Description	Use
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 134): blank No Y Yes	G
VERSION	VARCHAR(64) NOT NULL WITH DEFAULT	Version identifier for the DBRM.	G
PRECOMPTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the DBRM was precompiled.	G

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.

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSFIELDS table

Contains one row for every column that has a field procedure.

Column name	Data type	Description	Use
TBCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the table that contains the column.	G
TBNAME	VARCHAR(18) 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(18) NOT NULL	Name of the column.	G
FLDTYPE	CHAR(8) NOT NULL	Data type of the encoded values in the field ⁴⁶ : 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 ⁴⁶ . 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
FLDPROC	CHAR(8) NOT NULL	For a row describing a field procedure, the name of the procedure ⁴⁶ .	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 ⁴⁶ .	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
EXITPARML	SMALLINT NOT NULL	For a row describing a field procedure, the length of the field procedure parameter value block ⁴⁶ .	G
PARMLIST	VARCHAR(254) 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 ⁴⁶ .	G
EXITPARAM	VARCHAR(1530) NOT NULL FOR BIT DATA	For a row describing a field procedure, the parameter value block of the field procedure (the control block passed to the field procedure when it is invoked) ⁴⁶ .	G

⁴⁶ Some columns might contain statistical values from a prior release.

SYSIBM.SYSFOREIGNKEYS table

Contains one row for every column of every foreign key.

Column name	Data type	Description	Use
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the table that contains the column.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table that contains the column.	G
RELNAME	CHAR(8) NOT NULL	Constraint name for the constraint for which the column is part of the foreign key.	G
COLNAME	VARCHAR(18) 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSINDEXES table

Contains one row for every index.

Column name	Data type	Description	Use
NAME	VARCHAR(18) NOT NULL	Name of the index.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the index.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table on which the index is defined.	G
TBCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner 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.	G
COLCOUNT	SMALLINT NOT NULL	The number of columns in the key.	G
CLUSTERING	CHAR(1) NOT NULL	Whether CLUSTER was specified when the index was created: 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
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	CHAR(8) NOT NULL	Name of the database that contains the index.	G
INDEXSPACE	CHAR(8) 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

SYSIBM.SYSINDEXES

Column name	Data type	Description	Use
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	Size, in bytes, of the leaf pages in the index: 256, 512, 1024, 2048, or 4096	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
		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
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes C V2R1 dependency indicator; not from MRM tape D V2R2 dependency indicator; not from MRM tape E V2R3 dependency indicator; not from MRM tape G V4 dependency indicator; not from MRM tape I V6 dependency indicator; not from MRM tape	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	CHAR(8) 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.	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

Column name	Data type	Description	Use
FULLKEYCARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of distinct values of the key. The value is -1 if statistics have not been gathered. This is an updatable column.	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 '0001-01-01.00.00.000000'.	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 '0001-01-01.00.00.000000'.	G
PIECESIZE	INTEGER NOT NULL WITH DEFAULT	Maximum size of a data set in kilobytes for nonpartitioning indexes. A value of zero (0) indicates that the index is a partitioning index or that the index was created in a DB2 release prior to Version 5.	G
COPY	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether COPY YES was specified for the index, which indicates if the index can be copied and if SYSIBM.SYSLGRNX recording is enabled for the index. N No Y Yes	G
COPYLRSN	CHAR(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 COPYLRSN is not changed when the index is altered to COPY NO.	G
CLUSTERRATIOF	FLOAT NOT NULL WITH DEFAULT	When multiplied by 100, the value of the column is the percentage of rows that are in clustering order. For example, a value of .9125 indicates 91.25%. For a partitioning index, it is the weighted average of all index partitions in terms of the number of rows in the partition. The value is 0 if statistics have not been gathered. The value is -2 if the index is for an auxiliary table. This is an updatable column.	G

SYSIBM.SYSINDEXPART table

Contains one row for each nonpartitioning index and one row for each partition of a partitioning index.

Column name	Data type	Description	Use
PARTITION	SMALLINT NOT NULL	Partition number; Zero if index is not partitioned.	G
IXNAME	VARCHAR(18) NOT NULL	Name of the index.	G
IXCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the index.	G
PQTY	INTEGER NOT NULL	Primary space allocation in units of 4KB storage blocks. For user-managed data sets, 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.	G
SQTY	SMALLINT NOT NULL	Secondary space allocation in units of 4KB storage blocks. For user-managed data sets, 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. If the value does not fit into the column, the value of the column is 0. See the description of column SECQTYI.	G
STORTYPE	CHAR(1) NOT NULL	Type of storage allocation: E Explicit, and STORNAME names an integrated catalog facility catalog I Implicit, and STORNAME names a storage group	G
STORNAME	CHAR(8) NOT NULL	Name of storage group or integrated catalog facility catalog used for space allocation.	G
VCATNAME	CHAR(8) 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.	S
	INTEGER NOT NULL	Not used	S
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
LIMITKEY	VARCHAR(512) NOT NULL FOR BIT DATA	The high value of the limit key of the partition in an internal format. Zero if the index is not partitioned. If any column of the key has a field procedure, the internal format is the encoded form of the value.	S
FREEPAGE	SMALLINT NOT NULL	Number of pages that are loaded before a page is left as free space.	G
PCTFREE	SMALLINT NOT NULL	Percentage of each leaf or nonleaf page that is left as free space.	G

#

Column name	Data type	Description	Use
SPACE	INTEGER NOT NULL WITH DEFAULT	Number of kilobytes of DASD storage allocated to the index space partition, as determined by the last execution of the STOSPACE utility. The value is 0 if STOSPACE or RUNSTATS has not been run. 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). The value is -1 if 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.	G
#			
#			
#			
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
	CHAR(1) NOT NULL	Not used	N
GBPCACHE	CHAR(1) NOT NULL WITH DEFAULT	Group buffer pool cache option specified for this index or index partition. blank Only changed pages are cached in the group buffer pool. A Changed and unchanged pages are cached in the group buffer pool. N No data is cached in the group buffer pool.	G
FAROFFPOSF	FLOAT NOT NULL WITH DEFAULT -1	Number of referred to rows far from optimal position because of an insert into a full page. The value is -1 if statistics have not been gathered. The 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. Not applicable for an index on an auxiliary table.	S
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Number of keys 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 if RUNSTATS INDEX with UPDATE(SPACE) or UPDATE(ALL) is executed; otherwise, the value is zero.	G
IPREFIX	CHAR(1) NOT NULL WITH DEFAULT 'I'	Reserved.	S
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

SYSIBM.SYSINDEXSTATS table

Contains one row for each partition of a partitioning 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 partition 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
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
PARTITION	SMALLINT NOT NULL	Partition number of the index.	G
OWNER	CHAR(8) NOT NULL	Authorization ID of the owner of the index.	G
NAME	VARCHAR(18) 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
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

SYSIBM.SYSKEYS table

Contains one row for each column of an index key.

Column name	Data type	Description	Use
IXNAME	VARCHAR(18) NOT NULL	Name of the index.	G
IXCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the index.	G
COLNAME	VARCHAR(18) 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 example, 4 (out of 4).	G
ORDERING	CHAR(1) NOT NULL	Order of the column in the key: A Ascending D Descending	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

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	Ratio of organization in the LOB table space. A value of 1 indicates perfect organization of the LOB table space. The greater the value exceeds 1, the more disorganized the LOB table space.	S
DBNAME	CHAR(8) NOT NULL	Name of the database that contains the LOB table space named in NAME.	G
NAME	CHAR(8) NOT NULL	Name of the LOB table space.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSPACKAGE table

Contains a row for every package.

Column name	Data type	Description	Use
LOCATION	CHAR(16) NOT NULL	Always contains blanks	S
COLLID	CHAR(18) NOT NULL	Name of the package collection. For a trigger package, it is the schema name of the trigger.	G
NAME	CHAR(8) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL	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	CHAR(8) 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	CHAR(8) 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	CHAR(8) 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 ⁴⁷ of the package, in bytes.	G
AVGSIZE	INTEGER NOT NULL	Average size, in bytes, of those sections ⁴⁷ 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
VALID	CHAR(1) NOT NULL	Whether the package is valid: <p>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.</p> <p>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 V5R1, the change invalidates the package.</p> <p>N No</p> <p>Y Yes</p>	G

⁴⁷ 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.

SYSIBM.SYSPACKAGE

Column name	Data type	Description	Use
OPERATIVE	CHAR(1) NOT NULL	Whether the package can be allocated: N No; an explicit BIND or REBIND is required before the package can be allocated. Y Yes	G
VALIDATE	CHAR(1) NOT NULL	Whether validity checking can be deferred until run time: B All checking must be performed at bind time. R Validation is done at run time for tables, views, and privileges that do not exist at bind time.	G
ISOLATION	CHAR(1) NOT NULL	Isolation level when the package was last bound or rebound R RR (repeatable read) S CS (cursor stability) T RS (read stability) U UR (uncommitted read) blank Not specified, and therefore at the level specified for the plan executing the package	G
RELEASE	CHAR(1) NOT NULL	The value used for RELEASE when the package was last bound or rebound: C Value used was COMMIT. D Value used was DEALLOCATE. blank Not specified, and therefore the value specified for the plan executing the package.	G
EXPLAIN	CHAR(1) NOT NULL	EXPLAIN option specified for the package; that is, whether information on the package's statements was added to the owner of the PLAN_TABLE table: N No Y Yes	G
QUOTE	CHAR(1) NOT NULL	SQL string delimiter for SQL statements in the package: N Apostrophe Y Quotation mark	G
COMMA	CHAR(1) NOT NULL	Decimal point representation for SQL statements in package: N Period Y Comma	G
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 67): N No Y Yes	G

Column name	Data type	Description	Use
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 134): 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
PCTIMESTAMP	TIMESTAMP NOT NULL	Date and time the application program was precompiled, or 0001-01-01-00.00.000000 if the LEVEL precompiler option was used, or if the package came from a non-DB2 location.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes B V1R3 dependency indicator; not from MRM tape C V2R1 dependency indicator; not from MRM tape D V2R2 dependency indicator; not from MRM tape E V2R3 dependency indicator; not from MRM tape F V3R1 dependency indicator; not from MRM tape G V4 dependency indicator; not from MRM tape H V5 dependency indicator; not from MRM tape I V6 dependency indicator for an auxiliary table; not from MRM tape	G
VERSION	VARCHAR(64) NOT NULL	Version identifier for the package. The value is blank for a trigger package (TYPE='T').	G
PDSNAME	VARCHAR(44) 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: • 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

SYSIBM.SYSPACKAGE

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 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	<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 61.</p>	G
REOPTVAR	CHAR(1) NOT NULL WITH DEFAULT 'N'	<p>Whether the access path is determined again at execution time using input variable values:</p> <p>N Bind option NOREOPT(VARS) indicates that the access path is determined at bind time.</p> <p>Y Bind option REOPT(VARS) indicates that the access path is determined at execution time for SQL statements with variable values.</p>	G

Column name	Data type	Description	Use
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(254) 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 was created: blank BIND PACKAGE command created the package. T CREATE TRIGGER statement created the package, and the package is a trigger 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
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	CHAR(8) NOT NULL WITH DEFAULT	Value of the OPTHINT bind option. Identifies rows in the authid.PLAN_TABLE to be used as input to the optimizer. Contains blanks if no rows in the authid.PLAN_TABLE are to be used as input.	G

SYSIBM.SYSPACKAUTH table

Records the privileges that are held by users over packages.

Column name	Data type	Description	Use
GRANTOR	CHAR(8) NOT NULL	Authorization ID of the user who granted the privilege. Could also be PUBLIC or PUBLIC followed by an asterisk ⁴⁸ .	G
GRANTEE	CHAR(8) 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	CHAR(16) NOT NULL	Always contains blanks	S
COLLID	CHAR(18) NOT NULL	Collection name for the package or packages on which the privilege was granted.	G
NAME	CHAR(8) 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	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 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.	G
		blank Not applicable A PACKADM (on collection *) C DBCTL D DBADM L SYSCTRL M DBMAINT P PACKADM (on a specific collection) S SYSADM	
BINDAUTH	CHAR(1) NOT NULL	Whether GRANTEE can use the BIND and REBIND subcommands against the package:	G
		blank Privilege is not held	
		G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	
COPYAUTH	CHAR(1) NOT NULL	Whether GRANTEE can COPY the package:	G
		blank Privilege is not held	
		G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	
EXECUTEAUTH	CHAR(1) NOT NULL	Whether GRANTEE can execute the package:	G
		blank Privilege is not held	
		G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape:	G
		N No Y Yes	

⁴⁸ PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see Section 3 (Volume 1) of *DB2 Administration Guide*.

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(18) NOT NULL	The name of an object that a package depends on.	G
BQUALIFIER	CHAR(8) NOT NULL	If BNAME identifies a table space, the name of its database. Otherwise, the authorization ID of the owner of BNAME.	G
BTYPE	CHAR(1) NOT NULL	Type of object identified by BNAME and BQUALIFIER: A Alias F User-defined function or cast function I Index O Stored procedure P Partitioned table space R Table space S Synonym T Table V View	G
DLOCATION	CHAR(16) NOT NULL	Always contains blanks	S
DCOLLID	CHAR(18) NOT NULL	Name of the package collection.	G
DNAME	CHAR(8) NOT NULL	Name of the package.	G
DCONTOKEN	CHAR(8) NOT NULL	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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
DOWNER	CHAR(8) NOT NULL WITH DEFAULT	Owner of the package.	G
DTYPE	CHAR(1) NOT NULL WITH DEFAULT	Type of package: T Trigger package blank Not a trigger package	G

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	CHAR(8) NOT NULL	Name of the application plan.	G
SEQNO	SMALLINT NOT NULL	Sequence number of the entry in the package list.	G
LOCATION	CHAR(16) NOT NULL	Location of the package. Blank if this is local. An asterisk (*) indicates location to be determined at run time.	G
COLLID	CHAR(18) NOT NULL	Collection name for the package. An asterisk (*) indicates that the collection name is determined at run time.	G
NAME	CHAR(8) 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSPACKSTMT table

Contains one or more rows for each statement in a package.

Column name	Data type	Description	Use
LOCATION	CHAR(16) NOT NULL	Always contains blanks	S
COLLID	CHAR(18) NOT NULL	Name of the package collection.	G
NAME	CHAR(8) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL	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	SMALLINT NOT NULL	Sequence number of the row with respect to a statement in the package ⁴⁹ . 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 displayed as zero or a or a negative number (see STMTNOI for the statement number). ⁵⁰	G
SECTNO	SMALLINT NOT NULL	The section number of the statement. ⁵⁰	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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
VERSION	VARCHAR(64) NOT NULL	Version identifier for the package.	G
STMT	VARCHAR(254) NOT NULL	All or a portion of the text for the SQL statement that the row represents.	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 KEEP UPDATE LOCKS for an RS isolation X KEEP UPDATE LOCKS for an RR isolation blank The WITH clause was not specified on this statement. The isolation level is recorded in SYSPACKAGE.ISOLATION and in SYSPLAN.ISOLATION.	G

⁴⁹ Rows in which the value of SEQNO, STMTNO, and SECTNO are zero are for internal use.

⁵⁰ 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).

SYSIBM.SYSPACKSTMT

Column name	Data type	Description	Use
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Status of binding the statement:	S
		A	Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using defaults for input variables during access path selection.
		B	Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using values for input variables during access path selection.
		C	Compiled - statement was bound successfully using defaults for input variables during access path selection.
		E	Explain - statement is an SQL EXPLAIN statement. The explain is done at bind time using defaults for input variables during access path selection.
		F	Parsed - statement did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using values for input variables during access path selection.
		G	Compiled - statement bound successfully, but REOPT is specified. The statement will be rebound at execution time using values for input variables during access path selection.
		H	Parsed - statement is either a data definition statement or a statement that did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using defaults for input variables during access path selection. Data manipulation statements use defaults for input variables during access path selection.
		I	Indefinite - statement is dynamic. The statement will be bound at execution time using defaults for input variables during access path selection.
		J	Indefinite - statement is dynamic. The statement will be bound at execution time using values for input variables during access path selection.
		K	Control - CALL statement.
		L	Bad - the statement has some allowable error. The bind continues but the statement cannot be executed.
		blank	The statement is non-executable, or was bound in a DB2 release prior to Version 5.
ACCESSPATH	CHAR(1) NOT NULL WITH DEFAULT	For static statements, indicates if the access path for the statement is based on user-specified optimization hints. A value of 'H' indicates that optimization hints were used. A blank value indicates that the access path was determined without the use of optimization hints, or that there is no access path associated with the statement. For dynamic statements, the value is blank.	G
STMTNOI	INTEGER NOT NULL WITH DEFAULT	The statement number of the statement in the source program.	G
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement.	G

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	CHAR(8) NOT NULL	Schema of the routine.	G
OWNER	CHAR(8) NOT NULL	Owner of the routine.	G
NAME	CHAR(18) NOT NULL	Name of the routine.	G
SPECIFICNAME	CHAR(18) 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
#			
#			
#			
PARAMNAME	CHAR(18) NOT NULL	Name of the parameter.	G
ROUTINEID	INTEGER NOT NULL	Internal identifier of the routine.	S
ROWTYPE	CHAR(1) NOT NULL	Type of parameter: P Input parameter. O Output parameter; not applicable for functions B Both an input and an output parameter; not applicable for functions R Result before casting; not applicable for stored procedures C Result after casting; not applicable for stored procedures This column can have an additional value. If the routine is a user-defined function that is sourced on a built-in function, an additional row is inserted into the catalog table for each input parameter (ROWTYPE='P'). The additional row describes the data type of the corresponding input parameter of the built-in function. When such a row is added, the value of ROWTYPE is 'S'.	G
ORDINAL	SMALLINT NOT NULL	If ROWTYPE is B, O, P, or S, the ordinal number of the parameter within the routine signature. If ROWTYPE is C or R, the value is 0.	G
TYPESCHEMA	CHAR(8) NOT NULL	Schema of the data type of the parameter.	G
TYPENAME	CHAR(18) NOT NULL	Name of the data type of the parameter.	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 sourced.	S

SYSIBM.SYSPARMS

Column name	Data type	Description	Use
LOCATOR	CHAR(1) NOT NULL	Indicates whether a locator to a value, instead of the actual value, is to be passed as the input value 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 in the case of a decimal parameter, its precision.	G
SCALE	SMALLINT NOT NULL	Scale of the data type of the parameter.	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.	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 blank The source type is not a character, graphic, or datetime type.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

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	CHAR(16) NOT NULL	Always contains blanks	S
COLLID	CHAR(18) NOT NULL	Name of the package collection.	G
NAME	CHAR(8) NOT NULL	Name of the package.	G
CONTOKEN	CHAR(8) NOT NULL	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	CHAR(8) NOT NULL	Environment. Values can be: <p>BATCH TSO batch</p> <p>CICS Customer Information Control System</p> <p>DB2CALL DB2 call attachment facility</p> <p>DLIBATCH DLI batch support facility</p> <p>IMSBMP IMS BMP region</p> <p>IMSMPP IMS MPP and IFP region</p> <p>REMOTE remote application server</p>	G
ENABLE	CHAR(1) NOT NULL	Indicates whether the connections represented by the row are enabled or disabled: <p>N Disabled</p> <p>Y Enabled</p>	G
CNAME	CHAR(20) 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 an application 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. <p>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.</p>	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: <p>N No</p> <p>Y Yes</p>	G

SYSIBM.SYSPLAN table

Contains one row for each application plan.

Column name	Data type	Description	Use
NAME	CHAR(8) NOT NULL	Name of the application plan.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the application plan.	G
BINDDATE	CHAR(6) NOT NULL	Date on which the plan was last bound, in the form <i>yymmdd</i> .	G
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 The 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". The changes do not invalidate the plan. H The ALTER TABLE statement changed the description of the table or base table of a view that is referred to by the application plan. For releases of DB2 prior to Version 5, the change invalidates 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
BINDTIME	CHAR(8) NOT NULL	Time of the BIND in the form <i>hhmmssst</i> .	G
PLSIZE	INTEGER NOT NULL	Size of the base section ⁵¹ of the plan, in bytes.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes B V1R3 dependency indicator; not from MRM tape C V2R1 dependency indicator; not from MRM tape D V2R2 dependency indicator; not from MRM tape E V2R3 dependency indicator; not from MRM tape F V3R1 dependency indicator; not from MRM tape G V4 dependency indicator; not from MRM tape H V5 dependency indicator; not from MRM tape I V6 dependency indicator for an auxiliary table; not from MRM tape	G

⁵¹ Plans are divided into *sections*. The base section of the plan must be in the EDM pool during the entire time the application program is executing. Other sections of the plan, corresponding roughly to sets of related SQL statements, are brought into the pool as needed.

Column name	Data type	Description	Use
AVGSIZE	INTEGER NOT NULL	Average size, in bytes, of those sections ⁵¹ of the plan that contain SQL statements processed at bind time.	G
ACQUIRE	CHAR(1) NOT NULL	When resources are acquired: A At allocation U At first use	G
RELEASE	CHAR(1) NOT NULL	When resources are released: C At commit D At deallocation	G
	CHAR(1) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
	CHAR(1) NOT NULL	Not used	N
EXPLAN	CHAR(1) NOT NULL	EXPLAIN option specified for the plan; that is, whether information on the plan's statements was added to the owner's PLAN_TABLE table: N No Y Yes	G
EXPREDICATE	CHAR(1) NOT NULL	Indicates the CURRENTDATA option when the plan was bound or rebound: B Data currency is not required for ambiguous cursors. Allow blocking for ambiguous cursors. C Data currency is required for ambiguous cursors. Inhibit blocking for ambiguous cursors. N Blocking is inhibited for ambiguous cursors, but the plan was created before the CURRENTDATA option was available.	G
BOUNDBY	CHAR(8) NOT NULL WITH DEFAULT	Primary authorization ID of the binder of the plan.	G
QUALIFIER	CHAR(8) 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.SYPACKLIST 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
CURRENTSERVER	CHAR(16) 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

SYSIBM.SYSPLAN

Column name	Data type	Description	Use
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) (EXPLICIT) A DISCONNECT(AUTOMATIC) (AUTOMATIC) C DISCONNECT(CONDITIONAL) (CONDITIONAL) blank A migrated plan	G
GROUP_MEMBER	CHAR(8) 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
REOPTVAR	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether the access path is determined again at execution time using input variable values: N Bind option NOREOPT(VARS) indicates that the access path is determined at bind time. Y Bind option REOPT(VARS) indicates that the access path is determined at execution time for SQL statements with variable values.	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(254) 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	CHAR(8) NOT NULL WITH DEFAULT	Value of the OPTHINT bind option. Identifies rows in the authid.PLAN_TABLE to be used as input to the optimizer. Contains blanks if no rows in the authid.PLAN_TABLE are to be used as input.	G

SYSIBM.SYSPLANAUTH table

Records the privileges that are held by users over application plans.

Column name	Data type	Description	Use
GRANTOR	CHAR(8) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	CHAR(8) NOT NULL	Authorization ID of the user who holds the privileges. Could also be PUBLIC for a grant to PUBLIC.	G
NAME	CHAR(8) NOT NULL	Name of the application plan on which the privileges are held.	G
	CHAR(12) NOT NULL	Internal use only	I
DATEGRANTED	CHAR(6) NOT NULL	Date the privileges were granted; in the form <i>yyymmdd</i> .	G
TIMEGRANTED	CHAR(8) NOT NULL	Time the privileges were granted; in the form <i>hhmmssst</i> .	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 DBCTL 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G

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(18) NOT NULL	The name of an object the plan depends on.	G
BCREATOR	CHAR(8) NOT NULL	If BNAME is a table space, its database. Otherwise, the authorization ID of the owner of BNAME.	G
BTYPE	CHAR(1) NOT NULL	Type of object identified by BNAME: A Alias F User-defined function or cast function I Index O Stored procedure P Partitioned table space R Table space S Synonym T Table V View	G
DNAME	CHAR(8) NOT NULL	Name of the plan.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

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	CHAR(8) NOT NULL	Name of the plan.	G
SYSTEM	CHAR(8) 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	CHAR(8) 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 CNAME can also be blank when SYSTEM is not equal to BATCH or DB2CALL. When this is so, the row applies to all connections for the indicated environment.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.SYSPROCEDURES table

In releases of DB2 for OS/390 prior to Version 6, users were required to use the SYSPROCEDURES catalog table to define stored procedures to DB2. In Version 6, the SYSROUTINES catalog table contains information about stored procedures. When Version 6 was installed, the rows in SYSPROCEDURES that had non-blank values for AUTHID and LUNAME were copied, with appropriate formatting, to SYSROUTINES.

Although Version 6 of DB2 for OS/390 does not use SYSPROCEDURES, SYSPROCEDURES is available for fallback to Version 5. For information about falling back and remigrating, see *DB2 Installation Guide*. However, any procedures that are defined with this version will not be available for fallback to Version 5. Likewise, any procedure definitions that are altered for this version with the ALTER PROCEDURE statement will not be changed in SYSPROCEDURES and thus will not be available in Version 5.

Column name	Data type	Description	Use
PROCEDURE	CHAR(18) NOT NULL	Name of the stored procedure specified on the SQL CALL statement.	G
AUTHID	CHAR(8) NOT NULL WITH DEFAULT	SQL authorization ID of the user running the SQL application that issued the SQL CALL statement. When the SQL CALL statement is received from a remote location, this column is compared to the value of the authorization ID after outbound and inbound name translation operations have been performed. If AUTHID is blank, values in this row apply to all authorization IDs.	G
LUNAME	CHAR(8) NOT NULL WITH DEFAULT	LUNAME of the system that issued the SQL CALL statement. <ul style="list-style-type: none"> If the LUNAME column contains the local DB2 system's LUNAME, this row applies to local applications that issue the SQL CALL statement. If the LUNAME column contains the LUNAME of a remote client, this row applies to SQL CALL statements received from that remote client. If LUNAME is blank, the values in this row apply to all systems, including the local DB2 system and clients connected through TCP/IP or SNA. To ease migration to future releases of DB2, specify blanks in this field.	G
LOADMOD	CHAR(8) NOT NULL	Member name of the MVS load module that DB2 should load to satisfy the request for the stored procedure. When the value of LANGUAGE is COMPJAVA, this column value is not used.	G
LINKAGE	CHAR(1) NOT NULL WITH DEFAULT	Linkage convention used to pass parameters to the stored procedure: N The SIMPLE WITH NULLS convention is used where an indicator array is passed to the stored procedure. Null input parameters are allowed. blank The SIMPLE linkage convention is used where input parameters cannot be null. Conventions for passing parameters to stored procedures are described in Section 7 of <i>DB2 Application Programming and SQL Guide</i> .	G

Column name	Data type	Description	Use
COLLID	CHAR(18) NOT NULL	Name of the package collection to use when the stored procedure is executed. A blank value indicates that the package collection is the same as the package collection of the program that issued the SQL CALL statement.	G
LANGUAGE	CHAR(8) NOT NULL	Programming language used to create the stored procedure. Possible values are 'ASSEMBLE', 'PLI', 'COBOL', 'C', 'REXX', or 'COMPJAVA'.	G
ASUTIME	INTEGER NOT NULL WITH DEFAULT	Number of service units permitted for any single invocation of this stored procedure. If ASUTIME is zero, there is no limit on the service units. If a stored procedure uses more service units than allowed by the ASUTIME value, DB2 cancels the stored procedure.	G
STAYRESIDENT	CHAR(1) NOT NULL WITH DEFAULT	Determines whether the stored procedure load module is deleted from memory when the stored procedure ends. Y The load module remains resident in memory after the stored procedure ends. blank The load module is deleted from memory after the stored procedure ends.	G
IBMREQD	CHAR(1) NOT NULL	Indicates whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
RUNOPTS	VARCHAR(254) NOT NULL	The Language Environment (Language Environment for MVS & VM) run-time options to use for this stored procedure. If this column contains an empty string, the installation default Language Environment run-time options are used. When the value of LANGUAGE is COMPJAVA, this column value is the stored procedure program name, in the format <i>class.method</i> . An example Language Environment run-time option list follows: 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'	G
PARMLIST	VARCHAR(3000) NOT NULL	Defines the parameter list expected by the stored procedure. For syntax and a description of the information contained in the PARMLIST string, see Section 7 of <i>DB2 Application Programming and SQL Guide</i> .	G
RESULT_SETS	SMALLINT NOT NULL WITH DEFAULT	Maximum number of query result sets that can be returned by this stored procedure. Zero indicates there are no query result sets.	G
WLM_ENV	CHAR(18) NOT NULL WITH DEFAULT	Name of the WLM environment to be used to run this stored procedure. A blank value results in the stored procedure being run in the DB2-established stored procedures address space.	G
PGM_TYPE	CHAR(1) NOT NULL WITH DEFAULT 'M'	Whether the stored procedure runs as a main routine or a subroutine: M The stored procedure runs as a main routine. S The stored procedure runs as a subroutine.	G

#

SYSIBM.SYSPROCEDURES

Column name	Data type	Description	Use
EXTERNAL_SECURITY	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether a special RACF environment is required to control access to non-SQL resources: N RACF access to non-SQL resources is not required for the stored procedure. This option is sufficient when the stored procedure only accesses SQL objects. Y A RACF environment should be automatically created by DB2 each time the stored procedure is invoked so that RACF can manage access to non-SQL resources.	G
COMMIT_ON_RETURN	CHAR(1) WITH DEFAULT 'N'	Whether the unit of work is always to be committed immediately upon 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. A null value means the same as the value N.	G

SYSIBM.SYSRELS table

Contains one row for every referential constraint.

Column name	Data type	Description	Use
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the dependent table of the referential constraint.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the dependent table of the referential constraint.	G
RELNAME	CHAR(8) NOT NULL	Constraint name.	G
REFTBNAME	VARCHAR(18) NOT NULL	Name of the parent table of the referential constraint.	G
REFTBCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	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	CHAR(8) NOT NULL WITH DEFAULT	Owner 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(18) 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

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	CHAR(8) NOT NULL	Authorization ID of the user who granted the privilege.	G
GRANTEE	CHAR(8) NOT NULL	Authorization ID of the user who holds the privilege. Could also be PUBLIC for a grant to PUBLIC.	G
QUALIFIER	CHAR(8) 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'). Otherwise, the value is blank.	G
NAME	CHAR(18) 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 C DBCTL D DBADM L SYSCTRL M DBMAINT S SYSADM P PACKADM (on a specific collection) A PACKADM (on collection *)	G
OBTYP	CHAR(1) NOT NULL	Type of object:	G
		B Buffer pool C Collection D Distinct type R Table space S Storage group	
	CHAR(12) NOT NULL	Internal use only	I
DATEGRANTED	CHAR(6) NOT NULL	Date the privilege was granted; in the form <i>yymmdd</i> .	G
TIMGRANTED	CHAR(8) NOT NULL	Time the privilege was granted; in the form <i>hhmmssst</i> .	G
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 OBTYP is C (a collection) and QUALIFIER is PACKADM. The authority held is CREATE IN when the OBTYP is C and QUALIFIER is blank.	G

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes I V6 dependency indicator; not from MRM tape. The column is set to 'I' only if the USAGE privilege is granted on a distinct type, or the USE privilege is granted on any buffer pool in the range BP8K0 to BP8K0 or BP16K0 to BP16K9.	G
GRANTEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the GRANT statement was executed.	G

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	CHAR(8) NOT NULL	Authorization ID of the user who granted the privilege.	G
GRANTEE	CHAR(8) 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	CHAR(8) NOT NULL	Schema of the routine	G
SPECIFICNAME	CHAR(18) 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 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	CHAR(18) NOT NULL	If the GRANTEE is a package, its collection name. Otherwise, the value is blank.	G
CONTOKEN	CHAR(8) NOT NULL	If the GRANTEE is a package, the consistency token of the DBRM from which the package was derived. Otherwise, the value is blank.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: I V6 dependency indicator; not from MRM tape	G

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	CHAR(8) NOT NULL	Schema of the routine.	G
OWNER	CHAR(8) NOT NULL	Owner of the routine.	G
NAME	CHAR(18) 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	CHAR(8) NOT NULL	Authorization ID under which the routine was created.	G
SPECIFICNAME	CHAR(18) 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 user-defined function or stored procedure U Sourced on user-defined function or built-in function S System-generated function	G
FUNCTION_TYPE	CHAR(1) NOT NULL	Type of function: C Column 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
LANGUAGE	CHAR(8) NOT NULL	Implementation language of the routine: ASSEMBLE C COBOL COMPJAVA PLI REXX SQL blank ORIGIN is not 'E'.	G
COLLID	CHAR(18) NOT NULL	Name of the package collection to be used when the routine is executed. A blank value indicates the package collection is the same as the package collection of the program that invoked the routine.	G
SOURCESCHEMA	CHAR(8) 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

SYSIBM.SYSROUTINES

Column name	Data type	Description	Use
SOURCESPECIFIC	CHAR(18) 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 may differ with a given set of input values). Y Deterministic (results are consistent). blank ROUTINETYPE='F' and ORIGIN is not 'E' (the routine is a function, but not an external function).	G
# EXTERNAL_ACTION	CHAR(1) NOT NULL	The external action option of an external 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' 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. A cast function is generated by DB2 for a CREATE DISTINCT TYPE statement.	G
# 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

Column name	Data type	Description	Use
PARALLEL	CHAR(1) NOT NULL	The PARALLEL option of an external function:	G
		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').
PARAMETER_STYLE	CHAR(1) NOT NULL	The PARAMETER STYLE option of an external function or stored procedure:	G
		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'.
FENCED	CHAR(1) NOT NULL	Y Indicates that this routine runs separately in the DB2 address space. All user-defined functions run in the DB2 address space.	G
SQL_DATA_ACCESS	CHAR(1) NOT NULL	The SQL statements that are allowed in an external function or stored procedure:	G
		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.		
DBINFO	CHAR(1) NOT NULL	The DBINFO option of an external function or stored procedure:	G
		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.
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.	G
		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'.

SYSIBM.SYSROUTINES

Column name	Data type	Description	Use
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. If a routine consumes more CPU service units than the ASUTIME value allows, DB2 cancels the routine.	G
WLM_ENVIRONMENT	CHAR(18) NOT NULL	Name of the WLM environment to be used to run this routine. If the ROUTINETYPE = 'P', the value might be blank. Blank causes the stored procedure to be run in the DB2 stored procedure address space.	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.	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
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 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 exist or ROUTINETYPE = 'F', the value is zero.	G

Column name	Data type	Description	Use
LOBCOLUMNS	SMALLINT NOT NULL	If ORIGIN = 'E', 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', 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	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
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

SYSIBM.SYSROUTINES

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. The value is -1 if the predicted cardinality is not known.	S
RESULT_COLS	SMALLINT NOT NULL DEFAULT 1	For a table function, the number of columns in the result table. Otherwise, the value is 1.	S
EXTERNAL_NAME	CHAR(254) NOT NULL	The path/module/function that DB2 should load to execute the routine. The column is blank if the ORIGIN is not 'E'.	G
PARM_SIGNATURE	VARCHAR(150) NOT NULL FOR BIT DATA	Internal use only	I
RUNOPTS	VARCHAR(254) 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.	G
REMARKS	VARCHAR(254) NOT NULL	A character string provided by the user with the COMMENT ON statement.	G

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	CHAR(8) NOT NULL	Authorization ID of the user who granted the privileges or SYSADM.	G
GRANTEE	CHAR(8) NOT NULL	Authorization ID of the user or group who holds the privileges. Can also be PUBLIC for a grant to PUBLIC.	G
SCHEMANAME	CHAR(8) 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 SYCTRL 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	Whether the row came from the basic machine-readable material (MRM) tape: I V6 dependency indicator; not from MRM tape	G

SYSIBM.SYSSEQUENCES table

Contains one row for each identity column.

#	Column name	Data type	Description	Use
#	SCHEMA	CHAR(8) NOT NULL	The value of TBCREATOR from the SYSCOLUMNS entry for the identity column.	G
#	OWNER	CHAR(8) NOT NULL	The value of TBCREATOR from the SYSCOLUMNS entry for the identity column.	G
#	NAME	CHAR(18) NOT NULL	Name that DB2 generated for the identity column.	G
#	SEQTYPE	CHAR(1) NOT NULL	Type of entry: I For an identity column	G
#	SEQUENCEID	INTEGER NOT NULL	Internal identifier of the identity column.	G
#	CREATEDBY	CHAR(8) NOT NULL	The authorization ID under which the identity column was created.	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 data type (and precision if the data type is decimal).	G
#	MINVALUE	DECIMAL(31,0) NOT NULL	Minimum value allowed for the data type (and precision if the data type is decimal).	G
#	CYCLE	CHAR(1) NOT NULL	The value is always 'N' for an identity column.	G
#	CACHE	INTEGER NOT NULL	Number of identity column 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	The value is always 'N' for an identity column.	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 sourced.	S
#	CREATEDTS	TIMESTAMP NOT NULL	Timestamp when the identity column was created.	G
#	ALTEREDTS	TIMESTAMP NOT NULL	Timestamp when the identity column was created.	G
#	MAXASSIGNEDVAL	DECIMAL(31,0)	Last possible assigned value. Initialized to null when the sequence 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	J V6 dependency indicator.	G
#	REMARKS	VARCHAR(254) NOT NULL	The value is always blank for an identity column.	G

SYSIBM.SYSSTMT table

Contains one or more rows for each SQL statement of each DBRM.

Column name	Data type	Description	Use
NAME	CHAR(8) NOT NULL	Name of the DBRM.	G
PLNAME	CHAR(8) NOT NULL	Name of the application plan.	G
PLCREATOR	CHAR(8) 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 ⁵² . 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 displayed as zero (see STMTNOI for the statement number). ⁵²	G
SECTNO	SMALLINT NOT NULL	The section number of the statement. ⁵²	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
TEXT	VARCHAR(254) NOT NULL	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 KEEP UPDATE LOCKS for an RS isolation X KEEP UPDATE LOCKS for an RR isolation blank The WITH clause was not specified on this statement. The isolation level is recorded in SYSPACKAGE.ISOLATION and in SYSPLAN.ISOLATION.	G

⁵² Rows in which the values of SEQNO, STMTNO, and SECTNO are zero are for internal use.

Column name	Data type	Description	Use
STATUS	CHAR(1) NOT NULL WITH DEFAULT	Status of binding the statement:	S
		A	Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using defaults for input variables during access path selection.
		B	Distributed - statement uses DB2 private protocol access. The statement will be parsed and executed at the server using values for input variables during access path selection.
		C	Compiled - statement was bound successfully using defaults for input variables during access path selection.
		E	Explain - statement is an SQL EXPLAIN statement. The explain is done at bind time using defaults for input variables during access path selection.
		F	Parsed - statement did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using values for input variables during access path selection.
		G	Compiled - statement bound successfully, but REOPT is specified. The statement will be rebound at execution time using values for input variables during access path selection.
		H	Parsed - statement is either a data definition statement or a statement that did not bind successfully and VALIDATE(RUN) was used. The statement will be rebound at execution time using defaults for input variables during access path selection. Data manipulation statements use defaults for input variables during access path selection.
		I	Indefinite - statement is dynamic. The statement will be bound at execution time using defaults for input variables during access path selection.
		J	Indefinite - statement is dynamic. The statement will be bound at execution time using values for input variables during access path selection.
		K	Control - CALL statement.
		L	Bad - the statement has some allowable error. The bind continues but the statement cannot be executed.
		blank	The statement is non-executable, or was bound in a DB2 release prior to Version 5.
ACCESSPATH	CHAR(1) NOT NULL WITH DEFAULT	For static statements, indicates if the access path for the statement is based on user-specified optimization hints. A value of 'H' indicates that optimization hints were used. A blank value indicates that the access path was determined without the use of optimization hints, or that there is no access path associated with the statement. For dynamic statements, the value is blank.	G
STMTNOI	INTEGER NOT NULL WITH DEFAULT	The statement number of the statement in the source program.	G
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement.	G

SYSIBM.SYSSTOGROUP table

Contains one row for each storage group.

Column name	Data type	Description	Use
NAME	CHAR(8) NOT NULL	Name of the storage group.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the storage group.	G
VCATNAME	CHAR(8) NOT NULL	Name of the integrated catalog facility catalog.	G
		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
SPCDATE	CHAR(5) NOT NULL	Date when the SPACE column was last updated, in the form <i>yyddd</i> .	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
CREATEDBY	CHAR(8) 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

SYSIBM.SYSSTRINGS table

Contains information about character conversion. Each row describes a conversion from one coded character set to another.

If OS/390 Version 2 Release 9 (or a subsequent release) is installed, refer to
OS/390 C/C++ Programming Guide for information on the additional conversions
that are supported.

Column name	Data type	Description	Use
INCCSID	INTEGER NOT NULL	The source CCSID for the character conversion represented by this row.	G
OUTCCSID	INTEGER NOT NULL	The target CCSID for the character conversion represented by this row.	G
TRANSTYPE	CHAR(2) NOT NULL	Indicates the nature of the conversion. Values can be: GG GRAPHIC to GRAPHIC MM EBCDIC MIXED to EBCDIC MIXED MS EBCDIC MIXED to SBCS PM ASCII MIXED to EBCDIC MIXED PS ASCII MIXED to SBCS SM SBCS to EBCDIC MIXED SS SBCS to SBCS MP EBCDIC MIXED to ASCII MIXED PP ASCII MIXED to ASCII MIXED SP SBCS to ASCII MIXED	G
ERRORBYTE	CHAR(1) FOR BIT DATA (Nulls are allowed)	The byte used in the conversion table as an error byte. 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. Null indicates the absence of a substitution character.	S
TRANSPROC	CHAR(8) NOT NULL WITH DEFAULT	The name of a module or blanks. If IBMREQD is 'N', a nonblank value is the name of a conversion procedure provided by the user. If IBMREQD is 'Y', a nonblank value is the name of a DB2 module that contains DBCS conversion tables. 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.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape (see the information following this table): N No Y Yes	G
TRANSTAB	VARCHAR(256) FOR BIT DATA NOT NULL WITH DEFAULT	Either a conversion table or an empty string.	S

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 would then use 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

SYSIBM.SYSSTRINGS

information about the use of inserted rows for character conversion, see Appendix C of *DB2 Installation Guide*.

SYSIBM.SYSSYNONYMS table

Contains one row for each synonym of a table or view.

Column name	Data type	Description	Use
NAME	VARCHAR(18) NOT NULL	Synonym for the table or view.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the synonym.	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table or view.	G
TBCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the table or view.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
CREATEDBY	CHAR(8) 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

SYSIBM.SYSTABAUTH table

Records the privileges that users hold on tables and views.

Column name	Data type	Description	Use
GRANTOR	CHAR(8) NOT NULL	Authorization ID of the user who granted the privileges. Could also be PUBLIC, or PUBLIC followed by an asterisk. ⁵³	G
GRANTEE	CHAR(8) NOT NULL	Authorization ID 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 P An application plan or a package. The grantee is a package if COLLID is not blank.	G
DBNAME	CHAR(8) 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	CHAR(8) NOT NULL	If the row of SYSIBM.SYSTABAUTH was created as a result of a CREATE VIEW statement, SCREATOR is the authorization ID of the owner of a table or view referred to in the CREATE VIEW statement. Otherwise, SCREATOR is the same as TCREATOR.	G
STNAME	VARCHAR(18) NOT NULL	If the row of SYSIBM.SYSTABAUTH was created as a result of a CREATE VIEW statement, STNAME is the name of a table or view referred to in the CREATE VIEW statement. Otherwise, STNAME is the same as TTNAME.	G
TCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the table or view.	G
TTNAME	VARCHAR(18) 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 DBCTL D DBADM L SYSCTRL M DBMAINT S SYSADM	G
	CHAR(12) NOT NULL	Internal use only	I
DATEGRANTED	CHAR(6) NOT NULL	Date the privileges were granted, in the form <i>yymmdd</i> .	G
TIMGRANTED	CHAR(8) NOT NULL	Time the privileges were granted, in the form <i>hhmmssst</i> .	G
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

⁵³ PUBLIC followed by an asterisk (PUBLIC*) denotes PUBLIC AT ALL LOCATIONS. For the conditions where GRANTOR can be PUBLIC or PUBLIC*, see Section 3 (Volume 1) of *DB2 Administration Guide*.

Column name	Data type	Description	Use
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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes I V6 dependency indicator; not from MRM tape. The column is set to 'I' only if any of the following privileges are granted: <ul style="list-style-type: none"> • The TRIGGER privilege on a table • The USAGE privilege on a distinct type • The USE privilege is on any buffer pool in the range BP8K0 to BP8K0 or BP16K0 to BP16K9 	G
	CHAR(16) NOT NULL WITH DEFAULT	Not used	N
	CHAR(16) NOT NULL WITH DEFAULT	Not used	N
COLLID	CHAR(18) 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	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

SYSIBM.SYSTABAUTH

Column name	Data type	Description	Use
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 triggering table: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G

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	CHAR(8) NOT NULL	Name of the table space.	G
DBNAME	CHAR(8) NOT NULL	Name of the database that contains the table space.	G
IXNAME	VARCHAR(18) NOT NULL	Name of the partitioning index. This column is blank if the table space is not partitioned.	G
IXCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the partitioning index. This column is blank if the table space is not partitioned.	G
PQTY	INTEGER NOT NULL	Primary space allocation in units of 4KB storage blocks. For user-managed data sets, 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 1KB units.	G
SQTY	SMALLINT NOT NULL	Secondary space allocation in units of 4KB blocks. For user-managed data sets, 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 1KB units.	G
#		If the value does not fit into the column, the value of the column is 0. See the description of column SECQTYI.	
#			
STORTYPE	CHAR(1) NOT NULL	Type of storage allocation: E Explicit (storage group not used) I Implicit (storage group used)	G
STORNAME	CHAR(8) NOT NULL	Name of storage group used for space allocation. Blank if storage group not used.	G
VCATNAME	CHAR(8) 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

SYSIBM.SYSTABLEPART

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes C V2R1 dependency indicator; not from MRM tape E V2R3 dependency indicator; not from MRM tape I V6 dependency indicator for a LOB table space; not from MRM tape	G
LIMITKEY	VARCHAR(512) NOT NULL	The high value of the partition in external format. The value is 0 if the table space is not partitioned.	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	C The table space partition is in a check pending status and there are rows in the table that can violate referential constraints, table check constraints, or both. blank The table space is not a partition, or does not contain rows that may violate referential constraints, table check constraints, or both.	G
	CHAR(4) NOT NULL WITH DEFAULT FOR BIT DATA	Not used	N
SPACE	INTEGER NOT NULL WITH DEFAULT	Number of kilobytes of DASD storage allocated to the table space partition, as determined by the last execution of the STOSPACE utility or RUNSTATS utility. The value is 0 if STOSPACE or RUNSTATS has not been run. 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). The value is -1 if 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.	G
#			
#			
#			
#			
COMPRESS	CHAR(1) NOT NULL WITH DEFAULT	Indicates the following: • For a table space partition, whether the COMPRESS attribute for the partition is YES. • For a nonpartitioned table space, whether the COMPRESS attribute is YES for the table space. Values for the column can be: Y Compression is defined for the table space blank No compression	G
PAGESAVE	SMALLINT NOT NULL WITH DEFAULT	Percentage of pages saved in the table space or partition as a result of defining the table space with COMPRESS YES 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.	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'.	G
GBPCACHE	CHAR(1) NOT NULL WITH DEFAULT	Group buffer pool cache option specified for this table space or table space partition. A Changed and unchanged pages are cached in the group buffer pool. N No data is cached in the group buffer pool. S Only changed system pages, such as space map pages that do not contain actual data values, are cached in the group buffer pool. blank Only changed pages are cached in the group buffer pool.	G
CHECKRID5B	CHAR(5) NOT NULL WITH DEFAULT	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, table 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 is incremented whenever an 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 if RUNSTATS TABLESPACE with UPDATE(SPACE) or UPDATE(ALL) is executed; otherwise, the value is zero.	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'	Reserved.	S
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

SYSIBM.SYSTABLES table

Contains one row for each table, view, or alias.

Column name	Data type	Description	Use
NAME	VARCHAR(18) NOT NULL	Name of the table, view, or alias.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the table, view, or alias.	G
TYPE	CHAR(1) NOT NULL	Type of object: A Alias G Created global temporary table T Table V View X Auxiliary table	G
DBNAME	CHAR(8) 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	CHAR(8) 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. 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.	S
OBID	SMALLINT NOT NULL	Internal identifier of the table; 0 if the row describes a view, an alias, or a created temporary table.	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	CHAR(8) 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	CHAR(8) 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
	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

Column name	Data type	Description	Use
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N no Y Yes B V1R3 dependency indicator; not from MRM tape C V2R1 dependency indicator; not from MRM tape D V2R2 dependency indicator; not from MRM tape E V2R3 dependency indicator; not from MRM tape F V3R1 dependency indicator; not from MRM tape G V4 dependency indicator; not from MRM tape H V5 dependency indicator; not from MRM tape I V6 dependency indicator; not from MRM tape	G
REMARKS	VARCHAR(254) NOT NULL	A character string provided by the user with the COMMENT ON 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, or a created temporary 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, or a created temporary table.	G
KEYCOLUMNS	SMALLINT NOT NULL	Number of columns in the table's primary key. The value is 0 if the row describes a view, an alias, or a created temporary table.	G
RECLENGTH	SMALLINT NOT NULL	For user tables, the maximum length of any record in the table. Length is 8+N+L, where: <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 a column's maximum length, 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 66. <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 on page 592.</p>	G
STATUS	CHAR(1) NOT NULL	Indicates the status of the table definition: <p>I The definition of the table is incomplete. The TABLESTATUS column indicates the reason for the table definition being incomplete.</p> <p>X The table has a parent index and the table definition is complete.</p> <p>blank The table has no parent index, or is a catalog table, or the row describes a view or alias. The definition of the table, view, or alias is complete.</p>	G
KEYOBID	SMALLINT NOT NULL	Internal DB2 identifier of the index that enforces uniqueness of the table's primary key; 0 if not applicable.	S
LABEL	VARCHAR(30) NOT NULL	The label as given by a LABEL ON statement; otherwise an empty string.	G

SYSIBM.SYSTABLES

Column name	Data type	Description	Use
CHECKFLAG	CHAR(1) NOT NULL WITH DEFAULT	C The table space that contains the table is in a check pending status and there are rows in the table that can violate referential constraints, table check constraints, or both. blank The table contains no rows that violate referential constraints, table check constraints, or both; or the row describes a view, alias, or created temporary table.	G
	CHAR(4) NOT NULL WITH DEFAULT FOR BIT DATA	Not used	N
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	CHAR(8) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the table, view, or alias.	G
LOCATION	CHAR(16) NOT NULL WITH DEFAULT	Location name of the object of an alias. Blank for a table, a view, or for an alias that was not defined with a three-part object name.	G
TBCREATOR	CHAR(8) NOT NULL WITH DEFAULT	For an alias, the authorization ID of the owner of the referred to table or view; blank otherwise.	G
TBNAME	VARCHAR(18) NOT NULL WITH DEFAULT	For an alias, the name for the referred to table or view; blank otherwise.	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 a view or alias, ALTEREDTS has the value of CREATEDTS.	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, or a created temporary 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	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, table 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'	Default encoding scheme for tables, views, and local aliases: E EBCDIC A ASCII 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, DSND04.	G
TABLESTATUS	VARCHAR(10) NOT NULL WITH DEFAULT	Indicates the reason for an incomplete table definition: L Definition is incomplete because an auxiliary table or auxiliary index has not been defined for a LOB column. P Definition is incomplete because the table lacks a parent index. R Definition is incomplete because the table lacks a required index on a row ID. blank Definition is complete.	G

SYSIBM.SYSTABLESPACE table

Contains one row for each table space.

Column name	Data type	Description	Use
NAME	CHAR(8) NOT NULL	Name of the table space.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the table space.	G
DBNAME	CHAR(8) 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	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 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
IMPLICIT	CHAR(1) NOT NULL	Whether the table space was created implicitly: N No Y Yes	G
NTABLES	SMALLINT NOT NULL	Number of tables defined in the table space.	G
NACTIVE	INTEGER NOT NULL	Number of active pages in the table space. A page is termed active if it is formatted for rows, even if it currently contains none. The value is 0 if statistics have not been gathered. This is an updatable column.	S
	CHAR(8) NOT NULL	Not used	N

Column name	Data type	Description	Use
CLOSERULE	CHAR(1) NOT NULL	Whether the data sets are candidates for closure when the limit on the number of open data sets is reached. N No Y Yes	G
SPACE	INTEGER NOT NULL	Number of kilobytes of DASD storage allocated to the table space, as determined by the last execution of the STOSPACE utility. The value is 0 if the table space is not related to a storage group, or if STOSPACE has not been run. If the table space is partitioned, the value is the total kilobytes of DASD storage allocated to all partitions that are storage group defined.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes C V2R1 dependency indicator; not from MRM tape F V3R1 dependency indicator; not from MRM tape G V4 dependency indicator; not from MRM tape H V5 dependency indicator; not from MRM tape I V6 dependency indicator for a LOB table space; not from MRM tape J Release dependency marker for a table space that contains a table with an identity column; not from MRM tape	G
	VARCHAR(18) NOT NULL	Internal use only	I
	CHAR(8) NOT NULL	Internal use only	I
SEGSIZE	SMALLINT NOT NULL WITH DEFAULT	Number of pages in each segment of a segmented table space. The value is 0 if the table space is not segmented.	G
CREATEDBY	CHAR(8) NOT NULL WITH DEFAULT	Primary authorization ID of the user who created the table space.	G
STATSTIME	TIMESTAMP NOT NULL WITH DEFAULT	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics. The default value is '0001-01-01.00.00.00.000000'. This is an updatable column.	G
LOCKMAX	INTEGER	The maximum number of locks per user to acquire for the table or table space before escalating to the next locking level. 0 Lock escalation does not occur. n n, where n > 0, is the maximum number of locks (row, page, or LOB locks for the table or table space) an application process can acquire before lock escalation occurs. -1 Represents LOCKMAX SYSTEM. The value of field LOCKS PER TABLE(SPACE) on installation panel DSNTIPJ determines lock escalation. If the value of the field is 0, lock escalation does not occur. If the value is n, where n > 0, lock escalation occurs as it does for LOCKMAX n.	G

|

#

|

SYSIBM.SYSTABLESPACE

Column name	Data type	Description	Use
TYPE	CHAR(1) NOT NULL WITH DEFAULT	The type of table space: blank The table space was created without any of the following options: DSSIZE, LARGE, LOB, and MEMBER CLUSTER. I The table space was defined with the MEMBER CLUSTER option and is not greater than 64 gigabytes. K The table space was defined with the MEMBER CLUSTER option and can be greater than 64 gigabytes. L The table space can be greater than 64 gigabytes. O The table space was defined with the LOB option (the table space is a LOB table space).	G
CREATEDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the CREATE statement was executed for the table space. If the table space was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ALTEREDTS	TIMESTAMP NOT NULL WITH DEFAULT	Time when the most recent ALTER TABLESPACE statement was executed for the table space. If no ALTER TABLESPACE statement has been applied, ALTEREDTS has the value of CREATEDTS. If the index was created in a DB2 release prior to Version 5, the value is '0001-01-01.00.00.00.000000'.	G
ENCODING_SCHEME	CHAR(1) NOT NULL WITH DEFAULT 'E'	Default encoding scheme for the table space: E EBCDIC A ASCII blank For tables spaces in a work file database or a TEMP database (a database that was created AS TEMP, which is for declared temporary tables.) The value is 'E' for tables in non work file databases and blank for tables in work file databases created prior to Version 5 or the default database, DSNDB04.	G
# SBCS_CCSID	INTEGER NOT NULL WITH DEFAULT	Default SBCS CCSID for the table space. For a table space in a 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 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 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
LOCKPART	CHAR(1) NOT NULL WITH DEFAULT	Y LOCKPART YES is specified for the table space. blank LOCKPART NO is specified, or LOCKPART is not specified or not a partitioned table space.	G
LOG	CHAR(1) NOT NULL WITH DEFAULT 'Y'	Whether the changes to a table space are to be logged. N No, only applies to LOB table spaces Y Yes	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

SYSIBM.SYSTABSTATS table

Contains one row for each partition of a partitioned table space. Rows in this table can be inserted, updated, and deleted.

Column name	Data type	Description	Use
CARD	INTEGER NOT NULL	Total number of rows in the partition.	S
NPAGES	INTEGER NOT NULL	Total number of pages on which rows of the partition appear.	S
PCTPAGES	SMALLINT NOT NULL	Percentage of total active pages in the partition that contain rows of the table.	S
NACTIVE	INTEGER NOT NULL	Number of active pages in the partition.	S
PCTROWCOMP	SMALLINT NOT NULL	Percentage of rows compressed within the total number of active rows in the partition. This includes any row in a table space that is defined with COMPRESS YES.	S
STATSTIME	TIMESTAMP NOT NULL	If RUNSTATS updated the statistics, the date and time when the last invocation of RUNSTATS updated the statistics.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
DBNAME	CHAR(8) NOT NULL	Database that contains the table space named in TSNAME.	G
TSNAME	CHAR(8) 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	CHAR(8) NOT NULL	Authorization ID of the owner of the table.	G
NAME	VARCHAR(18) NOT NULL	Name of the table.	G
CARDF	FLOAT NOT NULL WITH DEFAULT -1	Total number of rows in the partition.	S

SYSIBM.SYSTRIGGERS table

Contains one row for each trigger.

Column name	Data type	Description	Use
NAME	CHAR(8) NOT NULL	Name of the trigger and trigger package.	G
SCHEMA	CHAR(8) 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	CHAR(8) NOT NULL	Authorization ID of the owner of the trigger. The value is set to the current authorization ID (the plan or packge owner for static CREATE TRIGGER statement; the current SQLID for a dynamic CREATE TRIGGER statement).	G
CREATEDBY	CHAR(8) NOT NULL	Authorization ID of the owner of the trigger. The value is set to the current authorization ID (the plan or packge owner for static CREATE TRIGGER statement; the current SQLID for a dynamic CREATE TRIGGER statement).	G
TBNAME	VARCHAR(18) NOT NULL	Name of the table to which this trigger applies.	G
TBOWNER	CHAR(8) NOT NULL	Qualifier of the name of the table 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: B Trigger is applied before the event. A Trigger is applied after 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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
TEXT	VARCHAR(3460) NOT NULL	Full text of the CREATE TRIGGER statement.	G
REMARKS	VARCHAR(254) NOT NULL	A character string provided by the user with the COMMENT ON statement.	G

SYSIBM.SYSUSERAUTH table

Records the system privileges that are held by users.

Column name	Data type	Description	Use
GRANTOR	CHAR(8) NOT NULL	Authorization ID of the user who granted the privileges.	G
GRANTEE	CHAR(8) 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
DATEGRANTED	CHAR(6) NOT NULL	Date the privileges were granted; in the form <i>yymmdd</i> .	G
TIMEGRANTED	CHAR(8) NOT NULL	Time the privileges were granted; in the form <i>hhmmssth</i> .	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.	G
		blank Not applicable C DBCTL D DBADM L SYSCTRL M DBMAINT S SYSADM	
	CHAR(1) NOT NULL	Not used	N
BINDADDAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can use the BIND subcommand with the ADD option:	G
		blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	
BSDSAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can issue the RECOVER BSDS command:	G
		blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	
CREATEDBAAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can create databases and automatically receive DBADM authority over the new databases:	G
		blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	
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:	G
		blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	
CREATESGAUTH	CHAR(1) NOT NULL	Whether the GRANTEE can execute the CREATE STOGROUP statement to create new storage groups:	G
		blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	

SYSIBM.SYSUSERAUTH

Column name	Data type	Description	Use
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	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
MON1AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can obtain IFC serviceability data: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
MON2AUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can obtain IFC data: blank Privilege is not held G Privilege is held with the GRANT option Y Privilege is held without the GRANT option	G
CREATEALIASAUTH	CHAR(1) NOT NULL WITH DEFAULT	Whether the GRANTEE can execute the CREATE ALIAS statement: blank Privilege is not held G Privilege held with the GRANT option Y Privilege held without the GRANT option	G

Column name	Data type	Description	Use
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 See "GRANT (system privileges)" on page 730 for a description of the BINDAGENT privilege.	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
CREATETMTABAUTH	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

SYSIBM.SYSVIEWDEP table

Records the dependencies of views on tables, functions, and other views.

Column name	Data type	Description	Use
BNAME	VARCHAR(18) NOT NULL	Name of the object on which the view is dependent. If the object type is a function (BTYPE='F'), the name is the specific name of the function.	G
BCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of BNAME. For functions, it is the schema name of the BNAME.	G
BTYPE	CHAR(1) NOT NULL	Type of object: F Function T Table V View	G
DNAME	VARCHAR(18) NOT NULL	Name of the view.	G
DCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the view.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G
BSCHEMA	CHAR(8) NOT NULL WITH DEFAULT	Schema of BNAME.	G

SYSIBM.SYSVIEWS table

Contains one or more rows for each view.

Column name	Data type	Description	Use
NAME	VARCHAR(18) NOT NULL	Name of the view.	G
CREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the view.	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.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes B V1R3 dependency indicator; not from MRM tape C V2R1 dependency indicator; not from MRM tape D V2R2 dependency indicator; not from MRM tape E V2R3 dependency indicator; not from MRM tape F V3R1 dependency indicator; not from MRM tape G V4 dependency indicator; not from MRM tape H V5 dependency indicator; not from MRM tape I V6 dependency indicator; not from MRM tape	G
TEXT	VARCHAR(254) NOT NULL	Text or portion of the text of the CREATE VIEW statement.	G
PATHSCHEMAS	VARCHAR(254) NOT NULL WITH DEFAULT	SQL path at the time the view was defined. The path is used to resolve unqualified data type and function names used in the view definition.	G

SYSIBM.SYSVOLUMES table

Contains one row for each volume of each storage group.

Column name	Data type	Description	Use
SGNAME	CHAR(8) NOT NULL	Name of the storage group.	G
SGCREATOR	CHAR(8) NOT NULL	Authorization ID of the owner of the storage group.	G
VOLID	CHAR(6) NOT NULL	Serial number of the volume or * if SMS-managed.	G
IBMREQD	CHAR(1) NOT NULL	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.USERNAMES table

Each row in the table is used to carry out one of the following operations:

- Outbound ID translation
- 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: O For outbound translation. I For inbound translation and “come from” checking.	G
AUTHID	CHAR(8) NOT NULL WITH DEFAULT	Authorization ID to be translated. Applies to any authorization ID if blank.	G
LINKNAME	CHAR(8) 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 nonblank 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	CHAR(8) NOT NULL WITH DEFAULT	Translated value of AUTHID. Blank specifies no translation.	G
PASSWORD	CHAR(8) NOT NULL WITH DEFAULT	Password to accompany an outbound request, if passwords are not encrypted. If passwords are encrypted, or the row is for inbound requests, the column is not used.	G
IBMREQD	CHAR(1) NOT NULL WITH DEFAULT 'N'	Whether the row came from the basic machine-readable material (MRM) tape: N No Y Yes	G

SYSIBM.USERNAMES

Appendix E. SQL reserved words

Table 67 on page 1028 lists the words that cannot be used as ordinary identifiers in some contexts because they 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. In addition, some sections of this book might indicate words that cannot be used in the specific context that is being described.

SQL reserved words

Table 67. SQL reserved words

ADD	CURRENT_TIME	GOTO	NULL	SECQTY
AFTER	CURRENT_TIMESTAMP	GRANT	NULLS	SECURITY
ALL	CURSOR	GROUP	NUMPARTS	SELECT
# ALLOCATE	DATA	HANDLER	OBID	SET
ALLOW	DATABASE	HAVING	OF	SIMPLE
ALTER	DAY	HOUR	ON	SOME
# AND	DAYS	HOURS	OPEN	SOURCE
# ANY	DBINFO	IF	OPTIMIZATION	SPECIFIC
AS	DB2SQL	IMMEDIATE	OPTIMIZE	STANDARD
# ASSOCIATE	DECLARE	IN	OR	STAY
ASUTIME	DEFAULT	INDEX	ORDER	STOGROUP
AUDIT	DELETE	INNER	OUT	STORES
AUX	DESCRIPTOR	INOUT	OUTER	STYLE
AUXILIARY	DETERMINISTIC	INSERT	PACKAGE	SUBPAGES
BEFORE	DISALLOW	INTO	PARAMETER	SYNONYM
BEGIN	DISTINCT	IS	PART	SYSFUN
# BETWEEN	DO	ISOBID	PATH	SYSIBM
# BUFFERPOOL	DOUBLE	JAVA	PIECESIZE	SYSPROC
BY	DROP	JOIN	PLAN	SYSTEM
CALL	DSSIZE	KEY	PRECISION	TABLE
# CAPTURE	DYNAMIC	LABEL	PREPARE	TABLESPACE
CASCADED	EDITPROC	LANGUAGE	PRIQTY	THEN
CASE	ELSE	LC_CTYPE	PRIVILEGES	TO
# CAST	ELSEIF	LEAVE	PROCEDURE	TRIGGER
# CCSID	END	LEFT	PROGRAM	UNDO
CHAR	END-EXEC1	LIKE	PSID	UNION
CHARACTER	ERASE	LOCAL	QUERYNO	UNIQUE
# CHECK	ESCAPE	LOCALE	READS	UNTIL
# CLOSE	EXCEPT	LOCATOR	REFERENCES	UPDATE
# CLUSTER	EXECUTE	LOCATORS	RELEASE	USER
# COLLECTION	EXISTS	LOCK	RENAME	USING
# COLLID	EXIT	LOCKMAX	REPEAT	VALIDPROC
COLUMN	EXTERNAL	LOCKSIZE	RESTRICT	VALUES
# COMMENT	FENCED	LONG	RESULT	VARIANT
# COMMIT	FETCH	LOOP	RESULT_SET_LOCATOR	VIEW
CONCAT	FIELDPROC	MICROSECOND	RETURN	VIEW
# CONDITION	FINAL	MICROSECONDS	RETURNS	VOLUMES
# CONNECT	FOR	MINUTE	REVOKE	WHEN
CONNECTION	FROM	MINUTES	RIGHT	WHERE
# CONSTRAINT	FULL	MODIFIES	ROLLBACK	WHILE
CONTAINS	FUNCTION	MONTH	RUN	WITH
# CONTINUE	GENERAL	MONTHS	SAVEPOINT	WLM
# CREATE	GENERATED	NO	SCHEMA	YEAR
# CURRENT	GET	NOT	SCRATCHPAD	YEARS
# CURRENT_DATE	GLOBAL		SECOND	
CURRENT_LC_CTYPE	GO		SECONDS	
CURRENT_PATH				

Note: 1COBOL only

IBM SQL has additional reserved words that DB2 for OS/390 does not enforce. Therefore, we suggest that you do not use these additional reserved words as ordinary identifiers in names that have a continuing use. See *IBM SQL Reference* for a list of the words.

Appendix F. Sample user-defined functions

This appendix describes the sample user-defined functions that are provided with DB2. You can use the functions in the following ways:

- In your applications just as you would use other user-defined functions. 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*.
- As examples to help you define and implement your own user-defined functions. Data set *prefix.SDSNSAMP* contains the code for the sample functions.

Table 68 lists the sample user-defined functions. The detailed descriptions of the functions that follow the table include their external program names and specific names. The functions are in schema DSN8.

Table 68. DB2 sample user-defined functions

Function Name	Description	Page
ALTDATE	Returns the current date or a user-specified date in a user-specified format	1030
ALTTIME	Returns the current time or a user-specified time in a user-specified format	1033
CURRENCY	Returns a floating-point number as a currency value	1035
DAYNAME	Returns the name of the day of the week on which a date in ISO format falls	1037
MONTHNAME	Returns the name of the month in which a date in ISO format falls	1038
TABLE_LOCATION	Returns the location name of a table or view after resolving any aliases	1039
TABLE_NAME	Returns the unqualified name of a table or view after resolving any aliases	1041
TABLE_SCHEMA	Returns the schema name of a table or view after resolving any aliases	1043
WEATHER	Shows how to use a user-defined table function to make non-relational data available for SQL manipulation	1045

ALTDATA

ALTDATA(*input date, input format, output format*)

The schema is DSN8.

The ALTDATA 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 ALTDATA 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 ALTDATA function has two forms.

Form 1: ALTDATA(*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 has a VARCHAR data type and an actual length that is not greater than 17 bytes.

Form 2: ALTDATA(*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 has a VARCHAR data type and an actual length that is not greater than 17 bytes.

Table 69 shows the external and specific names for the two forms of the function, which are based on the input to the function.

Table 69. External program and specific names for ALTDATE

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.ALTDATE( 'DD MONTH YY' );
```

Example 2: Convert the current date into format 'D.M.YYYY', a format that will suppress any leading zero for the day or month and include the year with the century.

```
VALUES DSN8.ALTDATE( 'D.M.YYYY' );
```

Example 3: Convert the current date into format 'YYYY-XX-DD', a format that will include the century, the month of the year as a roman numeral, and the day of the month with any leading zero.

```
VALUES DSN8.ALTDATE( 'YYYY-XX-DD' );
```

Example 4: Convert a date in the format of 'DD MONTH YYYY' to a date in the format of 'YYYY/MM/DD'.

```
VALUES DSN8.ALTDATE( '11 November 1918',  
                    'DD MONTH YYYY',  
                    'YYYY/MM/DD' );
```

ALTDATE

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 FIRSTNAME || ' '
       || LASTNAME || ' was hired on '
       || DSN8.ALTDATE( HIREDATE,
                       'YYYY-MM-DD',
                       'D.M.YY' )
FROM EMP
WHERE EMPNO = '000130';
```

Assuming that the HIREDATE is 1971-07-28, the above example returns: DELORES QUINTANA was hired on 28.7.71.

ALTTIME

▶ `ALTTIME(` `input time, input format,` `)` *output format* ▶

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 has a VARCHAR data type and an actual length that is not greater than 11 bytes.

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 has a VARCHAR data type and an actual length that is not greater than 11 bytes.

Table 70 shows the external program and specific names for the two forms of the function, which are based on the input to the function.

Table 70. 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 date</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

► CURRENCY(*-input amount, currency symbol* [, *credit/debit indicator*]) ►

The schema is DSN8.

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.

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 a character string with a data type of VARCHAR and an actual length that is not greater than 19 bytes.

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.

Table 71 shows the external program and specific names for CURRENCY. The specific names differ depending on the input to the function.

Table 71. 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>debit/credit indicator</i>	DSN8DUCY	DSN8.DSN8DUCYFVV

CURRENCY

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

► DAYNAME(*input date*) ►

The schema is DSN8.

The DAYNAME function returns the name of the weekday on which a given date falls. The name is returned in English.

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 a character string with a data type of VARCHAR and an actual length that is not greater than 9 bytes.

The DAYNAME function uses the IBM C++ class *IDate*.

Table 72 shows the external and specific names for DAYNAME. The specific names differ depending on the data type of the input argument.

Table 72. 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 2000.

```
VALUES DSN8.DAYNAME( '2000-02-29' );
```

The result of the function is Tuesday.

Example 3: Find the day of the week on which Delores Quintana, employee number 000130, was hired.

```
SELECT  FIRSTNME || ' '
        || LASTNAME || ' was hired on '
        || DSN8.DAYNAME( HIREDATE ) || ', '
        || CHAR( HIREDATE )
FROM    EMP
WHERE   EMPNO = '000130';
```

The result of the function is DELORES QUINTANA was hired on Wednesday, 1971-07-28.

MONTHNAME

▶—MONTHNAME(*input date*)—▶

The schema is DSN8.

The MONTHNAME function returns the calendar name of the month in which a given date falls. The name is returned in English.

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 a character string with a data type of VARCHAR and an actual length that is not greater than 9 bytes.

The MONTHNAME function uses the IBM C++ class *IDate*.

Table 73 shows the external and specific names for MONTHNAME. The specific names differ depending on the data type of the input argument.

Table 73. 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.

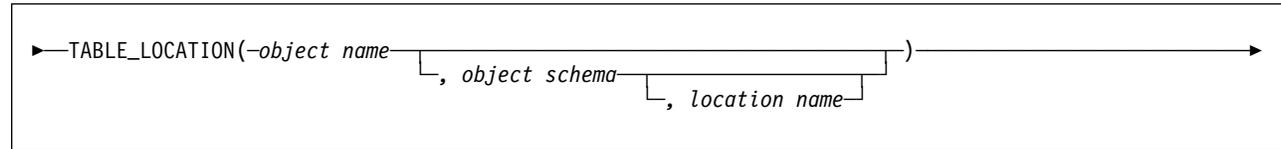
```
VALUES DSN8.MONTHNAME( CURRENT DATE );
```

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 schema is DSN8.

The TABLE_LOCATION function searches for an object and returns the location name of the object after any alias chains have been resolved. 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.

object location

A character expression that represents the location that is used to qualify the value specified in *object name* before resolution. *object location* must have a data type of VARCHAR and an actual length that is no greater than 16 bytes.

If *object location* is not specified or is null, the location name is equivalent to "any."

The result of the function has a data type of VARCHAR and an actual length that is no greater than 16 bytes. If *object name* can be null, the result can be null; if *object name* is null, the result is the null value.

Table 74 shows the external and specific names for TABLE_LOCATION. The specific names differ depending on the number of input arguments to the function.

Table 74. 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>schema name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILVV
<i>object name</i> (VARCHAR) <i>schema name</i> (VARCHAR) <i>location name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTILVVV

TABLE_LOCATION

Example: Assume that:

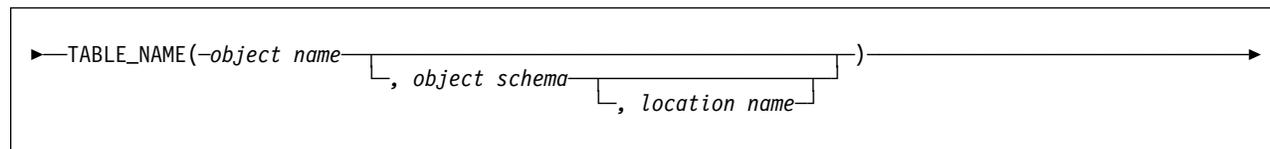
- DSN8.ALIAS_RS_SYSTABLES is an alias of SYSIBM.SYSTABLES at location name REMOTE_SITE.
- The current SQLID is DSN8.

Use TABLE_LOCATION to find the location name where the base object for ALIAS_OF_SYSTABLES resides.

```
VALUES DSN8.TABLE_LOCATION( 'ALIAS_RS_SYSTABLES' );
```

The result of the function is REMOTE_SITE.

TABLE_NAME



The schema is DSN8.

The TABLE_NAME function searches for an object and returns the unqualified name of the object after any alias chains have been resolved. 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.

object location

A character expression that represents the location that is used to qualify the value specified in *object name* before resolution. *object location* must have a data type of VARCHAR and an actual length that is no greater than 16 bytes.

If *object location* is not specified or is null, the location name is equivalent to "any."

The result of the function has a data type of VARCHAR and an actual length that is no greater than 18 bytes. If *object name* can be null, the result can be null; if *object name* is null, the result is the null value.

Table 75 shows the external and specific names for TABLE_NAME. The specific names differ depending on the number of input arguments to the function.

Table 75. 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>schema name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINVV
<i>object name</i> (VARCHAR) <i>schema name</i> (VARCHAR) <i>location name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTINVVV

TABLE_NAME

Example: Assume that:

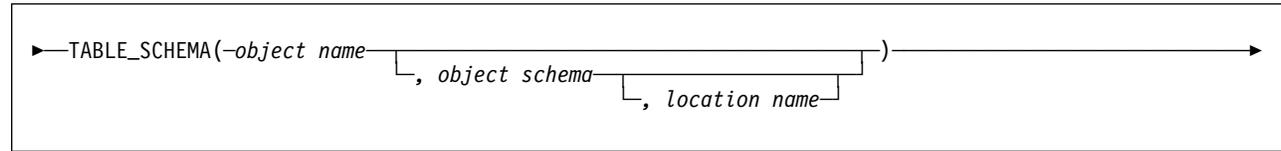
- DSN8.VIEW_OF_SYSTABLES is a view of SYSIBM.SYSTABLES.
- DSN8.ALIAS_OF_VIEW is an alias of DSN8.VIEW_OF_SYSTABLES.
- The current SQLID is DSN8.

Use TABLE_NAME to find the name of the base object for ALIAS_OF_VIEW.

```
VALUES DSN8.TABLE_NAME( 'ALIAS_OF_SYSVIEW' );
```

The result of the function is VIEW_OF_SYSTABLES.

TABLE_SCHEMA



The schema is DSN8.

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 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.

object location

A character expression that represents the location that is used to qualify the value specified in *object name* before resolution. *object location* must have a data type of VARCHAR (and an actual length that is no greater than 16 bytes).

If *object location* is not specified or is null, the location name is equivalent to "any."

The result of the function has a data type of VARCHAR and an actual length that is no greater than 8 bytes. If *object name* can be null, the result can be null; if *object name* is null, the result is the null value.

Table 76 shows the external and specific names for TABLE_SCHEMA. The specific names differ depending on the number of input arguments.

Table 76. 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>schema name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTISVV
<i>object name</i> (VARCHAR) <i>schema name</i> (VARCHAR) <i>location name</i> (VARCHAR)	DSN8DUTI	DSN8.DSN8DUTISVVV

TABLE_SCHEMA

Example: Assume that:

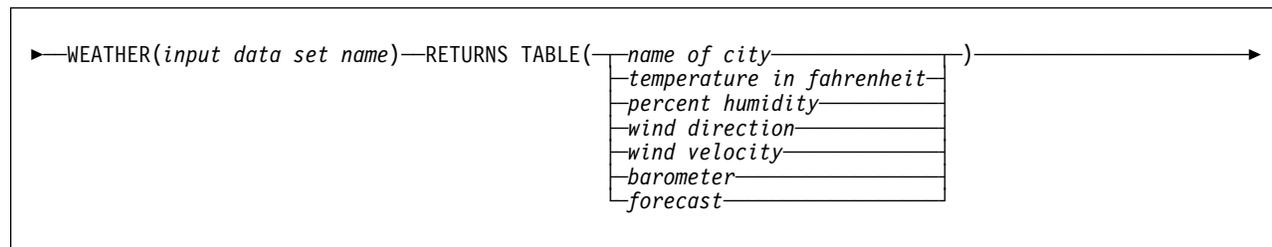
- DSN8.ALIAS_OF_SYSTABLES is an alias of SYSIBM.SYSTABLES.
- The current SQLID is DSN8.

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 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. The WEATHER function is provided primarily to help you design and implement table functions.

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.

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:

Bismark, ND	Slight chance of snow
Cheyenne, WY	Continued cooling
Helena, MT	Heavy snow
Pierre, SD	Continued cold

WEATHER

Appendix G. DB2 objects required by the DB2 for OS/390 SQL procedure processor

The DB2 for OS/390 SQL procedure processor (DSNTPSMP) uses the tables and indexes that are described in the following sections. You can create these objects by customizing and running job DSNTIJSQ, which is in data set DSN610.SDSNSAMP. DSNTIJSQ creates the objects in database DSNPSM and table space DSNPSM.

Table spaces and indexes

1047 shows the table spaces to which the SQL procedure tables are assigned, and which indexes are defined on the tables.

Table 77. Table spaces and indexes for SQL procedure tables

TABLE SPACE DSNPSM. ...	TABLE SYSIBM. ...	Page	INDEX SYSIBM. ...	INDEX FIELDS
DSNPSM	SYSPSM	1047	DSNPSMX1	PROCEDURENAME
			DSNPSMX2	SCHEMA PROCEDURENAME SEQNO
	SYSPSMOPTS	1047	DSNPSMOX1	SCHEMA PROCEDURENAME

The SQL procedure source table (SYSIBM.SYSPSM)

SYSIBM.SYSPSM is used by the SQL procedure processor and IBM DB2 Stored Procedure Builder to hold the source code for a stored procedure.

SYSIBM.SYSPSM contains at least one row for each SQL procedure that is prepared by the SQL procedure processor or SQL Procedure Builder. The number of rows that represent an SQL procedure is

$$\text{CEILING}(n/3800)$$

n is the number of bytes in the SQL procedure source statement.

Column Name	Data Type	Description	Use
SCHEMA	CHAR(8)	Schema of the SQL procedure. Blank for SQL procedures created before DB2 Version 6.	G
PROCEDURENAME	CHAR(18) NOT NULL	Name of the SQL procedure.	G
SEQNO	SMALLINT NOT NULL	Number of the SQL statement piece in PROCCREATESTMT. SEQNO is between 1 and CEILING($n/3800$), where n is the number of bytes in the SQL procedure source statement.	G
PSMDATE	DATE NOT NULL	The date on which the SQL procedure was created.	G
PSMTIME	TIME NOT NULL	The time at which the SQL procedure was created.	G
PSMTIME	TIME NOT NULL	The time at which the SQL procedure was created.	G

Column Name	Data Type	Description	Use
PROCCREATESTMT	VARCHAR(3800) NOT NULL	All or part of an SQL procedure source statement. If the SQL procedure statement is more than 3800 bytes, this field contains the portion of the source statement indicated by SEQNO.	G

The SQL procedure options table (SYSIBM.SYSPSMOPTS)

SYSIBM.SYSPSMOPTS is used by the SQL procedure processor and IBM DB2 Stored Procedure Builder to hold the program preparation options for an SQL procedure. SYSIBM.SYSPSMOPTS contains one row for each SQL procedure that is prepared by the SQL procedure processor or SQL Procedure Builder.

Column Name	Data Type	Description	Use
SCHEMA	CHAR(8)	Schema of the SQL procedure. Blank for SQL procedures created before DB2 Version 6.	G
PROCEDURENAME	CHAR(18) NOT NULL	Name of the SQL procedure.	G
BUILDSHEMA	CHAR(8)	The schema name that is the qualifier for the procedure name that is specified in the BUILDNAME column.	G
BUILDNAME	CHAR(18)	A procedure name that is associated with stored procedure DSNTPSMP. Users of DSNTPSMP might create several stored procedure definitions for DSNTPSMP so that they can run DSNTPSMP in different WLM environments. The caller specifies the environment in which DSNTPSMP runs by specifying the procedure name that is associated with that environment in the SQL CALL statement.	G
BUILDOWNER	CHAR(8)	The authorization ID that was used to create the SQL procedure.	G
PRECOMPILE_OPTS	VARCHAR(255)	The options that were specified in the <i>precompiler-options</i> parameter in the most recent invocation of DSNTPSMP for the SQL procedure specified in this row.	G
COMPILE_OPTS	VARCHAR(255)	The options that were specified in the <i>compiler-options</i> parameter in the most recent invocation of DSNTPSMP for the SQL procedure specified in this row.	G
PRELINK_OPTS	VARCHAR(255)	The options that were specified in the <i>prelink-edit-options</i> parameter in the most recent invocation of DSNTPSMP for the SQL procedure specified in this row.	G
LINK_OPTS	VARCHAR(255)	The options that were specified in the <i>link-edit-options</i> parameter in the most recent invocation of DSNTPSMP for the SQL procedure specified in this row.	G
BIND_OPTS	VARCHAR(1024)	The options that were specified in the <i>bind-options</i> parameter in the most recent invocation of DSNTPSMP for the SQL procedure specified in this row.	G
SOURCEDSN	VARCHAR(255)	If the SQL procedure source code that is input to DSNTPSMP is stored in a data set, the name of that data set.	G

Created temporary table **SYSIBM.SYSPSMOUT**

SYSIBM.SYSPSMOUT is used by the SQL procedure processor and IBM DB2 Stored Procedure Builder to hold error information that is returned in a result set.

This SQL statement creates the temporary table:

```
CREATE GLOBAL TEMPORARY TABLE SYSIBM.SYSPSMOUT
  (STEP          VARCHAR(16),
   FILE          VARCHAR(8),
   SEQN          INTEGER,
   LINE         VARCHAR(255) )
CCSID EBCDIC;
```

Appendix H. 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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country 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 publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is as your own risk.

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
J74/G4
555 Bailey Avenue
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information 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 illustrates 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.

Programming interface information

This book is intended to help you to code SQL statements. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by IBM DATABASE 2 Universal Database Server for OS/390 (DB2 for OS/390).

General-use programming interfaces allow the customer to write programs that obtain the services of DB2 for OS/390.

However, this book also documents 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 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 an introductory statement to a chapter or section or by the following marking:

```

┌────────────────── Product-sensitive Programming Interface ───────────────────┐
Product-sensitive Programming Interface and Associated Guidance Information ...
└────────────────── End of Product-sensitive Programming Interface ───────────────────┘
  
```

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AD/Cycle	DRDA
AIX	IBM
APL2	IMS
AS/400	IMS/ESA
BookManager	Language Environment
CICS	MVS/ESA
CICS/ESA	MVS/XA
CICS/MVS	Net.Data
COBOL/370	OS/2
C/370	OS/390
DATABASE 2	OS/400
DataPropagator	Parallel Sysplex
DB2	QMF
DB2 Extenders	RACF
DB2 Universal Database	SQL/DS
DFSMSdfp	System/370
DFSMSHsm	System/390
DFSMS/MVS	VTAM
Distributed Relational Database Architecture	

Lotus and Notes are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Microsoft™, Windows™, Windows NT™, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, or other countries, or both.

NetView™ is a trademark of Tivoli Systems Inc. in the United States, or other countries, or both.

Bibliography

Other company, product, and service names may be trademarks or service marks of others.

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library. If you do not find the term you are looking for, refer to the index or to *IBM Dictionary of Computing*.

A

abend. Abnormal end of task.

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with DB2. A complete list of DB2 abend reason codes and their explanations is contained in *DB2 Messages and Codes*.

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 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.

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, whereas the archive log holds those records that are older and no longer fit on the active log.

active member state. A state of a member of a data sharing group. An active member is identified with a group by XCF, which associates the member with a particular task, address space, and MVS system. A member that is not active has either a failed member state or a quiesced member state.

after trigger. A trigger that is defined with the trigger activation time AFTER.

agent. As used in DB2, the structure that associates all processes that are involved in a DB2 unit of work. An *allied agent* is generally synonymous with an *allied thread*. *System agents* are units of work that process independently of the allied agent, such as prefetch processing, deferred writes, and service tasks.

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.

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 stored procedure result sets by using the SQL ALLOCATE CURSOR statement.

already verified. An LU 6.2 security option that allows DB2 to provide the user's verified authorization ID when allocating a conversation. The user is not validated by the partner DB2 subsystem.

ambiguous cursor. A database cursor that is not defined with the FOR FETCH ONLY clause or the FOR UPDATE OF clause, is not defined on a read-only result table, is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement, and is in a plan or package that contains either PREPARE or EXECUTE IMMEDIATE SQL statements.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

API. 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 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 orderable 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 (AR). See *requester*.

application server (AS). See *server*.

AR. Application requester. See *requester*.

archive log • call level interface (CLI)

archive log. The portion of the DB2 log that contains log records that have been copied from the active log.

AS. Application server. See *server*.

ASCII. An encoding scheme that is used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC*.

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 one of that employee's attributes.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges are allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

auxiliary index. An index on an auxiliary table in which each index entry refers to a LOB.

auxiliary table. A table that stores columns outside the table in which they are defined. Contrast with *base table*.

B

base table. (1) A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with *result table* and *temporary table*.

(2) A table containing a LOB column definition. The actual LOB column data is not stored with the base table. The base table contains a row identifier for each row and an indicator column for each of its LOB columns. Contrast with *auxiliary table*.

base table space. A table space that contains base tables.

basic predicate. A predicate that compares two values.

before trigger. A trigger that is defined with the trigger activation time BEFORE.

binary integer. A basic data type that can be further classified as small integer or large integer.

binary large object (BLOB). A sequence of bytes, where the size of the value ranges from 0 bytes to 2 GB - 1. Such a string does not have an associated CCSID.

binary string. A sequence of bytes that is not associated with a CCSID. For example, the BLOB data type is a binary string.

bind. The process by which the output from the DB2 precompiler is converted to a usable control structure (which is called a package or an application plan). During the process, access paths to the data are selected and some authorization checking is performed.

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.

dynamic bind. A process by which SQL statements are bound as they are entered.

incremental bind. A process by which SQL statements are bound during the execution of an application process, because they could not be bound during the bind process, and VALIDATE(RUN) was specified.

static bind. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time.

bit data. Data that is character type CHAR or VARCHAR and is not associated with a coded character set.

BLOB. Binary large object.

BMP. Batch Message Processing (IMS).

bootstrap data set (BSDS). A VSAM data set that contains name and status information for DB2, as well as RBA range specifications, for all active and archive log data sets. It also contains passwords for the DB2 directory and catalog, and lists of conditional restart and checkpoint records.

BSDS. Bootstrap data set.

buffer pool. Main storage that is reserved to satisfy the buffering requirements for one or more table spaces or indexes.

built-in function. A function that DB2 supplies. Contrast with *user-defined function*.

C

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.

call level interface (CLI). A callable application

programming interface (API) for database access, which is an alternative to using embedded SQL. In contrast to embedded SQL, DB2 ODBC (which is based on the CLI architecture) does not require the user to precompile or bind applications, but instead provides a standard set of functions to process SQL statements and related services at run time.

cascade delete. The way in which DB2 enforces referential constraints when it deletes all descendent rows of a deleted parent row.

CASE expression. Allows an expression to be 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. In general, a cast function has the name of the target data type. It has one single argument whose type is the source data type; its return type is the target data type.

castout. The DB2 process of writing changed pages from a group buffer pool to DASD.

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. Coded character set identifier.

CDB. Communications database.

CDRA. Character data representation architecture.

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 sequence of bytes representing single-byte characters or a mixture of single- and double-byte characters where the size of the value can be up to 2 GB - 1. In general, character large object values 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- and double-byte characters.

CHECK clause. An extension to the SQL CREATE TABLE and SQL ALTER TABLE statements that specifies a table check constraint. See also *table check constraint*.

check constraint. See *table check constraint*.

check integrity. The condition that exists when each row in a table conforms to the table check constraints that are defined on that table. Maintaining check integrity requires DB2 to enforce table check constraints on operations that add or change data.

check pending. A state of a table space or partition that prevents its use by some utilities and some SQL statements because of rows that violate referential constraints, table check constraints, or both.

checkpoint. A point at which DB2 records internal status information on the DB2 log; the recovery process uses this information if DB2 abnormally terminates.

CICS. Represents (in this publication) one of the following products:

CICS Transaction Server for OS/390: Customer Information Control Center Transaction Server for OS/390

CICS/ESA: Customer Information Control System/Enterprise Systems Architecture

CICS/MVS: Customer Information Control System/Multiple Virtual Storage

CICS attachment facility. A DB2 subcomponent that uses the MVS subsystem interface (SSI) and cross storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

clause. In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

CLI. Call level interface.

client. See *requester*.

CLOB. Character large object.

clustering index. An index that determines how rows are physically ordered in a table space.

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.

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 consisting of a character set identifier and an associated code page identifier.

code page. A set of assignments of characters to code points.

code point. In CDRA, a unique bit pattern that represents a character in a code page.

collection. A group of packages that have the same qualifier.

column. The vertical component of a table. A column has a name and a particular data type (for example, character, decimal, or integer).

column function. An SQL operation that derives its result from a collection of values across one or more rows. Contrast with *scalar function*.

"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.

command. A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.

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.

commit point. A point in time when data is considered consistent.

committed phase. The second phase of the multi-site update process that requests all participants to commit the effects of the logical unit of work.

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 =, >, <) that is used to specify a relationship between two values.

composite key. An ordered set of key columns of the same table.

concurrency. The shared use of resources by more than one application process at the same time.

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).

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*, *table check constraint*, and *uniqueness constraint*.

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.

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 designates a table, a view, or individual rows of a table or view within a single SQL statement. It can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

cost category. A category into which DB2 places cost estimates for SQL statements at the time the statement is bound. A cost estimate can be placed in either of the following cost categories:

- A: Indicates that DB2 had enough information to make a cost estimate without using default values.
- B: Indicates that some condition exists for which DB2 was forced to use default values for its estimate.

The cost category is externalized in the COST_CATEGORY column of DSN_STATEMNT_TABLE when a statement is explained.

created temporary table. A 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, so this kind of table is persistent and can be shared across application processes. Contrast with *declared temporary table*. See also *temporary table*.

CS. Cursor stability.

current data. Data within a host structure that is current with (identical to) the data within the base table.

cursor. A named control structure that an application program uses to point to a row of interest within some set of rows, and to retrieve rows from the set, possibly making updates or deletions.

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.

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.

D

DASD. Direct access storage device.

database. A collection of tables, or a collection of table spaces and index spaces.

database access thread. 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 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, and relationships.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and the access to the data 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 2 Interactive (DB2I). The DB2 facility that provides for the execution of SQL statements, DB2 (operator) commands, programmer commands, and utility invocation.

data currency. The state in which data that is retrieved into a host variable in your program is a copy of data in the base table.

data sharing. The ability of two or more DB2 subsystems to directly access and change a single set of data.

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 type. An attribute of columns, literals, host variables, special registers, and the results of functions and expressions.

date. A three-part value that designates a day, month, and year.

date duration. A decimal integer that represents a number of years, months, and days.

datetime value. A value of the data type DATE, TIME, or TIMESTAMP.

DBA. Database administrator.

DBCLOB. Double-byte character large object.

DBCS. Double-byte character set.

DBD. Database descriptor.

DBID. Database identifier.

DBMS. Database management system.

DBRM. Database request module.

DB2 catalog. Tables that are maintained by DB2 and that contain descriptions of DB2 objects, such as tables, views, and indexes.

DB2 command. An instruction to the DB2 subsystem allowing a user 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.

DB2 for VSE & VM. The IBM DB2 relational database management system for the VSE and VM operating systems.

DB2I. DATABASE 2 Interactive.

DCLGEN. Declarations generator.

DDF. Distributed data facility.

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. DCLGEN is also a DSN subcommand.

declared temporary table. A 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

default value • double-byte character large object (DBCLOB)

DB2 catalog, so this kind of table is not persistent and
can only be used 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 is explicitly specified.

deferred embedded SQL. SQL statements that are neither fully static nor fully dynamic. Like static statements, they are embedded within an application, but like dynamic statements, they are prepared during the execution of the application.

delete-connected. A table that is a dependent of table P or a dependent of a table to which delete operations from table P cascade.

delete rule. The rule that tells DB2 what to do to a dependent row when a parent row is deleted. For each relationship, the rule might be CASCADE, RESTRICT, SET NULL, or NO ACTION.

delete trigger. A trigger that is defined with the triggering SQL operation DELETE.

delimited identifier. A sequence of characters that are enclosed within double quotation marks ("). The sequence must consist of a letter followed by zero or more characters, each of which is a letter, digit, or the underscore character (_).

delimiter token. A string constant, a delimited identifier, an operator symbol, or any of the special characters that are shown in syntax diagrams.

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 *parent row*, *parent table*, *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 this with an *not-deterministic function* (sometimes called a *variant function*), which might not always produce the same result for the same inputs.

dimension. A data category such as time, products, or markets. The elements of a dimension are referred to as members. Dimensions offer a very concise, intuitive way of organizing and selecting data for retrieval, exploration, and analysis. 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*.

direct access storage device (DASD). A device in which access time is independent of the location of the data.

directory. The DB2 system database that contains internal objects such as database descriptors and skeleton cursor tables.

distinct type. A user-defined data type that is internally represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

distributed data facility (DDF). A set of DB2 components through which DB2 communicates with another RDBMS.

Distributed Relational Database Architecture (DRDA). A connection protocol for distributed relational database processing that is used by IBM's 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.

DNS. Domain name server.

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 sequence of bytes representing double-byte characters where the size of the values can be up to 2 GB. In general, double-byte character large object 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 and therefore requires special hardware to be displayed or printed. Contrast with *single-byte character set*.

double-precision floating point number. A 64-bit approximate representation of a real number.

DRDA. Distributed Relational Database Architecture.

DRDA access. A method of accessing distributed data by which you can connect to another location, using an SQL statement, to execute packages that have been previously bound at that location. The SQL CONNECT or three-part name statement is used to identify application servers, and SQL statements are executed using packages that were previously bound at those servers. Contrast with *private protocol access*.

DSN. (1) The default DB2 subsystem name. (2) The name of the TSO command processor of DB2. (3) The first three characters of DB2 module and macro names.

duration. A number that represents an interval of time. See *date duration*, *labeled duration*, and *time duration*.

dynamic SQL. SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

E

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.

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the OS/390, MVS, VM, VSE, and OS/400® environments. Contrast with *ASCII*.

EDM pool. A pool of main storage that is used for database descriptors, application plans, authorization cache, application packages, and dynamic statement caching.

embedded SQL. SQL statements that are coded within an application program. See *static SQL*.

encoding scheme. A set of rules to represent character data (ASCII or EBCDIC).

equi-join. A join operation in which the join-condition has the form *expression = expression*.

escape character. The symbol that is used to enclose an SQL delimited identifier. The escape character is the double quotation mark ("), except in COBOL applications, where the user assigns the symbol, which is either a double quotation mark or an apostrophe (').

EUR. IBM European Standards.

exclusive lock. A lock that prevents concurrently executing application processes from reading or changing data. Contrast with *shared lock*.

executable statement. An SQL statement that can be embedded in an application program, dynamically prepared and executed, or issued interactively.

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.

exposed name. A correlation name or a table or view name for which a correlation name is not specified. Names specified in a FROM clause are exposed or non-exposed.

expression. An operand or a collection of operators and operands that yields a single value.

external function. A function for which the body is written in a programming language that takes scalar argument values and produces a scalar result for each invocation. Contrast with *sourced function* and *built-in function*.

external routine. A user-defined function or stored procedure that is based on code that is written in an external programming language.

F

failed member state. A state of a member of a data sharing group. When a member fails, the XCF permanently records the failed member state. This state usually means that the member's task, address space, or MVS system terminated before the state changed from active to quiesced.

fallback. The process of returning to a previous release of DB2 after attempting or completing migration to a current release.

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.

filter factor • host variable

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 or graphic string whose length is specified and cannot be changed. Contrast with *varying-length string*.

foreign key. A key that is specified in the definition of a referential constraint. Because of the foreign key, the table is a dependent table. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table.

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*.

function. A specific purpose of an entity or its characteristic action such as a column function or scalar function. (See also *column function* and *scalar function*.)

Functions can be user-defined, built-in, or generated by DB2. (See *built-in function*, *cast function*, *external function*, *sourced 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 implementer. The authorization ID of the owner of the function program and function package.

function package. A package that results from binding the DBRM for a function program.

function resolution. The process, internal to the DBMS, by which a function invocation is bound to a particular function instance. This process uses the function name, the data types of the arguments, and a list of the applicable schema names (called the *SQL path*) to make the selection. This process is sometimes called *function selection*.

function selection. See *function resolution*.

function signature. The logical concatenation of a fully qualified function name with the data types of all of its parameters.

G

GB. Gigabyte (1 073 741 824 bytes).

GBP. 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 DASD.

graphic string. A sequence of DBCS characters.

gross lock. The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

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 name. The MVS 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.

H

host. The set of programs and resources that are available on a given TCP/IP instance.

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, an application variable that is referenced by embedded SQL statements.

I

ICF. Integrated catalog facility.

identity column. A column that provides a way for DB2 to automatically generate a guaranteed-unique numeric value for each row that is inserted into the table. Identity columns are defined with the AS IDENTITY clause. A table can have no more than one identity column.

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. Information Management System.

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 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 emergency 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.

inner join. The result of a join operation that includes only the matched rows of both tables being joined. See also *join*.

inoperative package. A package that cannot be used because one or more user-defined functions that the package depends on were dropped. Such a package must be explicitly rebound. Contrast with *invalid package*.

insert trigger. A trigger that is defined with the triggering SQL operation INSERT.

internal resource lock manager (IRLM). An MVS subsystem that DB2 uses to control communication and database locking.

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.

invalid package. 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*.

invariant character set. (1) A character set, such as the syntactic character set, whose code point assignments do not change from code page to code page. (2) A minimum set of characters that is available as part of all character sets.

IP address. A 4-byte value that uniquely identifies a TCP/IP host.

IRLM. Internal resource lock manager.

ISO. International Standards Organization.

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*.

J

Japanese Industrial Standards Committee (JISC). An organization that issues standards for coding character sets.

JIS. Japanese Industrial Standard.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values. See also *equi-join*, *full outer join*, *inner join*, *left outer join*, *outer join*, and *right outer join*.

K

KB. Kilobyte (1024 bytes).

key. A column or an ordered collection of columns identified in the description of a table, index, or referential constraint.

keyword. In SQL, a name that identifies an option used in an SQL statement.

L

labeled duration. A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

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 - 1 byte in length. See also *BLOB*, *CLOB*, and *DBCLOB*.

leaf page. A 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*.

L-lock. Logical lock.

LOB. 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 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 characters that are defined for a specific language and country, and a CCSID.

local subsystem. The unique RDBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

location name. The name by which DB2 refers to a particular DB2 subsystem in a network of subsystems. Contrast with *LU name*.

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.

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 level.

lock size. The amount of data 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.

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.

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 *P-lock*.

logical unit. An access point through which an application program accesses the SNA network in order to communicate with another application program.

logical unit of work (LUW). The processing that a program performs between synchronization points.

log initialization. The first phase of restart processing during which DB2 attempts to locate the current end of the log.

log record sequence number (LRSN). A number that DB2 generates and associates with each log record. DB2 also uses the LRSN for page versioning. The LRSNs that a particular DB2 data sharing group

generates from a strictly increasing sequence for each DB2 log and a strictly increasing sequence for each page across the DB2 group.

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.

long string. A string whose actual length, or a varying-length string whose maximum length, is greater than 255 bytes or 127 double-byte characters. Any LOB column, LOB host variable, or expression that evaluates to a LOB is considered a long string.

LRSN. Log record sequence number.

LU name. Logical unit name, which is the name by which VTAM refers to a node in a network. Contrast with *location name*.

LUW. Logical unit of work.

M

member name. The MVS XCF identifier for a particular DB2 subsystem in a data sharing group.

migration. The process of converting a DB2 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 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.

multi-site update. Distributed relational database processing in which data is updated in more than one location within a single unit of work.

MVS. Multiple Virtual Storage.

MVS/ESA. Multiple Virtual Storage/Enterprise Systems Architecture.

MVS/XA. Multiple Virtual Storage/Extended Architecture.

N

nested table expression. A subselect in a FROM clause (surrounded by parentheses).

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.

nonpartitioning index. Any index that is not a partitioning index.

not-deterministic 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 this with a *deterministic function* (sometimes called a *not-variant function*), which always produces the same result for the same inputs.

not-variant function. See *deterministic function*.

NUL. In C, a single character that denotes the end of the string.

null. A special value that indicates the absence of information.

NULLIF. A scalar function that evaluates two passed expressions, returning either NULL if the arguments are equal or the value of the first argument if they are not.

NUL-terminated host variable. A varying-length host variable in which the end of the data is indicated by the presence of a NUL terminator.

NUL terminator. In C, the value that indicates the end of a string. For character strings, the NUL terminator is X'00'.

O

ODBC. Open Database Connectivity.

OBID. Data object identifier.

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 • predicate

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.

OS/390. Operating System/390.

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*.

overloaded function. A function name for which multiple function instances exist.

P

package. An object containing a set of SQL statements that have been bound statically 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 created by a BIND PACKAGE or REBIND PACKAGE command. The object is a bound version of a database request module (DBRM). The name 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). In a table space, a page contains one or more rows of a table. In a LOB table space, a LOB value can span more than one page, but no more than one LOB 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.

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.

parameter marker. A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable could appear if the statement string were a static SQL statement.

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 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.

partition. A portion of a page set. Each partition corresponds to a single, independently extendable data set. Partitions can be extended to a maximum size of 1, 2, or 4 GB, depending on the number of partitions in the partitioned page set. All partitions of a given page set have the same maximum size.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, which are called members. Each partition can contain a program, part of a program, or data. The term partitioned data set is synonymous with program library.

partitioned table space. A table space that is subdivided into parts (based on index key range), each of which can be processed independently by utilities.

path. See *SQL path*.

PDS. Partitioned data set.

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 to execute it.

plan name. The name of an application plan.

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*.

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.

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 used to identify the application process to DB2.

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.

private connection. A communications connection that is specific to DB2.

private protocol access. A method of accessing distributed data by which you can direct a query to another DB2 system. Contrast with *DRDA access*.

private protocol connection. A DB2 private connection of the application process. See also *private connection*.

privilege. The capability of performing a specific function, sometimes on a specific object. The term includes:

explicit privileges, which have names and are held as the result of SQL GRANT and REVOKE statements. For example, the SELECT privilege.

implicit privileges, which accompany the ownership of an object, such as the privilege to drop a synonym one owns, or the holding of an authority, such as the privilege of SYSADM authority to terminate any utility job.

privilege set. For the installation SYSADM ID, the set of all possible privileges. For any other 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.

protected conversation. A VTAM conversation that supports two-phase commit flows.

Q

QMF. Query Management Facility.

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's internal processing of the query.

quiesced member state. A state of a member of a data sharing group. An active member becomes quiesced when a STOP DB2 command takes effect without a failure. If the member's task, address space, or MVS system fails before the command takes effect, the member state is failed.

R

RACF. Resource Access Control Facility.

RDB. Relational database.

RDBMS. Relational database management system.

RDBNAM. Relational database name.

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. Under level RS, an application that issues the same query more than once might read additional rows that were inserted and committed by a concurrently executing application process.

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.

record. The storage representation of a row or other data.

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.

referential constraint • RS

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 integrity. The condition that exists when all intended references from data in one column of a table to data in another column of the same or a different table are valid. Maintaining referential integrity requires that DB2 enforce referential constraints on all LOAD, RECOVER, INSERT, UPDATE, and DELETE operations.

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.

relational database (RDB). 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 database name (RDBNAM). A unique identifier for an RDBMS within a network. In DB2, this must be the value in the LOCATION column of table SYSIBM.LOCATIONS in the CDB. DB2 publications refer to the name of another RDBMS as a LOCATION value or a location name.

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.

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 RDBMS, 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 MVS 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.

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 referenced by the

program cannot be changed by other programs until the program reaches a commit point.

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 a remote RDBMS, the system that requests the data. A requester is sometimes called an *application requester (AR)*.

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 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.

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.

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*.

RLF. Resource limit facility.

rollback. The process of restoring data changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

routine. A term that refers to either a user-defined function or a stored procedure.

row. The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

ROWID. Row identifier.

row identifier (ROWID). A value that uniquely identifies a row. This value is stored with the row and never changes.

row trigger. A trigger that is defined with the trigger granularity FOR EACH ROW.

RS. Read stability.

S

savepoint. A named entity that represents the state of
data and schemas at a particular point in time within a
unit of work. SQL statements exist to set a savepoint,
release a savepoint, and restore data and schemas to
the state that the savepoint represents. The restoration
of data and schemas to a savepoint is usually referred
to as *rolling back to a savepoint*.

SBCS. Single-byte character set.

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. Contrast with *column function*.

schema. A logical grouping for user-defined functions, distinct types, triggers, and stored procedures. When an object of one of these types is created, it is assigned to one schema, which is determined by the name of the object. For example, the following statement creates a distinct type T in schema C:

```
CREATE DISTINCT TYPE C.T ...
```

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.

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.

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.

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.

server. A functional unit that provides services to one or more clients over a network. In the DB2 environment, a server is the target for a request from a remote RDBMS and is the RDBMS that provides the data. A server is sometimes also called an *application server (AS)*.

shared 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*.

short string. A string whose actual length, or a varying-length string whose maximum length, is 255 bytes (or 127 double-byte characters) or less. Regardless of length, a LOB string is not a short string.

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 table space. A table space that is neither partitioned nor segmented.

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*.

single-precision floating point number. A 32-bit approximate representation of a real number.

SMF. System management facility.

SMS. Storage Management Subsystem.

socket. A callable TCP/IP programming interface that is used by TCP/IP network applications 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 a column (aggregating) function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with *external function* and *built-in function*.

source program. A set of host language statements and SQL statements that is processed by an SQL precompiler.

source type. An existing type that is used to internally 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 USER and CURRENT DATE.

specific function name • subquery

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. SQL Processor Using File Input.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQL communication area (SQLCA). A structure that is used to provide an application program with information about the execution of its SQL statements.

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 ID. SQL authorization ID.

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 current path is found in the CURRENT PATH special register. In static SQL, it is defined in the PATH bind option.

SQL Processor Using File Input (SPUFI). SQL Processor Using File Input. 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 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 (").

SQLCA. SQL communication area.

SQLDA. SQL descriptor area.

SQL/DS. Structured Query Language/Data System. This product is now obsolete and has been replaced by DB2 for VSE & VM.

SSI. Subsystem interface (MVS).

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 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 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 host variables that are specified by the statement might change).

storage group. A named set of DASD volumes on which DB2 data can be stored.

stored procedure. A user-written application program, that can be invoked through the use of the SQL CALL statement.

string. See *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 US dollars. But you can provide a user-defined function to convert one currency to the other and then do the comparison.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

subject table. The table for which a trigger is created. When the defined triggering event occurs on this table, the trigger is activated.

subpage. The unit into which a physical index page can be divided.

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 does not include ORDER BY clause, UPDATE clause, or UNION operators.

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. A distinct instance of a relational database management system (RDBMS).

sync point. See *commit point*.

synonym. In SQL, an alternative name for a table or view. Synonyms can only be used to refer to objects at the subsystem in which the synonym is defined.

syntactic character set. A set of 81 graphic characters that are registered in the IBM registry as character set 00640. This set was originally recommended to the programming language community to be used for syntactic purposes toward maximizing portability and interchangeability across systems and country boundaries. It is contained in most of the primary registered character sets, with a few exceptions. See also *invariant character set*.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

system conversation. The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

T

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*.

table check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain.

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 only be referenced in the FROM clause of a subselect.

table locator. A mechanism that allows access to trigger transition tables in the FROM clause of SELECT statements, the subselect of INSERT statements, 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.

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.

temporary table. A table that holds temporary data; # for example, temporary tables are useful for holding or # sorting intermediate results from queries that contain a # large number of rows. The two kinds 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*.

thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services. Most DB2 functions execute under a thread structure. See also *allied thread* and *database access thread*.

three-part name. The full name of a table, view, or alias. It consists of a location name, authorization ID, and an object name, separated by a period.

time. A three-part value that designates a time of day in hours, minutes, and seconds.

time duration. A decimal integer that represents a number of hours, minutes, and seconds.

Time-Sharing Option (TSO). An option in MVS 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.

trace. A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

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.

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.

trigger • update trigger

trigger. A set of SQL statements that are stored in a DB2 database and executed when a certain event occurs in a DB2 table.

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.

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. 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

trigger package. A package that is created when a CREATE TRIGGER statement is executed. The package is executed when the trigger is activated.

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 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.

TSO. Time-Sharing Option.

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 1 indexes. Indexes that were created by a release of DB2 before DB2 Version 4 or that are specified as type 1 indexes in Version 4. Contrast with *type 2 indexes*. As of Version 6, type 1 indexes are no longer supported.

type 2 indexes. Indexes that are created on a release of DB2 after Version 5 or that are specified as type 2 indexes in Version 4 or Version 5.

U

UDF. User-defined function.

UDT. User-defined data type. In DB2 for OS/390, the term *distinct type* is used instead of user-defined function.

uncommitted read (UR). The isolation level that allows an application to read uncommitted data.

underlying view. The view on which another view is directly or indirectly defined.

UNION. An SQL operation that combines the results of two select statements. UNION is often used to merge lists of values that are obtained from several tables.

unique index. An index which ensures that no identical key values are stored in a table.

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.

unit of recovery. A recoverable sequence of operations within a single resource manager, such as an instance of DB2. Contrast with *unit of work*.

unit of work. 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 *multi-site update* operation, a single unit of work can include several *units of recovery*. Contrast with *unit of recovery*.

untyped parameter marker. A parameter marker that is specified without its target data type. It has the form of a single question mark (?).

update trigger. A trigger that is defined with the triggering SQL operation UPDATE.

UR. Uncommitted read.

user-defined data type (UDT). See *distinct type*.

user-defined function (UDF). A function that is defined to DB2 using the CREATE FUNCTION statement and that can be referenced thereafter in SQL statements. A user-defined function can be either an *external function* or a *sourced function*. Contrast with *built-in function*.

UT. Utility-only access.

V

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 variable. Contrast with *constant*.

variant function. See *not-deterministic function*.

varying-length string. A character or graphic 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 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 a package is the result of binding a DBRM within a particular database system. The 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.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns that are contained in tables on which it is defined.

view check option. An option that specifies whether every row that is inserted or updated through a view must conform to the definition of that view. A view check option can be specified with the WITH CASCADED CHECK OPTION, WITH CHECK OPTION, or WITH LOCAL CHECK OPTION clauses of the CREATE VIEW statement.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed- and varying-length records on direct access devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number.

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network.

VSAM. Virtual storage access method.

VTAM. Virtual Telecommunication Access Method (MVS).

Bibliography

DB2 Universal Database Server for OS/390 Version 6 Product Libraries:

DB2 Universal Database for OS/390

- *DB2 Administration Guide*, SC26-9003
- *DB2 Application Programming and SQL Guide*, SC26-9004
- *DB2 Application Programming Guide and Reference for Java™*, SC26-9018
- *DB2 ODBC Guide and Reference*, SC26-9005
- *DB2 Command Reference*, SC26-9006
- *DB2 Data Sharing: Planning and Administration*, SC26-9007
- *DB2 Data Sharing Quick Reference Card*, SX26-3843
- *DB2 Diagnosis Guide and Reference*, LY36-3736
- *DB2 Diagnostic Quick Reference Card*, LY36-3737
- *DB2 Image, Audio, and Video Extenders Administration and Programming*, SC26-9650
- *DB2 Installation Guide*, GC26-9008
- *DB2 Licensed Program Specifications*, GC26-9009
- *DB2 Messages and Codes*, GC26-9011
- *DB2 Master Index*, SC26-9010
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC26-9012
- *DB2 Reference Summary*, SX26-3844
- *DB2 Release Planning Guide*, SC26-9013
- *DB2 SQL Reference*, SC26-9014
- *DB2 Text Extender Administration and Programming*, SC26-9651
- *DB2 Utility Guide and Reference*, SC26-9015
- *DB2 What's New?* GC26-9017
- *DB2 Program Directory*, GI10-8182

DB2 Administration Tool

- *DB2 Administration Tool for OS/390 User's Guide*, SC26-9847

DB2 Buffer Pool Tool

- *DB2 Buffer Pool Tool for OS/390 User's Guide and Reference*, SC26-9306

DB2 Data Propagator

- *DB2 Replication Guide and Reference*, SC26-9642

Net.Data for OS/390

The following books are available at
<http://www.ibm.com/software/net.data/library.html>:

- *Net.Data Library: Administration and Programming Guide for OS/390*
- *Net.Data Library: Language Environment Interface Reference*
- *Net.Data Library: Messages and Codes*
- *Net.Data Library: Reference*

DB2 PM for OS/390

- *DB2 PM for OS/390 Batch User's Guide*, SC26-9167
- *DB2 PM for OS/390 Command Reference*, SC26-9166
- *DB2 PM for OS/390 General Information*, GC26-9172
- *DB2 PM for OS/390 Installation and Customization*, SC26-9171
- *DB2 PM for OS/390 Messages*, SC26-9169
- *DB2 PM for OS/390 Online Monitor User's Guide*, SC26-9168
- *DB2 PM for OS/390 Report Reference Volume 1*, SC26-9164
- *DB2 PM for OS/390 Report Reference Volume 2*, SC26-9165
- *DB2 PM for OS/390 Using the Workstation Online Monitor*, SC26-9170
- *DB2 PM for OS/390 Program Directory*, GI10-8183

Query Management Facility

- *Query Management Facility: Developing QMF Applications*, SC26-9579
- *Query Management Facility: Getting Started with QMF on Windows*, SC26-9582
- *Query Management Facility: High Performance Option User's Guide for OS/390*, SC26-9581
- *Query Management Facility: Installing and Managing QMF on OS/390*, GC26-9575
- *Query Management Facility: Installing and Managing QMF on Windows*, GC26-9583
- *Query Management Facility: Introducing QMF*, GC26-9576
- *Query Management Facility: Messages and Codes*, GC26-9580
- *Query Management Facility: Reference*, SC26-9577
- *Query Management Facility: Using QMF*, SC26-9578

Ada/370

- *IBM Ada/370 Language Reference, SC09-1297*
- *IBM Ada/370 Programmer's Guide, SC09-1414*
- *IBM Ada/370 SQL Module Processor for DB2 Database Manager User's Guide, SC09-1450*

APL2

- *APL2 Programming Guide, SH21-1072*
- *APL2 Programming: Language Reference, SH21-1061*
- *APL2 Programming: Using Structured Query Language (SQL), SH21-1057*

AS/400

- *DB2 for OS/400 SQL Programming, SC41-4611*
- *DB2 for OS/400 SQL Reference, SC41-4612*

BASIC

- *IBM BASIC/MVS Language Reference, GC26-4026*
- *IBM BASIC/MVS Programming Guide, SC26-4027*

BookManager READ/MVS

- *BookManager READ/MVS V1R3: Installation Planning & Customization, SC38-2035*

C/370

- *IBM SAA AD/Cycle C/370 Programming Guide, SC09-1841*
- *IBM SAA AD/Cycle C/370 Programming Guide for Language Environment/370, SC09-1840*
- *IBM SAA AD/Cycle C/370 User's Guide, SC09-1763*
- *SAA CPI C Reference, SC09-1308*

Character Data Representation Architecture

- *Character Data Representation Architecture Overview, GC09-2207*
- *Character Data Representation Architecture Reference and Registry, SC09-2190*

CICS/ESA

- *CICS/ESA Application Programming Guide, SC33-1169*
- *CICS for MVS/ESA Application Programming Reference, SC33-1170*
- *CICS for MVS/ESA CICS-RACF Security Guide, SC33-1185*
- *CICS for MVS/ESA CICS-Supplied Transactions, SC33-1168*
- *CICS for MVS/ESA Customization Guide, SC33-1165*
- *CICS for MVS/ESA Data Areas, LY33-6083*
- *CICS for MVS/ESA Installation Guide, SC33-1163*
- *CICS for MVS/ESA Intercommunication Guide, SC33-1181*

- *CICS for MVS/ESA Messages and Codes, GC33-1177*
- *CICS for MVS/ESA Operations and Utilities Guide, SC33-1167*
- *CICS/ESA Performance Guide, SC33-1183*
- *CICS/ESA Problem Determination Guide, SC33-1176*
- *CICS for MVS/ESA Resource Definition Guide, SC33-1166*
- *CICS for MVS/ESA System Definition Guide, SC33-1164*
- *CICS for MVS/ESA System Programming Reference, GC33-1171*

CICS/MVS

- *CICS/MVS Application Programmer's Reference, SC33-0512*
- *CICS/MVS Facilities and Planning Guide, SC33-0504*
- *CICS/MVS Installation Guide, SC33-0506*
- *CICS/MVS Operations Guide, SC33-0510*
- *CICS/MVS Problem Determination Guide, SC33-0516*
- *CICS/MVS Resource Definition (Macro), SC33-0509*
- *CICS/MVS Resource Definition (Online), SC33-0508*

IBM C/C++ for MVS/ESA

- *IBM C/C++ for MVS/ESA Library Reference, SC09-1995*
- *IBM C/C++ for MVS/ESA Programming Guide, SC09-1994*

IBM COBOL

- *IBM COBOL Language Reference, SC26-4769*
- *IBM COBOL for MVS & VM Programming Guide, SC26-4767*

Conversion Guide

- *IMS-DB and DB2 Migration and Coexistence Guide, GH21-1083*

Cooperative Development Environment

- *CoOperative Development Environment/370: Debug Tool, SC09-1623*

Data Extract (DXT)

- *Data Extract Version 2: General Information, GC26-4666*
- *Data Extract Version 2: Planning and Administration Guide, SC26-4631*

DataPropagator NonRelational

- *DataPropagator NonRelational MVS/ESA Administration Guide, SH19-5036*
- *DataPropagator NonRelational MVS/ESA Reference, SH19-5039*

Data Facility Data Set Services

- *Data Facility Data Set Services: User's Guide and Reference*, SC26-4388

Database Design

- *DB2 Design and Development Guide*, Gabrielle Wiorkowski and David Kull, Addison Wesley, ISBN 0-20158-049-8
- *Handbook of Relational Database Design*, C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8

DataHub

- *IBM DataHub General Information*, GC26-4874

DB2 Connect

- *DB2 Connect Enterprise Edition for OS/2 and Windows NT: Quick Beginnings*, GC09-2828
- *DB2 Connect Personal Edition Quick Beginnings*, GC09-2830
- *DB2 Connect User's Guide*, SC09-2838

DB2 Server for VSE & VM

- *DB2 Server for VM: DBS Utility*, SC09-2394
- *DB2 Server for VSE: DBS Utility*, SC09-2395

DB2 Universal Database (DB2 UDB)

- *DB2 UDB Administration Guide Volume 1: Design and Implementation*, SC09-2839
- *DB2 UDB Administration Guide Volume 2: Performance*, SC09-2840
- *DB2 UDB Administrative API Reference*, SC09-2841
- *DB2 UDB Application Building Guide*, SC09-2842
- *DB2 UDB Application Development Guide*, SC09-2845
- *DB2 UDB Call Level Interface Guide and Reference*, SC09-2843
- *DB2 UDB SQL Getting Started*, SC09-2856
- *DB2 UDB SQL Reference Volume 1*, SC09-2847
- *DB2 UDB SQL Reference Volume 2*, SC09-2848

Device Support Facilities

- *Device Support Facilities User's Guide and Reference*, GC35-0033

DFSMS/MVS

- *DFSMS/MVS: Access Method Services for the Integrated Catalog*, SC26-4906
- *DFSMS/MVS: Access Method Services for VSAM Catalogs*, SC26-4905
- *DFSMS/MVS: Administration Reference for DFSMSdss*, SC26-4929
- *DFSMS/MVS: DFSMSHsm Managing Your Own Data*, SH21-1077
- *DFSMS/MVS: Diagnosis Reference for DFSMSdfp*, LY27-9606

- *DFSMS/MVS Storage Management Library: Implementing System-Managed Storage*, SC26-3123
- *DFSMS/MVS: Macro Instructions for Data Sets*, SC26-4913
- *DFSMS/MVS: Managing Catalogs*, SC26-4914
- *DFSMS/MVS: Program Management*, SC26-4916
- *DFSMS/MVS: Storage Administration Reference for DFSMSdfp*, SC26-4920
- *DFSMS/MVS: Using Advanced Services*, SC26-4921
- *DFSMS/MVS: Utilities*, SC26-4926
- *MVS/DFP: Using Data Sets*, SC26-4749

DFSORT

- *DFSORT Application Programming: Guide*, SC33-4035

Distributed Relational Database

- *Data Stream and OPA Reference*, SC31-6806
- *IBM SQL Reference*, SC26-8416
- *Open Group Technical Standard (the Open Group presently makes the following books available through its Web site at <http://www.opengroup.org>):*
 - *DRDA Volume 1: Distributed Relational Database Architecture (DRDA)*, ISBN 1-85912-295-7
 - # – *DRDA Version 2 Volume 2: Formatted Data Object Content Architecture*, available only on Web
 - #
 - #
 - *DRDA Volume 3: Distributed Database Management (DDM) Architecture*, ISBN 1-85912-206-X

Domain Name System

- *DNS and BIND, Third Edition*, Paul Albitz and Cricket Liu, O'Reilly, SR23-8771

Education

- *IBM Dictionary of Computing*, McGraw-Hill, ISBN 0-07031-489-6
- *1999 IBM All-in-One Education and Training Catalog*, GR23-8105

Enterprise System/9000 and Enterprise System/3090

- *Enterprise System/9000 and Enterprise System/3090 Processor Resource/System Manager Planning Guide*, GA22-7123

High Level Assembler

- *High Level Assembler for MVS and VM and VSE Language Reference*, SC26-4940
- *High Level Assembler for MVS and VM and VSE Programmer's Guide*, SC26-4941

Parallel Sysplex Library

- *OS/390 Parallel Sysplex Application Migration*, GC28-1863
- *System/390 MVS Sysplex Hardware and Software Migration*, GC28-1862
- *OS/390 Parallel Sysplex Overview: An Introduction to Data Sharing and Parallelism*, GC28-1860
- *OS/390 Parallel Sysplex Systems Management*, GC28-1861
- *OS/390 Parallel Sysplex Test Report*, GC28-1963
- *System/390 9672/9674 System Overview*, GA22-7148

ICSF/MVS

- *ICSF/MVS General Information*, GC23-0093

IMS/ESA

- *IMS Batch Terminal Simulator General Information*, GH20-5522
- *IMS/ESA Administration Guide: System*, SC26-8013
- *IMS/ESA Administration Guide: Transaction Manager*, SC26-8731
- *IMS/ESA Application Programming: Database Manager*, SC26-8727
- *IMS/ESA Application Programming: Design Guide*, SC26-8016
- *IMS/ESA Application Programming: Transaction Manager*, SC26-8729
- *IMS/ESA Customization Guide*, SC26-8020
- *IMS/ESA Installation Volume 1: Installation and Verification*, SC26-8023
- *IMS/ESA Installation Volume 2: System Definition and Tailoring*, SC26-8024
- *IMS/ESA Messages and Codes*, SC26-8028
- *IMS/ESA Operator's Reference*, SC26-8030
- *IMS/ESA Utilities Reference: System*, SC26-8035

ISPF

- *ISPF V4 Dialog Developer's Guide and Reference*, SC34-4486
- *ISPF V4 Messages and Codes*, SC34-4450
- *ISPF V4 Planning and Customizing*, SC34-4443
- *ISPF V4 User's Guide*, SC34-4484

Language Environment

- *Debug Tool User's Guide and Reference*, SC09-2137

National Language Support

- *IBM National Language Support Reference Volume 2*, SE09-8002

NetView

- *NetView Installation and Administration Guide*, SC31-8043
- *NetView User's Guide*, SC31-8056

ODBC

- *Microsoft ODBC 3.0 Programmer's Reference and SDK Guide*, Microsoft Press, ISBN 1-55615-658-8

OS/390

- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 eNetwork Communications Server: IP Configuration*, SC31-8513
- *OS/390 Hardware Configuration Definition Planning*, GC28-1750
- *OS/390 Information Roadmap*, GC28-1727
- *OS/390 Introduction and Release Guide*, GC28-1725
- *OS/390 JES2 Initialization and Tuning Guide*, SC28-1791
- *OS/390 JES3 Initialization and Tuning Guide*, SC28-1802
- *OS/390 Language Environment for OS/390 & VM Concepts Guide*, GC28-1945
- *OS/390 Language Environment for OS/390 & VM Customization*, SC28-1941
- *OS/390 Language Environment for OS/390 & VM Debugging Guide*, SC28-1942
- *OS/390 Language Environment for OS/390 & VM Programming Guide*, SC28-1939
- *OS/390 Language Environment for OS/390 & VM Programming Reference*, SC28-1940
- *OS/390 MVS Diagnosis: Procedures*, LY28-1082
- *OS/390 MVS Diagnosis: Reference*, SY28-1084
- *OS/390 MVS Diagnosis: Tools and Service Aids*, LY28-1085
- *OS/390 MVS Initialization and Tuning Guide*, SC28-1751
- *OS/390 MVS Initialization and Tuning Reference*, SC28-1752
- *OS/390 MVS Installation Exits*, SC28-1753
- *OS/390 MVS JCL Reference*, GC28-1757
- *OS/390 MVS JCL User's Guide*, GC28-1758
- *OS/390 MVS Planning: Global Resource Serialization*, GC28-1759
- *OS/390 MVS Planning: Operations*, GC28-1760
- *OS/390 MVS Planning: Workload Management*, GC28-1761
- *OS/390 MVS Programming: Assembler Services Guide*, GC28-1762
- *OS/390 MVS Programming: Assembler Services Reference*, GC28-1910
- *OS/390 MVS Programming: Authorized Assembler Services Guide*, GC28-1763
- *OS/390 MVS Programming: Authorized Assembler Services Reference, Volumes 1-4*, GC28-1764, GC28-1765, GC28-1766, GC28-1767
- *OS/390 MVS Programming: Callable Services for High-Level Languages*, GC28-1768
- *OS/390 MVS Programming: Extended Addressability Guide*, GC28-1769

- OS/390 MVS Programming: Sysplex Services Guide, GC28-1771
- OS/390 MVS Programming: Sysplex Services Reference, GC28-1772
- OS/390 MVS Programming: Workload Management Services, GC28-1773
- OS/390 MVS Routing and Descriptor Codes, GC28-1778
- OS/390 MVS Setting Up a Sysplex, GC28-1779
- OS/390 MVS System Codes, GC28-1780
- OS/390 MVS System Commands, GC28-1781
- OS/390 MVS System Messages Volume 1, GC28-1784
- OS/390 MVS System Messages Volume 2, GC28-1785
- OS/390 MVS System Messages Volume 3, GC28-1786
- OS/390 MVS System Messages Volume 4, GC28-1787
- OS/390 MVS System Messages Volume 5, GC28-1788
- OS/390 MVS Using the Subsystem Interface, SC28-1789
- OS/390 Security Server (RACF) Auditor's Guide, SC28-1916
- OS/390 Security Server (RACF) Command Language Reference, SC28-1919
- OS/390 Security Server (RACF) General User's Guide, SC28-1917
- OS/390 Security Server (RACF) Introduction, GC28-1912
- OS/390 Security Server (RACF) Macros and Interfaces, SK2T-6700 (OS/390 Collection Kit), SK27-2180 (OS/390 Security Server Information Package)
- OS/390 Security Server (RACF) Security Administrator's Guide, SC28-1915
- OS/390 Security Server (RACF) System Programmer's Guide, SC28-1913
- OS/390 SMP/E Reference, SC28-1806
- OS/390 SMP/E User's Guide, SC28-1740
- OS/390 RMF User's Guide, SC28-1949
- OS/390 TSO/E CLISTS, SC28-1973
- OS/390 TSO/E Command Reference, SC28-1969
- OS/390 TSO/E Customization, SC28-1965
- OS/390 TSO/E Messages, GC28-1978
- OS/390 TSO/E Programming Guide, SC28-1970
- OS/390 TSO/E Programming Services, SC28-1971
- OS/390 TSO/E REXX Reference, SC28-1975
- OS/390 TSO/E User's Guide, SC28-1968
- OS/390 DCE Administration Guide, SC28-1584
- OS/390 DCE Introduction, GC28-1581
- OS/390 DCE Messages and Codes, SC28-1591
- OS/390 UNIX System Services Command Reference, SC28-1892
- OS/390 UNIX System Services Messages and Codes, SC28-1908
- OS/390 UNIX System Services Planning, SC28-1890

- OS/390 UNIX System Services User's Guide, SC28-1891
- OS/390 UNIX System Services Programming: Assembler Callable Services Reference, SC28-1899

j

PL/I for MVS & VM

- IBM PL/I MVS & VM Language Reference, SC26-3114
- IBM PL/I MVS & VM Programming Guide, SC26-3113

OS PL/I

- OS PL/I Programming Language Reference, SC26-4308
- OS PL/I Programming Guide, SC26-4307

Prolog

- IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide, SH19-6892

RAMAC

- # • IBM RAMAC Virtual Array, SG24-4951
- # • RAMAC Virtual Array: Implementing Peer-to-Peer Remote Copy, SG24-5338
- # • Enterprise Storage Server Introduction and Planning, GC26-7294

Remote Recovery Data Facility

- Remote Recovery Data Facility Program Description and Operations, LY37-3710

Storage Management

- DFSMS/MVS Storage Management Library: Implementing System-Managed Storage, SC26-3123
- MVS/ESA Storage Management Library: Leading a Storage Administration Group, SC26-3126
- MVS/ESA Storage Management Library: Managing Data, SC26-3124
- MVS/ESA Storage Management Library: Managing Storage Groups, SC26-3125
- MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide, SC26-4659

System/370 and System/390

- ESA/370 Principles of Operation, SA22-7200
- ESA/390 Principles of Operation, SA22-7201
- System/390 MVS Sysplex Hardware and Software Migration, GC28-1210

System Network Architecture (SNA)

- SNA Formats, GA27-3136
- SNA LU 6.2 Peer Protocols Reference, SC31-6808

- *SNA Transaction Programmer's Reference Manual for LU Type 6.2, GC30-3084*
- *SNA/Management Services Alert Implementation Guide, GC31-6809*

TCP/IP

- *IBM TCP/IP for MVS: Customization & Administration Guide, SC31-7134*
- *IBM TCP/IP for MVS: Diagnosis Guide, LY43-0105*
- *IBM TCP/IP for MVS: Messages and Codes, SC31-7132*
- *IBM TCP/IP for MVS: Planning and Migration Guide, SC31-7189*

VS COBOL II

- *VS COBOL II Application Programming Guide for MVS and CMS, SC26-4045*
- *VS COBOL II Application Programming: Language Reference, GC26-4047*
- *VS COBOL II Installation and Customization for MVS, SC26-4048*

VS FORTRAN

- *VS FORTRAN Version 2: Language and Library Reference, SC26-4221*
- *VS FORTRAN Version 2: Programming Guide for CMS and MVS, SC26-4222*

VTAM

- *Planning for NetView, NCP, and VTAM, SC31-8063*
- *VTAM for MVS/ESA Diagnosis, LY43-0069*
- *VTAM for MVS/ESA Messages and Codes, SC31-6546*
- *VTAM for MVS/ESA Network Implementation Guide, SC31-6548*
- *VTAM for MVS/ESA Operation, SC31-6549*
- *VTAM for MVS/ESA Programming, SC31-6550*
- *VTAM for MVS/ESA Programming for LU 6.2, SC31-6551*
- *VTAM for MVS/ESA Resource Definition Reference, SC31-6552*

Index

Special Characters

, (comma) as decimal point 166
: (colon) 120, 121
? (question mark) 689
/ (divide sign) 133
. (period) as decimal point 166
* (asterisk)
 COUNT function 178
 COUNT_BIG function 178
 multiply sign 133
 use in subselect 312
- (minus sign) 133
+ (plus sign) 133
|| (vertical bars) 131

A

ABS function 188
ACCESSPATH column
 SYSPACKSTMT catalog table 972
 SYSSTMT catalog table 999
ACOS function 189
ACQUIRE
 column of SYSPLAN catalog table 977
ADD
 clause of ALTER TABLE statement 397
ADD VOLUMES clause of ALTER STOGROUP statement 390
AFTER clause of CREATE TRIGGER statement 617
alias
 creating 457
 description 58
 dropping 676
 naming convention 50
 qualifying a column name 114
 referencing another DB2 33
 unqualified name 56
ALIAS clause
 COMMENT ON statement 440
 CREATE ALIAS statement 457
 DROP statement 676
 LABEL ON statement 749
ALL
 clause of RELEASE statement 766
 clause of subselect 312
 keyword
 AVG function 179
 column functions 178
 MAX function 182
 MIN function 183
 SUM function 185
ALL (*continued*)
 quantified predicate 151
ALL PRIVILEGES clause
 GRANT statement 733
 REVOKE statement 796
ALL SQL clause of RELEASE statement 766
ALLOCATE CURSOR statement
 description 346
ALLOW PARALLEL clause
 ALTER FUNCTION statement 359
 CREATE FUNCTION statement 486
alphabetic extender 47, 50
ALTDATE function 1030
ALTER DATABASE statement
 description 348
ALTER FUNCTION statement
 description 351
ALTER INDEX statement
 description 364
ALTER privilege
 GRANT statement 733
 REVOKE statement 796
ALTER PROCEDURE (SQL procedure) statement
 description 384
ALTER PROCEDURE statement
 description 376
ALTER STOGROUP statement 390
ALTER TABLE statement
 description 393
ALTER TABLESPACE statement 412
ALTERAUTH column of SYSTABAUTH catalog table 1004
ALTEREDTS column
 SYSDATABASE catalog table 944
 SYSINDEXES catalog table 957
 SYSINDEXPART catalog table 959, 1009
 SYSROUTINES catalog table 993
 SYSSEQUENCES catalog table 996
 SYSSTOGROUP catalog table 1000
 SYSTABLES catalog table 1012
 SYSTABLESPACE catalog table 1016
ALTERIN privilege
 GRANT statement 728
 REVOKE statement 791
ALTERINAUTH column of SYSSCHEMAAUTH catalog table 995
ALTTIME function 1033
AND
 truth table 163
ANY
 quantified predicate 151
 USING clause of DESCRIBE statement 660

ANY (*continued*)
 USING clause of PREPARE statement 758
 APOST option
 precompiler 168
 apostrophe
 string delimiter precompiler option 168
 APOSTSQL option
 precompiler 168
 application plan
 description 31
 invalidated
 ALTER TABLE statement 409
 privileges 727, 790
 application process 28
 application program
 recovery 28
 SQLCA 883
 SQLDA 890
 ARCHIVE privilege
 GRANT statement 730
 REVOKE statement 793
 ARCHIVEAUTH column of SYSUSERAUTH catalog
 table 1021
 arithmetic operators 133
 AS clause
 CREATE VIEW statement 629
 naming result columns 312
 use in subselect 312
 AS DEFINITION ONLY clause
 DECLARE GLOBAL TEMPORARY TABLE
 statement 643
 AS IDENTITY clause
 ALTER TABLE statement 400
 CREATE TABLE statement 581
 DECLARE GLOBAL TEMPORARY TABLE
 statement 642
 AS LOCATOR clause
 CREATE FUNCTION statement 478, 497, 513
 CREATE PROCEDURE statement 546
 AS TEMP clause of CREATE DATABASE
 statement 463
 AS WORKFILE clause of CREATE DATABASE
 statement 463
 ASC clause
 CREATE INDEX statement 529
 of select-statement 332
 ASCII and DBCS characters 67
 ASIN function 190
 Assembler application program
 host variable
 EXECUTE IMMEDIATE statement 692
 referencing 120
 INCLUDE SQLCA 886
 INCLUDE SQLDA 902
 varying-length string variables 69
 assignment
 compatibility rules 84
 datetime values 91
 distinct type values 92
 IEEE floating-point numbers 87
 numbers 86
 retrieval rules 90
 row ID values 92
 storage rules 89
 strings, basic rules for 89
 assignment statement
 example 849
 SQL procedure 849
 ASSOCIATE LOCATORS statement
 description 424
 asterisk (*)
 COUNT function 180
 COUNT_BIG function 181
 multiply sign 133
 use in subselect 312
 ASUTIME clause
 ALTER FUNCTION statement 361
 ALTER PROCEDURE statement 381, 387
 CREATE FUNCTION statement 487, 505
 CREATE PROCEDURE statement 551, 561
 ASUTIME column
 SYSPROCEDURES catalog table 983
 SYSROUTINES catalog table 991
 ATAN function 191
 ATAN2 function 193
 ATANH function 192
 AUDIT
 clause of ALTER TABLE statement 402
 clause of CREATE TABLE statement 589
 auditing
 ALTER TABLE statement 402
 CREATE TABLE statement 589
 AUDITING column of SYSTABLES catalog table 1012
 AUTHHOWGOT column
 SYSDBAUTH catalog table 947
 SYSPACKAUTH catalog table 968
 SYSPLANAUTH catalog table 979
 SYSRESAUTH catalog table 986
 SYSROUTINEAUTH catalog table 988
 SYSSCHEMAAUTH catalog table 995
 SYSTABAUTH catalog table 1004
 SYSUSERAUTH catalog table 1019
 AUTHID
 column of MODESELECT catalog table 926
 column of SYSPROCEDURES catalog table 982
 column of USERNAMES catalog table 1025
 authorization ID
 description 59
 primary
 description 59
 resulting from errors 807

- authorization ID (*continued*)
 - secondary
 - description 59
 - translating
 - concepts 64
 - AUX clause of CREATE AUXILIARY TABLE
 - statement 460
 - AUXILIARY clause of CREATE AUXILIARY TABLE
 - statement 460
 - auxiliary table
 - CREATE AUXILIARY TABLE statement 459
 - description 21
 - AUXRELOBID column of SYSAUXRELS catalog
 - table 927
 - AUXTBNAME column of SYSAUXRELS catalog
 - table 927
 - AUXTBOWNER column of SYSAUXRELS catalog
 - table 927
 - AVG function 179
 - AVGSIZE column
 - SYSLOBSTATS catalog table 962
 - SYSPACKAGE catalog table 963
 - SYSPLAN catalog table 976

B

- base table 21
- basic operations in SQL 84
- basic predicate 150
- BCREATOR column
 - SYSPLANDEP catalog table 980
 - SYSVIEWDEP catalog table 1022
- BEGIN DECLARE SECTION statement 427
- BETWEEN predicate 153
- binary large object (BLOB) 71
- BINARY LARGE OBJECT data type 71
- binary string
 - description 71
- bind behavior 61
- BIND PACKAGE subcommand of DSN
 - options
 - QUALIFIER 56
- BIND PLAN subcommand of DSN
 - options
 - QUALIFIER 56
- BIND privilege
 - GRANT statement 725, 727
 - REVOKE statement 788, 790
- bind process 60
- BIND_OPTS
 - column of SYSPSMOPTS table 1048
- BINDADD privilege
 - binding a package 63
 - GRANT statement 730
 - REVOKE statement 793

- BINDADDAUTH column of SYSUSERAUTH catalog
 - table 1019
- BINDAGENT privilege
 - GRANT statement 730
 - REVOKE statement 793
- BINDAGENTAUTH column of SYSUSERAUTH catalog
 - table 1021
- BINDAUTH column
 - SYSPACKAUTH catalog table 968
 - SYSPLANAUTH catalog table 979
- BINDDATE column of SYSPLAN catalog table 976
- BINDERROR column of SYSPACKSTMT catalog
 - table 971
- binding
 - description 19
 - process 60
- BINDTIME column
 - SYSPACKAGE catalog table 963
 - SYSPLAN catalog table 976
- bit data
 - conversion restrictions 41
 - description 67
- BLOB (binary large object)
 - data type 577
 - description 71
 - description 71
 - host variable 121
 - locator 122
 - restrictions 72
- BLOB function 194
- BLOB LARGE OBJECT data type 577
- BNAME column
 - SYSCONSTDEP catalog table 940
 - SYSPACKDEP catalog table 969
 - SYSPLANDEP catalog table 980
 - SYSVIEWDEP catalog table 1022
- BOTH
 - USING clause of DESCRIBE statement 660
- BOUNDBY column of SYSPLAN catalog table 977
- BOUNDTS column
 - SYSPLAN catalog table 978
- BPOOL column
 - SYSDATABASE catalog table 944
 - SYSINDEXES catalog table 956
 - SYSTABLESPACE catalog table 1014
- BQUALIFIER column of SYSPACKDEP catalog
 - table 969
- BSCHEMA column
 - SYSCONSTDEP catalog table 940
 - SYSVIEWDEP catalog table 1022
- BSDS (bootstrap data set)
 - granting privilege to recover 730
 - revoking privilege to recover 793
- BSDS privilege
 - granting 730
 - revoking 793

BSDSAUTH column of SYSUSERAUTH catalog table 1019
 BSEQUENCEID column of SYSSEQUENCESDEP catalog table 997
 BTYPE column
 SYSCONSTDEP catalog table 940
 SYSPACKDEP catalog table 969
 SYSPLANDEP catalog table 980
 SYSVIEWDEP catalog table 1022
 buffer pool
 naming convention 51
 BUFFERPOOL clause
 ALTER DATABASE statement 348
 ALTER INDEX statement 366
 ALTER TABLESPACE statement 413
 CREATE DATABASE statement 462
 CREATE INDEX statement 536
 CREATE TABLESPACE statement 608
 BUFFERPOOL privilege
 GRANT statement 736
 REVOKE statement 799
 BUILDNAME
 column of SYSPSMOPTS table 1048
 BUILDDOWNER
 column of SYSPSMOPTS table 1048
 BUILDSHEMA
 column of SYSPSMOPTS table 1048
 built-in data type 66
 built-in function 125, 173
 BY clause of REVOKE statement 773

C

C application program
 host variable
 EXECUTE IMMEDIATE statement 692
 referencing 120
 INCLUDE SQLCA 885
 INCLUDE SQLDA 902
 varying-length string 69
 CACHE column of SYSSEQUENCES catalog table 996
 CACHESIZE
 column of SYSPLAN catalog table 977
 CALL statement 428
 example 852
 SQL procedure 851
 CALLED ON NULL INPUT clause
 ALTER FUNCTION statement 356
 ALTER PROCEDURE statement 383, 389
 CREATE FUNCTION statement 482, 500
 capturing changed data
 ALTER TABLE statement 407
 CREATE TABLE statement 590
 CARD column
 SYSTABLEPART catalog table
 description 1007

CARD column (*continued*)
 SYSTABSTATS catalog table
 description 1017
 CARDF column
 SYSCOLDIST catalog table 931
 SYSCOLDISTSTATS catalog table 932
 SYSINDEXPART catalog table 959
 SYSTABLEPART catalog table 1009
 SYSTABLES catalog table 1013
 SYSTABSTATS catalog table 1017
 CARDINALITY column
 SYSROUTINES catalog table 994
 CASCADE delete rule
 ALTER TABLE statement 406
 CREATE TABLE statement 585
 description 23
 cascade revoke 773
 CASE expression
 description 143
 result data type 99
 CASE statement
 example 854
 SQL procedure 853
 CAST specification 146
 NULL 146
 parameter marker 146
 CAST_FUNCTION column
 SYSPARMS catalog table 973
 SYSROUTINES catalog table 990
 CAST_FUNCTION_ID column of SYSPARMS catalog table 974
 casts
 data types 83
 catalog name
 naming convention 51
 VCAT clause
 ALTER INDEX statement 368
 CREATE INDEX statement 530
 CREATE TABLESPACE statement 600, 603
 catalog tables 911
 description 26, 911
 indexes 911
 IPNAMES 920
 LOCATIONS 921
 LULIST 922
 LUMODES 923
 LUNAMES 924
 MODESELECT 926
 SQL statements allowed 915
 SYSAUXRELS 927
 SYSCHECKDEP 928
 SYSCHECKS 929
 SYSCOLAUTH 930
 SYSCOLDIST
 contents 931
 SYSCOLDISTSTATS
 contents 932

catalog tables (*continued*)

- SYSCOLSTATS
 - contents 933
- SYSCOLUMNS
 - contents 934
- SYSCONSTDEP 940
- SYSCOPY
 - contents 941
- SYSDATABASE
 - contents 944
- SYSDATATYPES 946
- SYSDBAUTH 947
- SYSDBRM 950
- SYSDUMMY1 952
- SYSFIELDS 953
- SYSFOREIGNKEYS 954
- SYSINDEXES
 - contents 955
- SYSINDEXPART
 - contents 958
- SYSINDEXSTATS 960
- SYSKEYS 961
- SYSLOBSTATS 962
- SYSPACKAGE 963
- SYSPACKAUTH 968
- SYSPACKDEP 969
- SYSPACKLIST 970
- SYSPACKSTMT 971
- SYSPARMS 973
- SYSPKSYSTEM 975
- SYSPLAN 976
- SYSPLANAUTH
 - contents 979
- SYSPLANDEP
 - contents 980
- SYSPLSYSTEM 981
- SYSPROCEDURES
 - contents 982
- SYSRELS
 - contents 985
- SYSRESAUTH 986
- SYSROUTINEAUTH 988
- SYSROUTINES 989
- SYSSCHEMAAUTH 995
- SYSSEQUENCES 996
- SYSSEQUENCESDEP 997
- SYSSTMT 998
- SYSSTOGROUP
 - contents 1000
- SYSSTRINGS
 - contents 1001
- SYSSYNONYMS 1003
- SYSTABAUTH
 - contents 1004
- SYSTABLEPART
 - contents 1007

catalog tables (*continued*)

- SYSTABLES
 - contents 1010
- SYSTABLESPACE
 - contents 1014
- SYSTABSTATS
 - contents 1017
- SYSTRIGGERS 1018
- SYSUSERAUTH 1019
- SYSVIEWDEP
 - contents 1022
- SYSVIEWS 1023
- SYSVOLUMES 1024
 - table space 911
- USERNAMES 1025

catalog, DB2

- description 26
- tables 911
 - See also catalog tables

CCSID

- clause of ALTER DATABASE statement 349
- clause of ALTER TABLESPACE statement 422
- clause of CREATE DATABASE statement 463
- clause of CREATE DISTINCT TYPE statement 468
- clause of CREATE FUNCTION statement 478, 496, 512
- clause of CREATE GLOBAL TEMPORARY TABLE statement 522
- clause of CREATE PROCEDURE statement 545, 558
- clause of CREATE TABLE statement 590
- clause of CREATE TABLESPACE statement 610
- clause of DECLARE GLOBAL TEMPORARY TABLE statement 645
- column of SYSPARMS catalog table 974

CCSID (coded character set identifier) 38

- See also character conversion
- definition 38
- description 39
- system 40

CD-ROM, books on 8

CEIL function 195

CEILING function 195

CHAR

- function 196

CHAR data type 69

CHAR LARGE OBJECT data type 69, 576

CHAR VARYING data type 69, 576

character 47

character conversion

- assignment rules 90
- character set 38
- code page 38
- code point 38
- coded character set 38
- comparison rules 95

character conversion (*continued*)
 concatenation rules 328
 contracting conversion 42
 description 38
 encoding scheme 39
 expanding conversion 41
 substitution byte 39
 SYSIBM.SYSSTRINGS catalog table 1001
 UNION and UNION ALL rules 328
 CHARACTER data type
 CREATE TABLE statement 576
 description 69
 character large object (CLOB) 71
 CHARACTER LARGE OBJECT data type 69, 576
 character set 38
 character string
 assignment 89
 comparison 94
 constants 102
 description 67
 empty 67
 CHARACTER VARYING data type 69, 576
 CHARSET column
 SYSDBRM catalog table 950
 SYSPACKAGE catalog table 964
 CHECK
 clause of ALTER TABLE statement 403
 clause of CREATE TABLE statement 586
 column of SYSVIEWS catalog table 1023
 check constraint
 See table check constraint
 CHECKCONDITION column
 SYSCHECKS catalog table 929
 CHECKFLAG column
 SYSTABLEPART catalog table 1008
 SYSTABLES catalog table 1012
 CHECKNAME column
 SYSCHECKDEP catalog table 928
 SYSCHECKS catalog table 929
 CHECKRID5B column
 SYSTABLEPART catalog table 1009
 SYSTABLES catalog table 1013
 CHECKS column
 SYSTABLES catalog table 1013
 CHILDREN column of SYSTABLES catalog table 1011
 CLOB (character large object)
 host variable 121
 locator 122
 CLOB (character large object)
 data type 576
 description 69
 description 71
 restrictions 72
 CLOB function 202
 CLOSE
 clause of ALTER INDEX statement 366
 clause of ALTER TABLESPACE statement 415
 clause of CREATE INDEX statement
 description 536
 clause of CREATE TABLESPACE statement
 description 609
 statement
 description 436
 closed state of cursor 755
 CLOSERULE column
 SYSINDEXES catalog table 956
 SYSTABLESPACE catalog table 1015
 CLUSTER clause of CREATE INDEX statement
 description 534
 CLUSTERED column of SYSINDEXES catalog table
 description 955
 CLUSTERING column of SYSINDEXES catalog table
 description 955
 CLUSTERRATIO column
 SYSINDEXES catalog table 956
 SYSINDEXSTATS catalog table 960
 CLUSTERRATIOF column
 SYSINDEXES catalog table 957
 SYSINDEXSTATS catalog table 960
 CLUSTERTYPE column of SYSTABLES catalog
 table 1010
 CNAME column
 SYSPKSYSTEM catalog table 975
 SYSPLSYSTEM catalog table 981
 COALESCE function 99, 203, 232
 COBOL application program
 host structure 123
 host variable
 description 120
 EXECUTE IMMEDIATE statement 692
 INCLUDE SQLCA 887
 varying-length string 69
 code page 38
 code point 38
 coded character set 38
 coded character set identifier 38
 See *also* CCSID (coded character set identifier)
 COLCARDATA column of SYSCOLSTATS catalog
 table 933
 COLCARDF column of SYSCOLUMNS catalog
 table 938
 COLCOUNT column
 SYSINDEXES catalog table 955
 SYSRELS catalog table 985
 SYSTABLES catalog table 1010
 COLGROUPCOLNO column
 SYSCOLDIST catalog table 931
 SYSCOLDISTSTATS catalog table 932
 collection, package
 granting privileges 714
 revoking privileges 777

- collection, package (*continued*)
 - SET CURRENT PACKAGESET statement 817
- COLLID clause
 - ALTER FUNCTION statement 360
 - ALTER PROCEDURE statement 380, 386
 - CREATE FUNCTION statement 487, 504
 - CREATE PROCEDURE statement 550, 560
- COLLID column
 - SYSCOLAUTH catalog table 930
 - SYSPACKAGE catalog table 963
 - SYSPACKAUTH catalog table 968
 - SYSPACKLIST catalog table 970
 - SYSPACKSTMT catalog table 971
 - SYSPKSYSTEM catalog table 975
 - SYSPROCEDURES catalog table 982
 - SYSROUTINEAUTH catalog table 988
 - SYSROUTINES catalog table 989
 - SYSTABAUTH catalog table 1005
- COLNAME column
 - SYSAUXRELS catalog table 927
 - SYSCHECKDEP catalog table 928
 - SYSCOLAUTH catalog table 930
 - SYSFOREIGNKEYS catalog table 954
 - SYSKEYS catalog table 961
- COLNO column
 - SYSCOLUMNS catalog table 934
 - SYSFIELDS catalog table 953
 - SYSFOREIGNKEYS catalog table 954
 - SYSKEYS catalog table 961
- colon 120
 - host variable in SQL 121
- COLSEQ column
 - SYSFOREIGNKEYS catalog table 954
 - SYSKEYS catalog table 961
- COLSTATUS column of SYSCOLUMNS catalog table 938
- COLTYPE column of SYSCOLUMNS catalog table 934
- column
 - controlling changes 25
 - derived
 - CREATE VIEW statement 629
 - functions 173
 - INSERT statement 743
 - null value 314
 - string comparison 95
 - UPDATE statement 836
 - description 21
 - name
 - ambiguous reference 115
 - correlated reference 116
 - in a result 314
 - undefined reference 115
 - restricting values 25
 - rules 327
- COLUMN clause
 - COMMENT ON statement 440
 - LABEL ON statement 750
- column function 178
- COLVALUE column
 - SYSCOLDIST catalog table
 - description 931
 - SYSCOLDISTSTATS catalog table
 - description 932
- COMMA
 - column of SYSDBRM catalog table 950
 - column of SYSPACKAGE catalog table 964
 - option of precompiler 166
- comment
 - adding 438
 - replacing 438
 - SQL 171
- COMMENT ON statement
 - column name qualification 114
 - description 438
- commit
 - description 28
- COMMIT ON RETURN clause
 - ALTER PROCEDURE statement 383, 388
 - CREATE PROCEDURE statement 552, 562
- COMMIT statement
 - description 444
- COMMIT_ON_RETURN column
 - SYSPROCEDURES catalog table 984
 - SYSROUTINES catalog table 992
- comparison
 - compatibility rules 84
 - datetime values 96
 - distinct type values 96
 - numbers 94
 - row ID values 96
 - strings 94
- compatibility
 - data types 84
 - rules 84
- COMPILE_OPTS
 - column of SYSPSMOPTS table 1048
- compound statement
 - example 859
 - order of statements in 858
 - SQL procedure 855
- COMPRESS
 - clause of ALTER TABLESPACE statement 419
 - clause of CREATE TABLESPACE statement 610
 - column of SYSTABLEPART catalog table 1008
- CONCAT
 - function 205
 - operator 131
- concatenation
 - CONCAT function 205
 - operator 131

- concurrency
 - application 28
 - LOCK TABLE statement 751
- CONNECT
 - option of precompiler 164
 - statement
 - differences, type 1 and type 2 446
 - type 1 449
 - type 2 454
- connected state 36
- connection
 - DB2 private 33
 - SQL 34
- connection exit routine
 - description 111
- connection state
 - application process 34, 451
 - CONNECT (Type 1) statement 451
 - SET CONNECTION statement 809
 - SQL 34
- constant
 - character string 102
 - datetime 103
 - decimal 102
 - floating-point 102
 - graphic string 103
 - hexadecimal 102
 - integer 101
- constraint
 - See table check constraint
- CONSTRAINT
 - clause of ALTER TABLE statement 403
 - clause of CREATE TABLE statement 586
- CONTAINS SQL clause
 - ALTER FUNCTION statement 356
 - ALTER PROCEDURE statement 379, 386
 - CREATE FUNCTION statement 483, 501
 - CREATE PROCEDURE statement 549, 560
- CONTINUE
 - clause of WHENEVER statement 845
- CONTINUE handler
 - SQL procedure 858
- CONTOKEN column
 - SYSCOLAUTH catalog table 930
 - SYSPACKAGE catalog table 963
 - SYSPACKSTMT catalog table 971
 - SYSPKSYSTEM catalog table 975
 - SYSROUTINEAUTH catalog table 988
 - SYSTABAUTH catalog table 1005
- control character 48
- conversion of numbers
 - errors 807
 - precision 87
 - scale 87
- conversion, character 38
 - See *also* character conversion
- CONVERT TO clause
 - ALTER INDEX statement 364
- CONVLIMIT column of LUMODES catalog table
 - description 923
- COPY
 - clause of ALTER INDEX statement 366
 - clause of CREATE INDEX statement 537
 - column of SYSINDEXES catalog table 957
- COPY privilege
 - GRANT statement 725
 - REVOKE statement 788
- COPYAUTH column of SYSPACKAUTH catalog table 968
- COPYLRSN column of SYSINDEXES catalog table 957
- correlated reference
 - correlation name
 - defining 114
 - FROM clause of subselect 315
 - naming convention 52
 - qualifying a column name 114
 - description 116
 - HAVING clause 322
 - WHERE clause 320
- COS function 206
- COSH function 207
- COUNT function 180
- COUNT_BIG function 181
- CREATE ALIAS statement 457
- CREATE AUXILIARY TABLE statement
 - description 459
- CREATE DATABASE statement
 - description 462
- CREATE DISTINCT TYPE statement
 - description 465
- CREATE FUNCTION (external scalar) statement
 - description 473
- CREATE FUNCTION (external table) statement
 - description 492
- CREATE FUNCTION (sourced) statement
 - description 508
- CREATE FUNCTION statement
 - description 472
- CREATE GLOBAL TEMPORARY TABLE statement
 - description 520
- CREATE IN privilege
 - binding a package 63
 - GRANT statement 714
 - REVOKE statement 777
- CREATE INDEX statement
 - description 525
- CREATE PROCEDURE (SQL procedure) statement
 - description 555
- CREATE PROCEDURE statement
 - assignment statement 849
 - description 541

CREATE PROCEDURE statement (*continued*)
 SQL procedure body 848
 CREATE STOGROUP statement 565
 CREATE SYNONYM statement 568
 CREATE TABLE statement
 description 570
 CREATE TABLESPACE statement 597
 CREATE TRIGGER statement
 description 615
 CREATE VIEW statement
 description 627
 use 27
 CREATEALIAS privilege
 GRANT statement 730
 REVOKE statement 793
 CREATEALIASAUTH column of SYSUSERAUTH
 catalog table 1020
 CREATEDBA privilege
 GRANT statement 730
 REVOKE statement 793
 CREATEDBAAUTH column of SYSUSERAUTH catalog
 table 1019
 CREATEDBC privilege
 GRANT statement 730
 REVOKE statement 794
 CREATEDBCAUTH column of SYSUSERAUTH catalog
 table 1019
 CREATEDBY column
 SYSDATABASE catalog table 944
 SYSDATATYPES catalog table 946
 SYSINDEXES catalog table 956
 SYSROUTINES catalog table 989
 SYSSEQUENCES catalog table 996
 SYSSTOGROUP catalog table 1000
 SYSSYNONYMS catalog table 1003
 SYSTABLES catalog table 1012
 SYSTABLESPACE catalog table 1015
 SYSTRIGGERS catalog table 1018
 CREATEDTS column
 SYSCOLUMNS catalog table 939
 SYSDATABASE catalog table 944
 SYSDATATYPES catalog table 946
 SYSINDEXES catalog table 957
 SYSROUTINES catalog table 993
 SYSSEQUENCES catalog table 996
 SYSSTOGROUP catalog table 1000
 SYSSYNONYMS catalog table 1003
 SYSTABLES catalog table 1012
 SYSTABLESPACE catalog table 1016
 SYSTRIGGERS catalog table 1018
 CREATEIN privilege
 GRANT statement 728
 REVOKE statement 791
 CREATEINAUTH column of SYSSCHEMAAUTH
 catalog table 995
 CREATESG privilege
 GRANT statement 731
 REVOKE statement 794
 CREATESGAUTH column of SYSUSERAUTH catalog
 table 1019
 CREATETAB privilege
 GRANT statement 715
 REVOKE statement 778
 CREATETABAUTH column of SYSDBAUTH catalog
 table 947
 CREATETMTAB privilege
 GRANT statement 731
 REVOKE statement 794
 CREATETMTABAUTH column of SYSUSERAUTH
 catalog table 1021
 CREATETS privilege
 GRANT statement 715
 REVOKE statement 778
 CREATETSAUTH column of SYSDBAUTH catalog
 table 947
 CREATOR column
 SYSCHECKS catalog table 929
 SYSCOLAUTH catalog table 930
 SYSDATABASE catalog table 944
 SYSFOREIGNKEYS catalog table 954
 SYSINDEXES catalog table 955
 SYSPACKAGE catalog table 963
 SYSPLAN catalog table 976
 SYSRELS catalog table 985
 SYSSTOGROUP catalog table 1000
 SYSSYNONYMS catalog table 1003
 SYSTABLES catalog table 1010
 SYSTABLESPACE catalog table 1014
 SYSVIEWS catalog table 1023
 CURRENCY function 1035
 CURRENT
 clause of RELEASE statement 765
 current connection state 35
 CURRENT DATE special register 106
 CURRENT DEGREE special register
 assigning a value 812
 description 106
 setting 812
 CURRENT LC_CTYPE special register
 description 107
 CURRENT LOCALE LC_CTYPE special register
 assigning a value 814
 description 107
 CURRENT OF clause
 DELETE statement 655
 CURRENT OPTIMIZATION HINT special register
 assigning a value 816
 description 107
 CURRENT PACKAGESET special register
 assigning a value 817
 description 108

CURRENT PACKAGESET special register (*continued*)
 stored procedures 818

CURRENT PATH clause
 SET CURRENT PATH statement 819

CURRENT PATH special register
 assigning a value 819
 description 108

CURRENT PRECISION special register
 assigning a value 822
 description 109

CURRENT RULES special register
 assigning a value 823
 description 109

current server
 description 447
 designating
 CONNECT (Type 1) statement 449
 CONNECT (Type 2) statement 454

CURRENT SERVER special register
 description 111

CURRENT SQLID special register
 assigning a value 824
 description 111
 initial value 61

CURRENT TIME special register
 description 112

CURRENT TIMESTAMP special register
 description 112

CURRENT TIMEZONE special register 112

CURRENTSERVER
 column of SYSPLAN catalog table 977
 option of BIND PLAN subcommand 448
 option of REBIND PLAN subcommand 448

cursor
 associating 346
 closed state 755
 closing
 CLOSE statement 436
 CONNECT (Type 1) statement 449
 CONNECT (Type 2) statement 454
 error in FETCH 709
 error in UPDATE 838
 declaring 634
 open state 709
 opening
 errors 755
 OPEN statement 753
 using
 current row 709
 FETCH statement 707
 positions 709

CYCLE column of SYSSEQUENCES catalog
 table 996

D

DATA CAPTURE clause
 ALTER TABLE statement 407
 CREATE TABLE statement 590

data compression
 COMPRESS clause of ALTER TABLESPACE
 statement 419
 COMPRESS clause of CREATE TABLESPACE
 statement 610

data type
 built-in 66
 casting between 83
 character string 67
 compatibility matrix 85
 CREATE TABLE statement 575
 datetime 75
 distinct 79
 graphic string 70
 list of built-in types 66
 name, unqualified 56
 naming convention
 built-in 51
 distinct type 52
 numeric 73
 promotion 81
 result column 314
 results of an operation 99
 row ID 78
 unqualified name 56

data types
 string restrictions 72

database
 altering
 ALTER DATABASE statement 348
 creating 462
 default database 55
 description 26
 dropping 676
 DSNDB04 (default database) 55
 limits 869
 naming convention 52
 privileges 715, 778

DATABASE
 clause of ALTER DATABASE statement 348
 clause of DROP statement 676
 clause of GRANT statement 716
 clause of REVOKE statement 779

DATA CAPTURE column of SYSTABLES catalog
 table 1012

DATATYPEID column
 DATATYPES catalog table 946
 SYSCOLUMNS catalog table 938
 SYSPARMS catalog table 973
 SYSSEQUENCES catalog table 996

- date
 - arithmetic 139
 - data type 75
 - duration 138
 - strings 76, 78
- DATE
 - data type
 - CREATE TABLE statement 577
 - description 75
 - function 208
- DATE FORMAT field of panel DSNTIP4 170
- date routine
 - CHAR function 196
- DATEGRANTED column
 - SYSCOLAUTH catalog table 930
 - SYSDBAUTH catalog table 947
 - SYSPLANAUTH catalog table 979
 - SYSRESAUTH catalog table 986
 - SYSTABAUTH catalog table 1004
 - SYSUSERAUTH catalog table 1019
- datetime
 - arithmetic 138
 - constants 103
 - data types
 - description 75
 - string representation 76
 - format
 - setting through the CHAR function 196
 - string formats 76
- DAY function 209
- day of week calculation 213
- DAYNAME function 1037
- DAYOFMONTH function 210
- DAYOFWEEK function 211
- DAYOFYEAR function 212
- DAYS function 213
- DB2 books online 8
- DB2 catalog tables
 - See catalog tables
- DB2 private protocol access
 - authorization ID 63
 - description 31, 33
 - mixed environment 873
- DB2 system tables
 - SYSPSM 1047
 - SYSPSMOPTS 1048
- DB2 version identification, current server 450, 455
- DBADM authority
 - GRANT statement 715
 - REVOKE statement 778
- DBADMAUTH column of SYSDBAUTH catalog table 947
- DBCLOB (double byte character large object)
 - locator 122
- DBCLOB (double-byte character large object)
 - data type 70, 577
- DBCLOB (double-byte character large object)
 - (continued)
 - description 71
 - host variable 121
 - restrictions 72
- DBCLOB function 214
- DBCS (double-byte character set)
 - ASCII 67
 - EBCDIC 48, 67
 - site 68
 - SQL ordinary identifier 47, 48
- DBCS_CCSID column
 - SYSDATABASE catalog table 945
 - SYSTABLESPACE catalog table 1016
- DBCTRL authority
 - GRANT statement 715
 - REVOKE statement 778
- DBCTRLAUTH column of SYSDBAUTH catalog table 947
- DBID
 - column of SYSCHECKS catalog table 929
 - column of SYSDATABASE catalog table 944
 - column of SYSINDEXES catalog table 955
 - column of SYSTABLES catalog table 1010
 - column of SYSTABLESPACE catalog table 1014
 - column of SYSTRIGGERS catalog table 1018
- DBINFO
 - clause of ALTER FUNCTION statement 360
 - clause of ALTER PROCEDURE statement 380
 - clause of CREATE FUNCTION statement 486, 504
 - clause of CREATE PROCEDURE statement 549
 - column of SYSROUTINES catalog table 991
- DBMAINT authority
 - GRANT statement 715
 - REVOKE statement 778
- DBMAINTAUTH column of SYSDBAUTH catalog table 947
- DBNAME column
 - SYSCOPY catalog table 941
 - SYSINDEXES catalog table 955
 - SYSLOBSTATS catalog table 962
 - SYSTABAUTH catalog table 1004
 - SYSTABLEPART catalog table 1007
 - SYSTABLES catalog table 1010
 - SYSTABLESPACE catalog table 1014
 - SYSTABSTATS catalog table 1017
- DBPROTOCOL column
 - SYSPACKAGE catalog table 967
 - SYSPLAN catalog table 978
- DBRM (database request module)
 - description 31
- DCLGEN subcommand of DSN
 - description 75
- DCOLLID column of SYSPACKDEP catalog table 969
- DCOLNAME column of SYSSEQUENCESDEP catalog table 997

DCONSTNAME column of SYSCONSTDEP catalog table 940
 DCONTOKEN column of SYSPACKDEP catalog table 969
 DCREATOR column
 SYSEQUENCESDEP catalog table 997
 SYSVIEWDEP catalog table 1022
 deadlock
 locks and uncommitted changes 28
 DEC function 215
 DEC15 precompiler option 134
 DEC31
 column of SYSDBRM catalog table 950
 column of SYSPACKAGE catalog table 964
 precompiler option 134
 decimal
 constants 102
 data type
 CREATE TABLE statement 575
 description 74
 function
 description 215
 numbers 74
 DECIMAL POINT IS field of panel DSNTIPF 166
 decimal point precompiler option 166
 DECLARE CURSOR statement
 description 634
 DECLARE GLOBAL TEMPORARY TABLE statement
 description 639
 DECLARE STATEMENT statement 649
 DECLARE TABLE statement
 description 650
 DEFAULT
 column of SYSCOLUMNS catalog table 936
 default database (DSNDB04)
 implicit specification 55
 DEFAULTVALUE column of SYSCOLUMNS catalog table 938
 DEFER
 clause of CREATE INDEX statement 536
 DEFERPREP column
 SYSPACKAGE catalog table 965
 SYSPLAN catalog table 977
 DEFERPREPARE column of SYSPACKAGE catalog table 966
 deferred embedded SQL 19
 define behavior 61
 DEFINE clause
 CREATE TABLESPACE statement 534, 606
 DEGREE
 column of SYSPACKAGE catalog table 965
 column of SYSPLAN catalog table 977
 DEGREES function 217
 DELETE
 clause of TRIGGER statement 617
 statement
 description 653
 DELETE privilege
 GRANT statement 733
 REVOKE statement 796
 delete rule 23, 656
 delete-connected 23
 DELETEAUTH column of SYSTABAUTH catalog table 1005
 DELETERULE column of SYSRELS catalog table 985
 deleting
 rows from a table 653
 SQL objects 674
 delimited identifier in SQL 49
 delimiter
 SQL 49
 dependency
 of objects on each other 683
 dependent
 row 23
 table 23
 DESC clause
 CREATE INDEX statement 529
 select-statement 332
 descendent table 23
 DESCRIBE CURSOR statement
 description 666
 DESCRIBE INPUT statement
 prepared statement 668
 DESCRIBE PROCEDURE statement
 description 671
 DESCRIBE statement
 prepared statement 659
 table or view 659
 variables 660
 descriptor name 52
 DETERMINISTIC clause
 ALTER FUNCTION statement 356
 ALTER PROCEDURE statement 379, 385
 CREATE FUNCTION statement 482, 500
 CREATE PROCEDURE statement 549, 559
 DETERMINISTIC column of SYSROUTINES catalog table 990
 DEVTYPE column of SYSCOPY catalog table 941
 DFSMSShm (Data Facility Hierarchical Storage Manager)
 dropping an index or table space 683
 digit, description in DB2 47
 DIGITS function 218
 DISALLOW PARALLEL clause
 ALTER FUNCTION statement 359
 CREATE FUNCTION statement 486, 504
 DISCONNECT
 column of SYSPLAN catalog table 978
 DISPLAY privilege
 GRANT statement 731
 REVOKE statement 794

DISPLAYAUTH column of SYSUSERAUTH catalog table 1020
 DISPLAYDB privilege
 GRANT statement 715
 REVOKE statement 778
 DISPLAYDBAUTH column of SYSDBAUTH catalog table 947
 DISTINCT
 clause of subselect 312
 keyword
 AVG function 179
 column functions 178
 COUNT function 180
 COUNT_BIG function 181
 MAX function 182
 MIN function 183
 STDDEV function 184
 SUM function 185
 VAR 186
 VARIANCE function 186
 distinct type
 assignment of values 92
 casting 83
 comparison of values 96
 CREATE TABLE statement 577
 creating 465
 description 79
 dropping 677
 name, unqualified 52, 56
 naming convention 52
 privileges 718, 781
 promotion 81
 unqualified name 56
 DISTINCT TYPE clause
 COMMENT ON statement 441
 DROP statement 677
 distributed data
 CONNECT statement 446
 CURRENT SERVER special register 111
 description 31
 RELEASE statement 765
 SET CONNECTION statement 809
 Distributed Relational Database Architecture (DRDA) 32
 distributed unit of work 31
 See also DB2 private protocol access
 DLOCATION column of SYSPACKDEP catalog table 969
 DNAME column
 SYSPACKDEP catalog table 969
 SYSPLANDEP catalog table 980
 SYSSEQUENCESDEP catalog table 997
 SYSVIEWDEP catalog table 1022
 dormant connection state 35
 DOUBLE data type
 CREATE TABLE statement 575
 DOUBLE data type (*continued*)
 description 74
 DOUBLE function 219
 DOUBLE PRECISION data type
 CREATE TABLE statement 575
 description 74
 double precision floating-point number 74
 double-byte character
 See also DBCS (double-byte character set)
 LABEL ON statement 750
 strings 70
 truncated during assignment 90
 double-byte character large object (DBCLOB) 71
 double-byte character set (DBCS) 47
 DOUBLE_PRECISION function 219
 DOWNER column of SYSPACKDEP catalog table 969
 DRDA (Distributed Relational Database Architecture) 32
 DRDA access
 authorization ID 63
 CONNECT (Type 1) statement 449
 CONNECT (Type 2) statement 454
 description 31, 32
 mixed environment 873
 restricted function 33
 DROP 674
 DROP FOREIGN KEY clause of ALTER TABLE statement 407
 DROP PRIMARY KEY clause of ALTER TABLE statement 407
 DROP privilege
 GRANT statement 715
 REVOKE statement 778
 DROPAUTH column of SYSDBAUTH catalog table 948
 DROPIN privilege
 GRANT statement 728
 REVOKE statement 791
 DROPINAUTH column of SYSSCHEMAAUTH catalog table 995
 DSN_FUNCTION_TABLE table
 EXPLAIN statement 695
 DSN_STATEMNT_TABLE table
 EXPLAIN statement 695
 DSNAME
 column of SYSCOPY catalog table 941
 DSNUM
 column of SYSCOPY catalog table 941
 DSSIZE clause
 CREATE TABLESPACE statement 607
 DSSIZE column
 SYSTABLESPACE catalog table 1016
 DSVOLSER column of SYSCOPY catalog table 942
 DTBCREATOR column of SYSCONSTDEP catalog table 940

DTBNAME column of SYSCONSTDEP catalog table 940

DTYPE column
 SYSCONSTDEP catalog table 940
 SYSPACKDEP catalog table 969

duplicate rows, UNION clause 327

duration
 date 138
 labeled 137
 time 138
 timestamp 138

DYNAMIC RESULT SET clause
 ALTER PROCEDURE statement 378, 385
 CREATE PROCEDURE statement 547, 559

dynamic SQL
 description 19, 340
 DYNAMICRULES bind option 61
 EXECUTE IMMEDIATE statement 692
 EXECUTE statement 689
 execution 342
 INTO clause
 DESCRIBE statement 659
 PREPARE statement 757
 invocation of SELECT statement 343
 preparation 342
 SQLDA 890
 statements allowed 873

DYNAMICRULES
 bind option 56
 dynamic SQL authorization 61
 column of SYSPACKAGE catalog table 966
 column of SYSPLAN catalog table 978
 unqualified names 56

DYNAMICRULES behavior 61

E

EBCDIC and DBCS characters 67

EBCDIC CODED CHAR SET field of panel DSNTIPF 169

edit routine
 named in CREATE TABLE statement 588
 specified by EDITPROC option 588

EDITPROC clause
 CREATE TABLE statement 588

EDPROC column of SYSTABLES catalog table 1010

empty table, description 21

ENABLE
 column of SYSPKSYSTEM catalog table 975
 column of SYSPLSYSTEM catalog table 981

encoding scheme 39

ENCODING_SCHEME column
 SYSDATABASE catalog table 944
 SYSDATATYPES catalog table 946
 SYSPARMS catalog table 974
 SYSTABLES catalog table 1013

ENCODING_SCHEME column (*continued*)
 SYSTABLESPACE catalog table 1016

ENCRYPTPSWDS column of LUNAMES catalog table 924

END DECLARE SECTION statement 687

EPOCH column of SYSTABLEPART catalog table 1009

ERASE clause
 ALTER INDEX statement 369
 ALTER TABLESPACE statement 419
 CREATE INDEX statement 531
 CREATE TABLESPACE statement 603

ERASERULE column
 SYSINDEXES catalog table 956
 SYSTABLESPACE catalog table 1014

error
 arithmetic expression 807
 closes cursor 755
 during FETCH 709
 during update 838
 numeric conversion 807
 signaling 831

ERRORBYTE column of SYSSTRINGS catalog table 1001

ESCAPE clause
 LIKE predicate 159

EUR (IBM European standard) 76

evaluation order 143

EXCLUSIVE
 option of LOCK TABLE statement 752

exclusive dependence 773

executable statement 340, 341

EXECUTE IMMEDIATE statement
 description 692

EXECUTE privilege
 GRANT statement 721, 725, 727
 REVOKE statement 784, 788, 790

EXECUTE statement
 description 689

EXECUTEAUTH column
 SYSPACKAUTH catalog table 968
 SYSPLANAUTH catalog table 979
 SYSROUTINEAUTH catalog table 988

EXISTS predicate 153

EXIT handler
 SQL procedure 858

exit procedure 402

exit routine 196, 589
 named in ALTER TABLE statement 402
 named in CREATE TABLE statement 582, 589

EXITPARAM column of SYSFIELDS catalog table 953

EXITPARML column of SYSFIELDS catalog table 953

EXP function 220

EXPLAIN
 column of SYSPACKAGE catalog table 964
 statement
 description 694

- explainable statement
 - description 694
 - EXPLAIN statement 695
 - using bind or rebind 696
- EXPLAN column of SYSPLAN catalog table 977
- exposed name 117
- EXPREDICATE column of SYSPLAN catalog table 977
- expression
 - arithmetic operators 133
 - CASE 143
 - CAST specification 146
 - concatenation operator 131
 - datetime operands 137
 - decimal operands 134
 - floating-point operands 137
 - integer operands 134
 - precedence of operation 143
 - subselect statement 312
 - without operators 131
- EXTERNAL ACTION clause
 - ALTER FUNCTION statement 357
 - CREATE FUNCTION statement 483, 501
- EXTERNAL clause
 - CREATE FUNCTION statement 481, 499
 - CREATE PROCEDURE statement 547
- external function 125
- EXTERNAL NAME clause
 - ALTER FUNCTION statement 355
 - ALTER PROCEDURE statement 378
- EXTERNAL_ACTION column of SYSROUTINES catalog table 990
- EXTERNAL_NAME column
 - SYSROUTINES catalog table 994
- EXTERNAL_SECURITY column
 - SYS PROCEDURES catalog table 983
 - SYSROUTINES catalog table 992

F

- FARINDREF column of SYSTABLEPART catalog table
 - description 1007
- FAROFFPOSF column of SYSINDEXPART catalog table 959
- FENCED
 - clause of CREATE FUNCTION statement 482, 500
 - clause of CREATE PROCEDURE statement 546, 559
 - column of SYSROUTINES catalog table 991
- FETCH statement
 - description 707
- field description 401
- field procedure
 - comparisons 94
 - named in ALTER TABLE statement 401
 - named in CREATE TABLE statement 582
- FIELDPROC clause
 - ALTER TABLE statement 401
 - CREATE TABLE statement 582
- FILESEQNO column of SYSCOPY catalog table 941
- FINAL CALL clause
 - ALTER FUNCTION statement 359, 485
 - CREATE FUNCTION statement 503
- FINAL_CALL column of SYSROUTINES catalog table 990
- FIRSTKEYCARD column
 - SYSINDEXSTATS catalog table 960
- FIRSTKEYCARDF column
 - SYSINDEXES catalog table 956
 - SYSINDEXSTATS catalog table 960
- FLDPROC column
 - SYSCOLUMNS catalog table 938
 - SYSFIELDS catalog table 953
- FLDTYPE column of SYSFIELDS catalog table 953
- FLOAT
 - data type
 - CREATE TABLE statement 575
 - description 74
 - function 219
 - floating-point
 - constants 102
 - double precision number 74
 - single precision number 74
- FLOOR function 222
- FOR
 - clause of CREATE ALIAS statement 458
 - clause of CREATE DISTINCT TYPE statement 467
 - clause of CREATE SYNONYM statement 568
 - clause of CREATE TABLE statement 576
 - clause of EXPLAIN statement 695
- FOR EACH ROW clause of TRIGGER statement 619
- FOR EACH STATEMENT clause of TRIGGER statement 619
- FOR FETCH ONLY clause 333
- FOR READ ONLY clause 333
- FOR RESULT SET clause of ALLOCATE CURSOR statement 346
- FOR UPDATE OF clause
 - NOFOR precompiler option 172
 - select-statement 332
- foreign key
 - description 23
 - See *also* key, foreign
- FOREIGN KEY clause
 - ALTER TABLE statement
 - description 404
 - CREATE TABLE statement
 - description 584
- FOREIGNKEY column of SYSCOLUMNS catalog table 938
- Fortran application program
 - host variable 120

Fortran application program (*continued*)

- INCLUDE SQLCA 887
 - varying-length string 69
- FREE LOCATOR statement 710
- free space
 - index 533
 - table space 416
- FREEPAGE
 - clause of ALTER INDEX statement
 - description 371
 - clause of ALTER TABLESPACE statement
 - description 416
 - clause of CREATE INDEX statement
 - description 533
 - clause of CREATE TABLESPACE statement
 - description 604
 - column of SYSINDEXPART catalog table 958
 - column of SYSTABLEPART catalog table 1008
- FREESPACE column of SYSLOBSTATS catalog table 962
- FREQUENCYF column
 - SYSCOLDIST catalog table 931
 - SYSCOLDISTSTATS catalog table 932
- FROM clause
 - DELETE statement 654
 - PREPARE statement 758
 - REVOKE statement 773
 - subselect 315
- FULL OUTER JOIN 319
 - example 325
 - FROM clause of subselect 319
- FULLKEYCARD column of SYSINDEXSTATS catalog table 960
- FULLKEYCARDF column
 - SYSINDEXES catalog table 956
 - SYSINDEXSTATS catalog table 960
- fullselect 327, 330
- function
 - column
 - column name 114
 - description 125
 - maximum number in select 870
 - name, unqualified 56
 - unqualified name 56
- FUNCTION clause
 - COMMENT ON statement 441
 - DROP statement 677
- function resolution 127
- function table 694
- FUNCTION_TYPE column
 - SYSROUTINES catalog table 989
- function, built-in
 - column
 - AVG 179
 - COUNT 180
 - COUNT_BIG 181
 - description 173, 178

function, built-in (*continued*)

- column (*continued*)
 - example 178
 - MAX 182
 - MIN 183
 - STDDEV 184
 - SUM 185
 - VAR 186
 - VARIANCE 186
- description 125
- invocation 127
- name, unqualified 56
- nesting 187
- resolution 127
- scalar
 - ABS 188
 - ACOS 189
 - ASIN 190
 - ATAN 191
 - ATAN2 193
 - ATANH 192
 - BLOB 194
 - CEIL or CEILING 195
 - CHAR 196
 - CLOB 202
 - COALESCE 203
 - CONCAT 205
 - COS 206
 - COSH 207
 - DATE 208
 - DAY 209
 - DAYOFMONTH 210
 - DAYOFWEEK 211
 - DAYOFYEAR 212
 - DAYS 213
 - DBCLOB 214
 - DECIMAL or DEC 215
 - DEGREES 217
 - description 187
 - DIGITS 218
 - DOUBLE or DOUBLE_PRECISION 219
 - example 187
 - EXP 220
 - FLOAT 219
 - FLOOR 222
 - GRAPHIC 223
 - HEX 225
 - hour 226
 - IDENTITY_VAL_LOCAL 227
 - IFNULL 232
 - INSERT 233
 - INTEGER or INT 236
 - JULIAN_DAY 237
 - LCASE 238
 - LEFT 239
 - LENGTH 241
 - LN 242

function, built-in (*continued*)
 scalar (*continued*)
 LOCATE 243
 LOG 242
 LOG10 245
 LOWER 238
 LTRIM 246
 MICROSECOND 247
 MIDNIGHT_SECONDS 248
 MINUTE 249
 MOD 250
 MONTH 252
 MULTIPLY_ALT 253
 NULLIF 254
 POSSTR 255
 POWER 257
 QUARTER 258
 RADIANS 259
 RAISE_ERROR 260
 RAND 261
 REAL 262
 REPEAT 263
 REPLACE 265
 RIGHT 267
 ROUND 269
 ROWID 271
 RTRIM 272
 SECOND 273
 SIGN 274
 SIN 275
 SINH 276
 SMALLINT 277
 SPACE 278
 SQRT 279
 STRIP 280
 SUBSTR 282
 TAN 284
 TANH 285
 TIME 286
 TIMESTAMP 287
 TIMESTAMP_FORMAT 289
 TO_CHAR 299
 TO_DATE 289
 TRANSLATE 290
 TRUNCATE 293
 UCASE 294
 UPPER 294
 VALUE 203
 VARCHAR 295
 VARCHAR_FORMAT 299
 VARGRAPHIC 301
 WEEK 304
 YEAR 305
 unqualified name 56
 function, user-defined 125

FUNCTIONS column
 SYSPACKAGE catalog table 967
 SYSPLAN catalog table 978

G

GBPCACHE clause
 ALTER INDEX statement 372
 ALTER TABLESPACE statement 419
 CREATE INDEX statement 533
 CREATE TABLESPACE statement 604
 GBPCACHE column
 SYSINDEXPART catalog table 959
 SYSTABLEPART catalog table 1009
 GENERATED clause
 ALTER TABLE statement 399
 CREATE TABLE statement 580
 DECLARE GLOBAL TEMPORARY TABLE
 statement 642
 GENERIC column of LUNAMES catalog table 925
 GET DIAGNOSTICS statement
 example 861
 SQL procedure 861
 GMT (Greenwich Mean Time) 105
 GO TO clause of WHENEVER statement 845
 GOTO statement
 example 862
 SQL procedure 862
 GRANT statement
 collection privileges 714
 database privileges 715
 description 711
 function privileges 720
 package privileges 725
 plan privileges 727
 procedure privileges 720
 schema privileges 728
 system privileges 730
 table privileges 733
 USAGE privilege 718
 use privileges 736
 view privileges 733
 GRANTEDTS column
 SYSCOLAUTH catalog table 930
 SYSDBAUTH catalog table 948
 SYSPLANAUTH catalog table 979
 SYSRESAUTH catalog table 987
 SYSROUTINEAUTH catalog table 988
 SYSSCHEMAAUTH catalog table 995
 SYSTABAUTH catalog table 1006
 SYSUSERAUTH catalog table 1021
 GRANTEE column
 SYSCOLAUTH catalog table 930
 SYSDBAUTH catalog table 947
 SYSPACKAUTH catalog table 968
 SYSPLANAUTH catalog table 979

GRANTEE column (*continued*)
 SYSRESAUTH catalog table 986
 SYSROUTINEAUTH catalog table 988
 SYSSCHEMAAUTH catalog table 995
 SYSTABAUTH catalog table 1004
 SYSUSERAUTH catalog table 1019
 GRANTEETYPE column
 SYSCOLAUTH catalog table 930
 SYSPACKAUTH catalog table 968
 SYSROUTINEAUTH catalog table 988
 SYSTABAUTH catalog table 1004
 GRANTOR column
 SYSCOLAUTH catalog table 930
 SYSDBAUTH catalog table 947
 SYSPACKAUTH catalog table 968
 SYSPLANAUTH catalog table 979
 SYSRESAUTH catalog table 986
 SYSROUTINEAUTH catalog table 988
 SYSSCHEMAAUTH catalog table 995
 SYSTABAUTH catalog table 1004
 SYSUSERAUTH catalog table 1019
 GRANULARITY column of SYSTRIGGERS catalog
 table 1018
 GRAPHIC
 data type
 CREATE TABLE statement 577
 description 70
 option of precompiler 68, 169
 GRAPHIC function 223
 graphic string
 constants 103
 description 70
 Greenwich Mean Time (GMT) 105
 GROUP BY clause
 cannot join view 631
 subselect
 description 321
 results 313
 GROUP_MEMBER column
 SYSCOPY catalog table 943
 SYSDATABASE catalog table 944
 SYSPACKAGE catalog table 965
 SYSPLAN catalog table 978
 grouping column 321

H

handler
 SQL procedure 857
 handling errors
 SQL procedure 857
 HAVING clause of subselect
 description 322
 results 313
 held connection state 35

HEX function 225
 hexadecimal constants 102
 HIGH2KEY column
 SYSCOLSTATS catalog table
 description 933
 SYSCOLUMNS catalog table
 description 935
 HIGHDSNUM column of SYSCOPY catalog table 943
 HIGHKEY column of SYSCOLSTATS catalog
 table 933
 HOLD LOCATOR statement
 description 738
 host identifier 50
 host structure
 description 123
 host variable
 colon 121
 description 120
 EXECUTE IMMEDIATE statement 692
 EXPLAIN statement 695
 FETCH statement 707
 input 120
 naming convention 52
 output 120
 PREPARE statement 758
 SELECT
 assignment 806
 substitution for parameter markers 689
 HOSTLANG column
 SYSDBRM catalog table 950
 SYSPACKAGE catalog table 964
 HOUR function 226

I

I/O processing
 CURRENT DEGREE special register 106
 IBM SQL 3
 IBMREQD column
 IPNAMES catalog table 920
 LOCATIONS catalog table 921
 LULIST catalog table 922
 LUMODES catalog table 923
 LUNAMES catalog table 925
 MODESELECT catalog table 926
 SYSAUXRELS catalog table 927
 SYSCHECKDEP catalog table 928
 SYSCHECKS catalog table 929
 SYSCOLAUTH catalog table 930
 SYSCOLDIST catalog table 931
 SYSCOLDISTSTATS catalog table 932
 SYSCOLSTATS catalog table 933
 SYSCOLUMNS catalog table 935
 SYSCONSTDEP catalog table 940
 SYSCOPY catalog table 941
 SYSDATABASE catalog table 944

IBMREQD column (*continued*)

- SYSDATATYPES catalog table 946
- SYSDBAUTH catalog table 948
- SYSDBRM catalog table 950
- SYSDUMMY1 catalog table 952
- SYSFIELDS catalog table 953
- SYSFOREIGNKEYS catalog table 954
- SYSINDEXES catalog table 956
- SYSINDEXPART catalog table 958
- SYSINDEXSTATS catalog table 960
- SYSKEYS catalog table 961
- SYSLOBSTATS catalog table 962
- SYSPACKAGE catalog table 965
- SYSPACKAUTH catalog table 968
- SYSPACKDEP catalog table 969
- SYSPACKLIST catalog table 970
- SYSPACKSTMT catalog table 971
- SYSPARMS catalog table 974
- SYSPKSYSTEM catalog table 975
- SYSPLAN catalog table 976
- SYSPLANAUTH catalog table 979
- SYSPLANDEP catalog table 980
- SYSPLSYSTEM catalog table 981
- SYSPROCEDURES catalog table 983
- SYSRELS catalog table 985
- SYSRESAUTH catalog table 986
- SYSROUTINEAUTH catalog table 988
- SYSROUTINES catalog table 993
- SYSSCHEMAAUTH catalog table 995
- SYSSEQUENCES catalog table 996
- SYSSEQUENCESDEP catalog table 997
- SYSSTMT catalog table 998
- SYSSTOGROUP catalog table 1000
- SYSSTRINGS catalog table 1001
- SYSSYNONYMS catalog table 1003
- SYSTABAUTH catalog table 1005
- SYSTABLEPART catalog table 1008
- SYSTABLES catalog table 1011
- SYSTABLESPACE catalog table 1015
- SYSTABSTATS catalog table 1017
- SYSTRIGGERS catalog table 1018
- SYSUSERAUTH catalog table 1020
- SYSVIEWDEP catalog table 1022
- SYSVIEWS catalog table 1023
- SYSVOLUMES catalog table 1024
- USERNAMES catalog table 1025

ICBACKUP column of SYSCOPY catalog table 942

ICDATE column of SYSCOPY catalog table 941

ICTIME column of SYSCOPY catalog table 942

ICTYPE column of SYSCOPY catalog table 941

ICUNIT column of SYSCOPY catalog table 942

identifier in SQL

- delimited 49
- long 50
- ordinary 48

identity column

- ALTER TABLE statement 400
- CREATE TABLE statement 581

IDENTITY_VAL_LOCAL function 227

IF statement

- example 864
- SQL procedure 864

IFNULL function 232

IMAGCOPY privilege

- GRANT statement 715
- REVOKE statement 779

IMAGCOPYAUTH column of SYSDBAUTH catalog table 948

IMPLICIT column of SYSTABLESPACE catalog table 1014

IN

- clause of CREATE AUXILIARY TABLE statement 460
- clause of CREATE PROCEDURE statement 544, 558
- clause of CREATE TABLE statement 588
- clause of CREATE TABLESPACE statement 600
- predicate 99, 155

IN EXCLUSIVE MODE clause of LOCK TABLE statement 752

IN SHARE MODE clause of LOCK TABLE statement 751

INCCSID column of SYSSTRINGS catalog table 1001

INCLUDE statement

- assembler declarations 886
- description 740
- SQLCA
 - C 886
 - COBOL 887
 - Fortran 887
- SQLDA
 - Assembler 902
 - C 903
 - C++ 903
 - COBOL 906
 - PL/I 888, 907

INCLUDING COLUMN DEFAULTS clause

DECLARE GLOBAL TEMPORARY TABLE statement 644

INCLUDING IDENTITY COLUMN ATTRIBUTES clause

DECLARE GLOBAL TEMPORARY TABLE statement 643, 644

INCREMENT column of SYSSEQUENCES catalog table 996

index

- altering
 - ALTER INDEX statement 364
- catalog table 911
- creating
 - CREATE INDEX statement 525
- description 22

- index (*continued*)
 - dropping 678
 - name, unqualified 56
 - naming convention 52
 - partitioning 534
 - primary 22
 - space
 - description 26
 - types
 - changing 364
 - unique 22
 - unqualified name 56
- INDEX clause
 - ALTER INDEX statement 364
 - CREATE INDEX statement 528
 - DROP statement 678
- INDEX privilege
 - GRANT statement 733
 - REVOKE statement 796
- INDEXAUTH column of SYSTABAUTH catalog table 1005
- INDEXBP
 - clause of ALTER DATABASE statement 348
 - clause of CREATE DATABASE statement 463
 - column of SYSDATABASE catalog table 945
- INDEXSPACE column of SYSINDEXES catalog table 955
- INDEXTYPE column
 - SYSINDEXES catalog table 956
- indicator array 123
- indicator variable
 - description 120
 - string expression 692
- infix operators 133
- INNER JOIN 319
 - example 324
 - FROM clause of subselect 319
- INOUT clause
 - CREATE PROCEDURE statement 544, 558
- input host variable 120
- INSERT clause of CREATE TRIGGER statement 617
- INSERT function 233
- INSERT privilege
 - GRANT statement 733
 - REVOKE statement 796
- insert rule 23, 745
- INSERT statement
 - description 742
- INSERTAUTH column of SYSTABAUTH catalog table 1005
- inserting
 - declaration in a program 740
 - rows in a table 742
- INSTS_PER_INVOC column
 - of SYSROUTINES catalog table 994
- INT function 236
- INTEGER
 - data type
 - CREATE TABLE statement 575
 - large 74
 - small 74
- integer constants 101
- INTEGER function 236
- integrated catalog facility
 - CREATE INDEX statement 532
 - identifier 51
- interactive SQL 20, 344
- INITIAL_INSTS column of SYSROUTINES catalog table 994
- INITIAL_IOS column of SYSROUTINES catalog table 994
- INTO clause
 - DESCRIBE CURSOR statement 666
 - DESCRIBE INPUT statement 668
 - DESCRIBE PROCEDURE statement 672
 - DESCRIBE statement 660
 - FETCH statement 707
 - INSERT statement 743
 - PREPARE statement 757
 - SELECT INTO statement 806
 - VALUES INTO statement 843
- invoke behavior 61
- IOS_PER_INVOC column
 - of SYSROUTINES catalog table 994
- IPADDR column of IPNAMES catalog table 920
- IPREFIX column
 - SYSINDEXPART catalog table 959
 - SYSTABLEPART catalog table 1009
- IS clause
 - COMMENT ON statement 442
 - LABEL ON statement 750
- ISO (International Standards Organization) 76
- ISOBID column of SYSINDEXES catalog table 955
- ISOLATION
 - column of SYSPACKAGE catalog table 964
 - column of SYSPACKSTMT catalog table 971
 - column of SYSPLAN catalog table 976
 - column of SYSSTMT catalog table 998
- isolation level
 - control by SQL statement
 - DELETE statement 656
 - INSERT statement 745
 - SELECT INTO statement 807
 - select-statement 334
- IXCREATOR column
 - SYSINDEXPART catalog table 958
 - SYSKEYS catalog table 961
 - SYSTABLEPART catalog table 1007
- IXNAME column
 - SYSINDEXPART catalog table 958
 - SYSKEYS catalog table 961

IXNAME column (*continued*)
SYSTABLEPART catalog table 1007
IXNAME column of SYSRELS catalog table 985
IXOWNER column of SYSRELS catalog table 985

J

Java Database Connectivity (JDBC) 20
JDBC (Java Database Connectivity) 20
JIS (Japanese Industrial Standard) 76
join operation
 example 324
 FROM clause of subselect 320
 FULL OUTER JOIN
 FROM clause of subselect 319
 INNER JOIN
 FROM clause of subselect 319
 joining tables 319
 LEFT OUTER JOIN
 FROM clause of subselect 319
 RIGHT OUTER JOIN
 FROM clause of subselect 319
 summary of results 320
JULIAN_DAY function 237

K

Katakana character 48
KATAKANA value for EBCDIC CODED CHAR SET 48
KEEPDYNAMIC column
 SYSPACKAGE catalog table 967
 SYSPLAN catalog table 978
key
 composite
 description 22
 description 22
 foreign
 description 23
 length
 maximum 870
 partitioning index 373, 534, 535, 836
 parent 23
 primary
 defining on a single column 578
 description 22
 unique 22
KEYCOLUMNS column of SYSTABLES catalog
 table 1011
KEYCOUNT column of SYSINDEXSTATS catalog
 table 960
KEYCOUNTF column of SYSINDEXSTATS catalog
 table 960
KEYOBID column of SYSTABLES catalog table 1011
KEYSEQ column of SYSCOLUMNS catalog table 937
keywords, reserved 1027

L

LABEL
 column of SYSCOLUMNS catalog table 938
 column of SYSTABLES catalog table 1011
LABEL ON statement 749
labeled duration 137
LABELS
 USING clause of DESCRIBE statement 660
 USING clause of PREPARE statement 758
LANGUAGE
 clause of ALTER FUNCTION statement 355
 clause of ALTER PROCEDURE statement 378
 clause of CREATE FUNCTION statement 481, 499
 clause of CREATE PROCEDURE statement 547,
 559
 column of SYSPROCEDURES catalog table 983
LANGUAGE column
 SYSROUTINES catalog table 989
LARGE clause
 CREATE TABLESPACE statement 599
large object (LOB) 71
LCASE function 238
LEAFDIST column of SYSINDEXPART catalog table
 description 958
LEAVE statement
 example 865
 SQL procedure 865
LEFT function 239
LEFT OUTER JOIN 319
 example 325
 FROM clause of subselect 319
LENGTH
 column of SYSCOLUMNS catalog table 934
 column of SYSFIELDS catalog table 953
 function 241
length attribute of column 69
LENGTH column
 SYSDATATYPES catalog table 946
 SYSPARMS catalog table 974
LENGTH2 column of SYSCOLUMNS catalog
 table 938
letter, description in DB2 47
library
 online 8
LIKE clause
 CREATE GLOBAL TEMPORARY TABLE
 statement 522
 CREATE TABLE statement 587
 DECLARE GLOBAL TEMPORARY TABLE
 statement 642
LIKE predicate 156
LIMITKEY column
 SYSINDEXPART catalog table 958
 SYSTABLEPART catalog table 1008

- limits, DB2 869
- LINK_OPTS
 - column of SYSPSMOPTS table 1048
- LINKAGE column of SYSPROCEDURES catalog table 982
- LINKNAME
 - USERNAMES catalog table 1025
- LINKNAME column
 - IPNAMES catalog table 920
 - LOCATIONS catalog table 921
 - LULIST catalog table 922
- literal 101
- LN function 242
- LOAD privilege
 - GRANT statement 716
 - REVOKE statement 779
- LOADAUTH column of SYSDBAUTH catalog table 948
- LOADMOD column
 - SYSPROCEDURES catalog table 982
- LOB
 - clause of CREATE TABLESPACE statement 599
- LOB (large object)
 - description 71
 - host variable 121
 - locator 71, 122
 - restrictions 72
- LOBCOLUMNS column of SYSROUTINES catalog table 992
- LOCAL 76
- local DB2 31
- locale
 - CURRENT LOCALE LC_CTYPE special register 107
- LOCATE function 243
- location
 - column of SYSPACKAGE catalog table 963
 - column of SYSPACKAUTH catalog table 968
 - column of SYSPACKSTMT catalog table 971
 - column of SYSPKSYSTEM catalog table 975
 - LOB 71
- LOCATION column
 - LOCATIONS catalog table 921
 - SYSPACKLIST catalog table 970
 - SYSTABLES catalog table 1012
- location identifier 50
- locator
 - LOB 71, 122
 - result set 123
- LOCATOR column of SYSPARMS catalog table 973
- locator variable
 - freeing 710
 - holding beyond a unit of work 738
- lock
 - ALTER TABLESPACE statement 414
 - CREATE TABLESPACE statement 608
- lock (*continued*)
 - description 28
 - during update 838
 - LOCK TABLE statement 751
 - object
 - table space (table) 751
- LOCK TABLE statement
 - description 751
- LOCKMAX clause
 - ALTER TABLESPACE statement description 415
 - CREATE TABLESPACE statement description 609
- LOCKMAX column
 - SYSTABLESPACE catalog table 1015
- LOCKPART
 - clause of ALTER TABLESPACE statement 420
 - clause of CREATE TABLESPACE statement 611
 - column of SYSTABLESPACE catalog table 1016
- LOCKRULE column of SYSTABLESPACE catalog table 1014
- LOCKSIZE clause
 - ALTER TABLESPACE statement description 414
 - CREATE TABLESPACE statement description 608
- LOG
 - clause of ALTER TABLESPACE statement 421
 - clause of CREATE TABLESPACE statement 606
 - column of SYSTABLESPACE catalog table 1016
- LOG function 242
- LOG10 function 245
- logical operator 163
- logical unit of work 28
 - See *also* unit of work
- long column string 69, 70
- long strings
 - use restrictions 72
- LONG VARCHAR data type
 - CREATE TABLE statement 573
 - description 69
- LONG VARGRAPHIC data type
 - CREATE TABLE statement 573
 - description 70
- LOOP statement
 - example 866
 - SQL procedure 866
- LOW2KEY column
 - SYSCOLSTATS catalog table 933
 - SYSCOLUMNS catalog table description 935
- LOWDSNUM column of SYSCOPY catalog table 943
- LOWER function 238
- lowercase character folded to uppercase 48
- LOWKEY column of SYSCOLSTATS catalog table 933

LTRIM function 246
 LUNAME
 column of LULIST catalog table 922
 column of LUMODES catalog table 923
 column of LUNAMES catalog table 924
 column of MODESELECT catalog table 926
 column of SYSPROCEDURES table 982

M

MAX function 182
 MAXASSIGNEDVAL column of SYSSEQUENCES catalog table 996
 MAXROWS
 clause of ALTER TABLESPACE statement 421
 clause of CREATE TABLESPACE statement 611
 column of SYSTABLESPACE catalog table 1016
 MAXVALUE column of SYSSEQUENCES catalog table 996
 MEMBER CLUSTER clause
 CREATE TABLESPACE statement 607
 message
 precompiler processing of DECLARE TABLE statement 651
 METATYPE column of SYSDATATYPES catalog table 946
 MICROSECOND function 247
 MIDNIGHT_SECONDS function 248
 MIN function 183
 MINUTE function 249
 MINVALUE column of SYSSEQUENCES catalog table 996
 MIXED column
 SYSDBRM catalog table 950
 SYSPACKAGE catalog table 964
 mixed data
 convention 5
 description 67
 in string assignments 90
 LIKE predicate 158
 MIXED DATA field of panel DSNTIPF 68, 169
 MIXED_CCSID column
 SYSDATABASE catalog table 945
 SYSTABLESPACE catalog table 1016
 MOD function 250
 MODE SQL clause of TRIGGER statement 619
 MODENAME column
 LUMODES catalog table 923
 MODESELECT catalog table 926
 MODESELECT column of LUNAMES catalog table 925
 MODIFIES SQL clause
 ALTER PROCEDURE statement 379
 MODIFIES SQL DATA clause
 ALTER FUNCTION statement 356
 ALTER PROCEDURE statement 386

MODIFIES SQL DATA clause (*continued*)
 CREATE FUNCTION statement 483
 CREATE PROCEDURE statement 549, 560
 MON1AUTH column of SYSUSERAUTH catalog table 1020
 MON2AUTH column of SYSUSERAUTH catalog table 1020
 MONITOR1 privilege
 GRANT statement 731
 REVOKE statement 794
 MONITOR2 privilege
 GRANT statement 731
 REVOKE statement 794
 MONTH function 252
 MONTHNAME function 1038
 MULTIPLY_ALT function 253

N

NACTIVE column
 SYSTABLESPACE catalog table description 1014
 SYSTABSTATS catalog table 1017
 NACTIVEF column of SYSTABLESPACE catalog table 1016
 NAME
 column of catalog table 998
 column of SYSCOLDIST catalog table 931
 column of SYSCOLDISTSTATS catalog table 932
 column of SYSCOLSTATS catalog table 933
 column of SYSCOLUMNS catalog table 934
 column of SYSDATABASE catalog table 944
 column of SYSDATATYPES catalog table 946
 column of SYSDBAUTH catalog table 947
 column of SYSDBRM catalog table 950
 column of SYSFIELDS catalog table 953
 column of SYSINDEXES catalog table 955
 column of SYSINDEXSTATS catalog table 960
 column of SYSLOBSTATS catalog table 962
 column of SYSPACKAGE catalog table 963
 column of SYSPACKAUTH catalog table 968
 column of SYSPACKLIST catalog table 970
 column of SYSPACKSTMT catalog table 971
 column of SYSPARMS catalog table 973
 column of SYSPKSYSTEM catalog table 975
 column of SYSPLAN catalog table 976
 column of SYSPLANAUTH catalog table 979
 column of SYSPLSYSTEM catalog table 981
 column of SYSRESAUTH catalog table 986
 column of SYSROUTINES catalog table 989
 column of SYSSEQUENCES catalog table 996
 column of SYSSTOGROUP catalog table 1000
 column of SYSSYNONYMS catalog table 1003
 column of SYSTABLES catalog table 1010
 column of SYSTABLESPACE catalog table 1014
 column of SYSTABSTATS catalog table 1017

NAME (*continued*)
 column of SYSTRIGGERS catalog table 1018
 column of SYSVIEWS catalog table 1023

NAMES
 USING clause of DESCRIBE statement 660
 USING clause of PREPARE statement 758

names, prepared SQL statements 649

naming convention
 SQL 50

NEARINDREF column of SYSTABLEPART catalog table 1007

NEAROFFPOSF column
 SYSINDEXPART catalog table 959

nested table expressions 316

NEW AS clause of TRIGGER statement 618

NEW TABLE AS clause of TRIGGER statement 618

NEW_TABLE AS clause of TRIGGER statement 616

NEWAUTHID column of USERNAMES catalog table 1025

NLEAF column
 SYSINDEXES catalog table
 description 955
 SYSINDEXSTATS catalog table 960

NLEVELS column
 SYSINDEXES catalog table
 description 955
 SYSINDEXSTATS catalog table 960

NO ACTION
 delete rule
 CREATE TABLE statement 585

NO ACTION delete rule
 description 23

NO CASCADE BEFORE clause of CREATE TRIGGER statement 617

NO COLLID clause
 ALTER FUNCTION statement 360
 CREATE FUNCTION statement 487, 504

NO DBINFO clause
 ALTER FUNCTION statement 360
 ALTER PROCEDURE statement 380
 CREATE FUNCTION statement 486, 504
 CREATE PROCEDURE statement 549, 560

NO EXTERNAL ACTION clause
 ALTER FUNCTION statement 357
 CREATE FUNCTION statement 483, 501

NO FINAL CALL clause
 ALTER FUNCTION statement 359, 485
 CREATE FUNCTION statement 503

NO SCRATCHPAD clause
 ALTER FUNCTION statement 358
 CREATE FUNCTION statement 484, 502

NO SQL clause
 ALTER FUNCTION statement 356
 ALTER PROCEDURE statement 379
 CREATE FUNCTION statement 483, 501
 CREATE PROCEDURE statement 549

NO WLM ENVIRONMENT clause
 ALTER PROCEDURE statement 381, 387
 CREATE PROCEDURE statement 550, 561

NOCOLLID clause
 ALTER PROCEDURE statement 380, 386
 CREATE PROCEDURE statement 550, 560

NOFOR option
 precompiler 172

NOGRAPHIC option of precompiler 169

nonexecutable statement 340, 341

NOT DETERMINISTIC clause
 ALTER FUNCTION statement 356
 ALTER PROCEDURE statement 379, 385
 CREATE FUNCTION statement 482, 500
 CREATE PROCEDURE statement 549, 559

NOT FOUND clause of WHENEVER statement 845

NOT NULL CALL clause
 ALTER FUNCTION statement 352
 CREATE FUNCTION statement 474, 493

NOT NULL clause
 ALTER TABLE statement 397
 CREATE GLOBAL TEMPORARY TABLE statement
 description 522
 CREATE TABLE statement
 description 578
 DECLARE GLOBAL TEMPORARY TABLE statement
 description 641

NOT VARIANT clause
 ALTER PROCEDURE statement 377, 385
 CREATE FUNCTION statement 474, 493
 CREATE PROCEDURE statement 542, 556

notices, legal 1051

NPAGES column
 SYSTABLES catalog table
 description 1010
 SYSTABSTATS catalog table 1017

NTABLES column of SYSTABLESPACE catalog table 1014

NULL
 CAST specification 146
 predicate 161

NULL CALL clause
 ALTER FUNCTION statement 352
 CREATE FUNCTION statement 474, 493

null value
 assigned to host variable 806
 assignment 85
 description 66
 duplicate rows 312
 grouping columns 321
 result columns 314
 specified by indicator variable 120

NULL_CALL column of SYSROUTINES catalog table 990

NULLIF function 254

- NULLS column of SYSCOLUMNS catalog table 935
- numbers in SQL 73
- NUMCOLUMNS
 - SYSCOLDIST catalog table 931
 - SYSCOLDISTSTATS catalog table 932
- numeric
 - assignments 86
 - comparisons 94
 - conversion errors 807
 - data type 73
- NUMERIC data type
 - CREATE TABLE statement 575
 - description 74
- NUMPARTS
 - clause of CREATE TABLESPACE statement 608

O

- OBID
 - clause of CREATE TABLE statement 589
 - column of SYSCHECKS catalog table 929
 - column of SYSINDEXES catalog table 955
 - column of SYSTABLES catalog table 1010
 - column of SYSTABLESPACE catalog table 1014
 - column of SYSTRIGGERS catalog table 1018
- object name, unqualified 56
- object table 114
- OBTYPE column of SYSRESAUTH catalog table 986
- ODBC (Open Database Connectivity) 20
- OLD AS clause of TRIGGER statement 618
- OLD TABLE AS clause of TRIGGER statement 618
- OLD_TABLE AS clause of TRIGGER statement 616
- ON clause
 - CREATE INDEX statement 528
 - CREATE TRIGGER statement 617
 - joining tables 319
- ON COMMIT ROWS clause
 - DECLARE GLOBAL TEMPORARY TABLE statement 645
- ON DELETE clause
 - ALTER TABLE statement 406
 - CREATE TABLE statement 585
- ON ROLLBACK RETAIN CURSORS clause
 - SAVEPOINT statement 804
- ON ROLLBACK RETAIN LOCKS clause
 - SAVEPOINT statement 805
- ON TABLE clause
 - GRANT statement 734
 - REVOKE statement 797
- online books 8
- OPEN
 - statement
 - description 753
- open cursor 709
- Open Database Connectivity (ODBC) 20

- operands
 - datetime 137
 - decimal 134
 - floating-point 137
 - integer 134
- operation
 - SQL
 - assignment 84
 - comparison 94
 - description 84
- OPERATIVE column
 - SYSPACKAGE catalog table 963
 - SYSPLAN catalog table 976
- operator
 - arithmetic 133
- OPTHINT column
 - SYSPACKAGE catalog table 967
 - SYSPLAN catalog table 978
- OPTIMIZE FOR n ROWS clause 334
- OR truth table 163
- ORDER BY clause
 - select-statement 331
- ORDER column of SYSSEQUENCES catalog table 996
- order of evaluation, operators 143
- order of statements
 - in a compound statement 858
- ORDERING column of SYSKEYS catalog table 961
- ORDINAL column of SYSPARMS catalog table 973
- ordinary identifier in SQL 48
- ORGRATIO column of SYSLOBSTATS catalog table 962
- ORIGIN column of SYSROUTINES catalog table 989
- OTYPE column of SYSCOPY catalog table 943
- OUT clause of CREATE PROCEDURE statement 544, 558
- OUTCCSID column of SYSSTRINGS catalog table 1001
- outer join 319
 - example 325
 - FULL OUTER JOIN 325
 - FROM clause of subselect 319
 - LEFT OUTER JOIN 325
 - FROM clause of subselect 319
 - RIGHT OUTER JOIN 325
 - FROM clause of subselect 319
- output host variable 120
- OVERRIDING USER VALUE
 - clause of INSERT statement 744
- OWNER
 - column of SYSDATATYPES catalog table 946
 - column of SYSINDEXSTATS catalog table 960
 - column of SYSPACKAGE catalog table 963
 - column of SYSPARMS catalog table 973
 - column of SYSROUTINES catalog table 989
 - column of SYSSEQUENCES catalog table 996

OWNER (*continued*)
column of SYSTABSTATS catalog table 1017
column of SYSTRIGGERS catalog table 1018

P

PACKADM authority

GRANT statement 714
REVOKE statement 777

package

binding

remote 63

description 31

dropping 679

invalidated

ALTER TABLE statement 409

privileges 725, 788

remote bind 63

PACKAGE

clause of DROP statement 679

clause of GRANT statement 725

clause of REVOKE statement 788

page set

description 26

PAGESAVE column of SYSTABLEPART catalog table

description 1008

PARALLEL column of SYSROUTINES catalog table 990

parallel processing

SET CURRENT DEGREE statement 812

parameter

passing to stored procedure 432, 851

parameter marker

CAST specification 146

description 759

EXECUTE statement 689

EXPLAIN statement 695

host variables in dynamic SQL 121

obtaining information with DESCRIBE INPUT 668

OPEN statement 754

PREPARE statement 759

rules 759

PARAMETER STYLE clause

ALTER PROCEDURE statement 378

CREATE FUNCTION statement 481, 499

CREATE PROCEDURE statement 548

PARAMETER_STYLE column of SYSROUTINES catalog table 991

parent key

description 23

parent row 23

parent table 23

PARENTS column of SYSTABLES catalog table 1011

PARAM_COUNT column of SYSROUTINES catalog table 989

PARAM_SIGNATURE column of SYSROUTINES catalog table 994

PARAM1 column of SYSROUTINES catalog table 993

PARAM10 column of SYSROUTINES catalog table 993

PARAM11 column of SYSROUTINES catalog table 993

PARAM12 column of SYSROUTINES catalog table 993

PARAM13 column of SYSROUTINES catalog table 993

PARAM14 column of SYSROUTINES catalog table 993

PARAM15 column of SYSROUTINES catalog table 993

PARAM16 column of SYSROUTINES catalog table 993

PARAM17 column of SYSROUTINES catalog table 993

PARAM18 column of SYSROUTINES catalog table 993

PARAM19 column of SYSROUTINES catalog table 993

PARAM2 column of SYSROUTINES catalog table 993

PARAM20 column of SYSROUTINES catalog table 993

PARAM21 column of SYSROUTINES catalog table 993

PARAM22 column of SYSROUTINES catalog table 994

PARAM23 column of SYSROUTINES catalog table 994

PARAM24 column of SYSROUTINES catalog table 994

PARAM25 column of SYSROUTINES catalog table 994

PARAM26 column of SYSROUTINES catalog table 994

PARAM27 column of SYSROUTINES catalog table 994

PARAM28 column of SYSROUTINES catalog table 994

PARAM29 column of SYSROUTINES catalog table 994

PARAM3 column of SYSROUTINES catalog table 993

PARAM30 column of SYSROUTINES catalog table 994

PARAM4 column of SYSROUTINES catalog table 993

PARAM5 column of SYSROUTINES catalog table 993

PARAM6 column of SYSROUTINES catalog table 993

PARAM7 column of SYSROUTINES catalog table 993

PARAM8 column of SYSROUTINES catalog table 993

PARAM9 column of SYSROUTINES catalog table 993

PARMLIST column

SYSFIELDS catalog table 953

SYS PROCEDURES catalog table 983

PARAMNAME column of SYSPARMS catalog table 973

PART

clause of ALTER INDEX statement 372

clause of ALTER TABLESPACE statement 415

clause of CREATE AUXILIARY TABLE statement 461

clause of CREATE INDEX statement 534

clause of CREATE TABLESPACE statement 608

clause of LOCK TABLE statement 751

PARTITION column

SYSAUXRELS catalog table 927

SYSCOLDISTSTATS catalog table 932

SYSCOLSTATS catalog table 933

SYSINDEXPART catalog table 958

SYSINDEXSTATS catalog table 960

SYSTABLEPART catalog table 1007

SYSTABLESPACE catalog table 1014

SYSTABSTATS catalog table 1017

PASSWORD

column of USERNAMES catalog table 1025

PATHSCHEMAS column
 SYSPACKAGE catalog table 967
 SYSPLAN catalog table 978
 SYSVIEWS catalog table 1023

PCTFREE
 clause of ALTER INDEX statement 371
 clause of ALTER TABLESPACE statement 416
 clause of CREATE INDEX statement 533
 clause of CREATE TABLESPACE statement 604
 column of SYSINDEXPART catalog table 958
 column of SYSTABLEPART catalog table 1008

PCTIMESTAMP column of SYSPACKAGE catalog table 965

PCTPAGES column
 SYSTABLES catalog table 1010
 SYSTABSTATS catalog table 1017

PCTROWCOMP column
 SYSTABLES catalog table
 description 1012
 SYSTABSTATS catalog table 1017

PDSNAME column
 SYSDBRM catalog table 950
 SYSPACKAGE catalog table 965

PERACTIVE column of SYSTABLEPART catalog table
 description 1007

PERCDROP column of SYSTABLEPART catalog table
 description 1007

PERIOD option of precompiler 166

PGM_TYPE column of SYSPROCEDURES catalog table 983

PGSIZE column
 SYSINDEXES catalog table 956
 SYSTABLESPACE catalog table 1014

PIECESIZE clause
 ALTER INDEX statement 366
 CREATE INDEX statement 536
 effect on index 366

PIECESIZE column of SYSINDEXES catalog table 957

PIT_RBA column of SYSCOPY catalog table 943

PKSIZE column of SYSPACKAGE catalog table 963

PL/I application program
 host structure 123
 host variable
 description 120
 INCLUDE SQLCA 888
 INCLUDE SQLDA 907
 varying-length string 69

PLAN
 clause of EXPLAIN statement 694
 clause of GRANT statement 727
 clause of REVOKE statement 790

plan element 31, 817
 plan table 694

PLAN_TABLE table
 EXPLAIN statement 695

plan, application 409

PLANNAME column
 SYSIBM.MODESELECT catalog table 926
 SYSPACKLIST catalog table 970

PLCREATOR column
 SYSDBRM catalog table 950
 SYSSTMT catalog table 998

PLENTRIES column of SYSPLAN catalog table 977

PLNAME column
 SYSDBRM catalog table 950
 SYSSTMT catalog table 998

PLSIZE column of SYSPLAN catalog table 976

point of consistency
 description 28

PORT column of LOCATIONS catalog table 921

POSSTR function 255

POWER function 257

PQTY column
 SYSINDEXPART catalog table 958
 SYSTABLEPART catalog table 1007

precedence of operators 143

precision of numbers
 description 73
 determined by SQLLEN variable 899
 in assignments 86
 in comparisons 94
 results of arithmetic operations 133
 values for data types 73

PRECOMPDATE column of SYSDBRM catalog table 950

PRECOMPILE_OPTS
 column of SYSPSMOPTS table 1048

precompiler
 checks SQL statements 650
 DECLARE TABLE statement 650
 escape character 49
 options
 COBOL decimal point 166
 CONNECT 164
 date 170
 NOFOR 172
 STDSQL 170
 string delimiter 168
 time 170
 using INCLUDE statements 740

PRECOMPTIME column of SYSDBRM catalog table 950

PRECOMPTS column
 SYSDBRM catalog table 951

predicate
 basic 150
 BETWEEN 153
 description 150
 EXISTS 153

predicate (*continued*)
 IN 155
 LIKE 156
 NULL 161
 quantified 151
 prefix operator 133
 PRELINK_OPTS
 column of SYSPSMOPTS table 1048
 PREPARE statement
 description 757
 prepared SQL statement
 dynamically prepared by PREPARE 757
 executing 689
 identifying by DECLARE 649
 obtaining information with DESCRIBE 659
 obtaining information with DESCRIBE INPUT 668
 SQLDA provides information 890
 statements allowed 873
 primary index 22
 See *also* index, primary
 primary key 22
 See *also* key, primary
 PRIMARY KEY clause
 ALTER TABLE statement 402
 CREATE TABLE statement
 description 578, 583
 PRIQTY clause
 ALTER INDEX statement 368
 ALTER TABLESPACE statement 417
 CREATE INDEX statement 530
 CREATE TABLESPACE statement 601
 privilege
 granting 711
 revoking 772
 types 711
 PRIVILEGE column of SYSCOLAUTH catalog
 table 930
 PROCEDURE
 clause of COMMENT ON statement 442
 clause of DROP statement 679
 column of SYSPROCEDURES catalog table 982
 procedure, stored
 naming convention 53
 PROCEDURENAME
 column of SYSPSM table 1047
 column of SYSPSMOPTS table 1048
 process
 description 28
 PROGRAM TYPE clause
 ALTER FUNCTION statement 361
 ALTER PROCEDURE statement 382, 387
 CREATE FUNCTION statement 488, 506
 CREATE PROCEDURE statement 551, 562
 PROGRAM_TYPE column of SYSROUTINES catalog
 table 992
 promotion
 data types 81
 PSID column of SYSTABLESPACE catalog table 1014
 PSMDATE
 column of SYSPSM table 1047
 PSMTIME
 column of SYSPSM table 1047
 PUBLIC AT ALL LOCATIONS clause
 GRANT statement 712
 REVOKE statement 773
 PUBLIC clause
 GRANT statement 712
 REVOKE statement 773

Q

qualification of column names 114
 QUALIFIER
 column of SYSPACKAGE catalog table 963
 column of SYSPLAN catalog table 977
 column of SYSRESAUTH catalog table 986
 unqualified object names 56
 quantified predicate 151
 QUARTER function 258
 query 309
 QUERYNO clause 335
 DELETE statement 656, 745, 808, 838
 question mark (?) 689
 quotation mark 49, 168
 QUOTE
 column of SYSDBRM catalog table 950
 column of SYSPACKAGE catalog table 964
 option of precompiler 168
 QUOTESQL option of precompiler 168

R

RACF (Resource Access Control Facility)
 security for remote execution 65
 RADIANS function 259
 RAISE_ERROR function 260
 RAND function 261
 RBA column of SYSCHECKS catalog table 929
 RBA1 column of SYSTABLES catalog table 1012
 RBA2 column of SYSTABLES catalog table 1012
 READ SQL clause
 ALTER PROCEDURE statement 379
 READ SQL DATA clause
 ALTER FUNCTION statement 356
 read-only
 FOR FETCH ONLY clause 333
 FOR READ ONLY clause 333
 result table 636
 view 631
 READS SQL DATA clause
 ALTER PROCEDURE statement 386

READS SQL DATA clause (*continued*)
 CREATE FUNCTION statement 483, 501
 CREATE PROCEDURE statement 549, 560
 REAL data type
 CREATE TABLE statement 575
 description 74
 REAL function 262
 RECLENGTH column of SYSTABLES catalog table 1011
 RECOVER privilege
 GRANT statement 731
 REVOKE statement 794
 RECOVERAUTH column of SYSUSERAUTH catalog table 1020
 RECOVERDB privilege
 GRANT statement 716
 REVOKE statement 779
 RECOVERDBAUTH column of SYSDBAUTH catalog table 948
 recovery 444
 description
 restoring data consistency 28
 REFCOLS column of SYSTABAUTH catalog table 1006
 REFERENCES clause
 ALTER TABLE statement 405
 CREATE TABLE statement 584
 REFERENCES privilege
 GRANT statement 734
 REVOKE statement 796
 REFERENCESAUTH column of SYSTABAUTH catalog table 1005
 REFERENCING clause of TRIGGER statement 618
 referential constraint
 ALTER TABLE statement 404
 CREATE TABLE statement 584
 description 23
 referential cycle 23
 referential integrity
 description 23
 REFTBCREATOR column of SYSRELS catalog table 985
 REFTBNAME column of SYSRELS catalog table 985
 RELEASE
 column of SYSPACKAGE catalog table 964
 column of SYSPLAN catalog table 977
 RELEASE (connection)
 statement 765
 release level identification, current server 450, 455
 release pending connection state 35
 RELEASE SAVEPOINT statement
 description 768
 RELNAME column
 SYSPROCEDURES catalog table 954
 SYSRELS catalog table 985
 RELOBID1 column of SYSRELS catalog table 985
 RELOBID2 column of SYSRELS catalog table 985
 REMARKS column
 SYSCOLUMNS catalog table 936
 SYSDATATYPES catalog table 946
 SYSROUTINES catalog table 994
 SYSSEQUENCES catalog table 996
 SYSTABLES catalog table 1011
 SYSTRIGGERS catalog table 1018
 REMOTE
 column of SYSPACKAGE catalog table 965
 Remote Recovery Data Facility (RRDF) 407, 590
 remote unit of work 31
 See also DRDA access
 REMOVE VOLUMES clause of ALTER STOGROUP statement 391
 RENAME statement 769
 REOPTVAR
 column of SYSPACKAGE catalog table 966
 column of SYSPLAN catalog table 978
 REORG privilege
 GRANT statement 716
 REVOKE statement 779
 REORGAUTH column of SYSDBAUTH catalog table 948
 REPAIR privilege
 GRANT statement 716
 REVOKE statement 779
 REPAIRAUTH column of SYSDBAUTH catalog table 948
 REPEAT function 263
 REPEAT statement
 example 867
 SQL procedure 867
 REPLACE function 265
 reserved keywords 1027
 RESET
 clause of CONNECT statement 455
 RESTRICT
 delete rule
 ALTER TABLE statement 406
 CREATE TABLE statement 585
 description 23
 RESTRICT clause of REVOKE statement 773, 781
 result column
 data type 314
 names 314
 result set locator 123
 result table
 description 21
 RESULT_COLS column of SYSROUTINES catalog table 994
 RESULT_SETS column
 SYSPROCEDURES catalog table 983
 SYSROUTINES catalog table 992

RETURN_TYPE column of SYSROUTINES catalog table 989
 RETURNS clause of CREATE FUNCTION statement 479, 514
 RETURNS NULL ON NULL INPUT clause
 ALTER FUNCTION statement 356
 CREATE FUNCTION statement 482, 500
 RETURNS TABLE clause of CREATE FUNCTION statement 498
 REVOKE statement
 cascading effect 773
 collection privileges 777
 database privileges 778
 description 772
 distinct type privileges 781
 function privileges 783
 package privileges 788
 plan privileges 790
 procedure privileges 783
 schema privileges 791
 system privileges 793
 table privileges 796
 use privileges 799
 view privileges 796
 RIGHT function 267
 RIGHT OUTER JOIN 319
 example 325
 FROM clause of subselect 319
 rollback
 description 28
 ROLLBACK statement
 description 801
 ROUND function 269
 ROUTINEID column
 SYSPARMS catalog table 973
 SYSROUTINES catalog table 989
 ROUTINETYPE column
 SYSPARMS catalog table 973
 SYSROUTINEAUTH catalog table 988
 SYSROUTINES catalog table 989
 row
 deleting 653
 description 21
 inserting 742
 selecting single row 806
 updating 833
 row ID
 assignment of values 92
 comparison of values 96
 row ID data type 78, 577
 ROWID
 data type
 CREATE TABLE statement 577
 description 78
 ROWID function 271

ROWTYPE column of SYSPARMS catalog table 973
 RRDF (Remote Recovery Data Facility)
 altering a table for 407
 creating a table for 590
 RTRIM function 272
 run behavior 61
 RUN OPTIONS clause
 ALTER FUNCTION statement 362
 ALTER PROCEDURE statement 382, 388
 CREATE FUNCTION statement 489, 506
 CREATE PROCEDURE statement 552, 562
 RUNOPTS column
 SYSPROCEDURES catalog table 983
 SYSROUTINES catalog table 994

S

sample table
 description 21
 sample user-defined functions 1029
 savepoint
 releasing 768
 setting 804
 savepoint identifier
 naming convention 54
 savepoint name
 naming convention 54
 SAVEPOINT statement
 description 804
 SBCS data
 description 67
 SBCS site 68
 SBCS_CCSID column
 SYSDATABASE catalog table 944
 SYSTABLESPACE catalog table 1016
 scalar function 187
 SCALE column
 SYSCOLUMNS catalog table 935
 SYSDATATYPES catalog table 946
 SYSFIELDS catalog table 953
 SYSPARMS catalog table 974
 scale of numbers
 assignments 87
 comparisons 94
 description 74
 results of arithmetic operations 135
 schema
 description 20
 privileges 728, 791
 SCHEMA column
 SYSDATATYPES catalog table 946
 SYSPARMS catalog table 973
 SYSPSM table 1047
 SYSPSMOPTS table 1048
 SYSROUTINEAUTH catalog table 988
 SYSROUTINES catalog table 989

SCHEMA column (*continued*)
 SYSSEQUENCES catalog table 996
 SYSTRIGGERS catalog table 1018
 schema name
 naming convention 53
 SCHEMANAME column
 SYSSCHEMAAUTH catalog table 995
 SCRATCHPAD clause
 ALTER FUNCTION statement 358
 CREATE FUNCTION statement 484, 502
 SCRATCHPAD column of SYSROUTINES catalog table 990
 SCRATCHPAD_LENGTH column of SYSROUTINES catalog table 990
 SCREATOR column of SYSTABAUTH catalog table 1004
 search condition
 DELETE statement 655
 description 163
 HAVING 322
 order of evaluation 163
 UPDATE statement 837
 WHERE clause 320
 SECOND function 273
 SECQTY
 clause of ALTER INDEX statement 369
 clause of ALTER TABLESPACE statement 418
 clause of CREATE INDEX statement 531
 clause of CREATE TABLESPACE statement 602
 SECQTYI column
 SYSINDEXPART catalog table 959
 SYSTABLEPART catalog table 1009
 SECTNO column
 SYSPACKSTMT catalog table 971
 SYSSTMT catalog table 998
 SECTNOI column
 SYSPACKSTMT catalog table 972
 SYSSTMT catalog table 999
 SECURITY clause
 ALTER FUNCTION statement 361
 ALTER PROCEDURE statement 382, 388
 CREATE FUNCTION statement 488, 506
 CREATE PROCEDURE statement 551, 562
 SECURITY_IN column of LUNAMES catalog table 924
 SECURITY_OUT column
 IPNAMES catalog table 920
 LUNAMES catalog table 924
 SEGSIZE
 clause of CREATE TABLESPACE statement 610
 column of SYSTABLESPACE catalog table 1015
 SELECT
 clause as syntax component 312
 SELECT INTO statement 806
 SELECT privilege
 GRANT statement 734
 REVOKE statement 796
 SELECT statement
 description 331
 dynamic invocation 343
 example 335
 fullselect 327
 list
 application 313
 description 312
 maximum number of elements 870
 notation 312
 static invocation 343
 subselect 311
 SELECTAUTH column of SYSTABAUTH catalog table 1005
 selecting
 single row 806
 self-referencing constraint 23
 self-referencing row 23
 self-referencing table 23
 SEQNO
 column of SYSPSM table 1047
 SEQNO column
 SYSPACKLIST catalog table 970
 SYSPACKSTMT catalog table 971
 SYSSTMT catalog table 998
 SYSTRIGGERS catalog table 1018
 SYSVIEWS catalog table 1023
 SEQTYPE column of SYSSEQUENCES catalog table 996
 SEQUENCEID column of SYSSEQUENCES catalog table 996
 server
 accessible 32
 current 33
 establishing with CONNECT 448
 remote 31
 SET clause of UPDATE statement 836
 SET CONNECTION statement 809
 SET CURRENT DEGREE statement 812
 SET CURRENT LOCALE LC_CTYPE statement 814
 SET CURRENT OPTIMIZATION HINT statement 816
 SET CURRENT PACKAGESET statement 817
 SET CURRENT PATH statement 819
 SET CURRENT PRECISION statement 822
 SET CURRENT RULES statement 823
 SET CURRENT SQLID statement 824
 SET host-variable assignment statement 826
 SET NULL delete rule
 ALTER TABLE statement 406
 CREATE TABLE statement 585
 description 23
 SET QUERYNO clause of EXPLAIN statement 695
 SET transition-variable assignment statement 828
 SGCREATOR column of SYSVOLUMES catalog table 1024

SGNAMES column of SYSVOLUMES catalog table 1024
 SHARE
 option of LOCK TABLE statement 751
 shift-in character
 convention 5
 LABEL ON statement 750
 not truncated by assignments 90
 shift-out character
 convention 5
 LABEL ON statement 750
 short identifier in SQL 50
 short string column 69, 70
 SHRLEVEL
 column of SYSCOPY catalog table 942
 SIGN function 274
 sign-on exit routine
 CURRENT SQLID special register 61, 111
 SIGNAL SQLSTATE statement 831
 SIN function 275
 single precision floating-point number 74
 SINH function 276
 SMALLINT function 277
 softcopy publications 8
 SOME quantified predicate 151
 SOURCE clause of CREATE FUNCTION statement 515
 sourced function 125
 SOURCEDSN
 column of SYSPSMOPTS table 1048
 SOURCESCHEMA column
 SYSDATATYPES catalog table 946
 SYSROUTINES catalog table 989
 SOURCESPECIFIC column of SYSROUTINES catalog table 989
 SOURCECETYPE column of SYSDATATYPES catalog table 946
 SOURCECETYPEID column
 DATATYPES catalog table 946
 SYSCOLUMNS catalog table 939
 SYSPARMS catalog table 973
 SYSSEQUENCES catalog table 996
 space character 48
 SPACE column
 SYSINDEXES catalog table 956
 SYSINDEXPART catalog table 958
 SYSSTOGROUP catalog table 1000
 SYSTABLEPART catalog table 1008
 SYSTABLESPACE catalog table 1015
 SPACE function 278
 SPCDATE column of SYSSTOGROUP catalog table 1000
 special character 47
 special register
 CURRENT DATE 106
 CURRENT DEGREE 106
 special register (*continued*)
 CURRENT LOCALE LC_CTYPE 107
 CURRENT OPTIMIZATION HINT 107
 CURRENT PACKAGESET 108
 CURRENT PATH 108
 CURRENT PRECISION 109
 CURRENT RULES 109
 CURRENT SERVER 111
 CURRENT SQLID 111
 CURRENT TIME 112
 CURRENT TIMESTAMP 112
 CURRENT TIMEZONE 112
 CURRENT_DATE 106
 CURRENT_TIME 112
 CURRENT_TIMESTAMP 112
 description 104
 USER 113
 values in trigger 623
 SPECIFIC clause
 CREATE FUNCTION statement 480, 498, 514
 SPECIFIC FUNCTION clause of ALTER FUNCTION statement 355
 specific name
 naming convention 53
 unqualified name 56
 SPECIFICNAME column
 SYSPARMS catalog table 973
 SYSROUTINEAUTH catalog table 988
 SYSROUTINES catalog table 989
 SQL (Structured Query Language)
 assignment operation 84
 character 47
 comparison operation 84
 constants 101
 data types
 binary strings 71
 casting 83
 character strings 67
 datetime 75
 description 66
 graphic strings 70
 LOBs (large objects) 71
 numbers 73
 promotion 81
 results of an operation 99
 row ID 78
 deferred embedded 19
 delimited identifier 49
 description 19
 dynamic
 description 19
 statements allowed 873
 identifier 48
 interactive 20
 Java Database Connectivity (JDBC) 20
 JDBC (Java Database Connectivity) 20

SQL (Structured Query Language) *(continued)*

- keywords, reserved 1027
- limits 869
- naming conventions 50
- null value 66
- ODBC (Open Database Connectivity) 20
- Open Database Connectivity (ODBC) 20
- ordinary identifier 47
- rules 109
- SQLJ 20
- standard 4, 170
- static
 - description 19
- token 47
- value 66
- variable names 50

SQL path 57, 127

SQL procedure

- statements allowed 879

SQL procedure statement

- assignment statement 849
- CALL statement 851
- CASE statement 853
- compound statement 855
- CONTINUE handler 858
- EXIT handler 858
- GET DIAGNOSTICS statement 861
- GOTO statement 862
- handler 857
- handling errors 857
- IF statement 864
- LEAVE statement 865
- LOOP statement 866
- order of statements 858
- REPEAT statement 867
- WHILE statement 868

SQL return code

- See SQLCODE

SQL statements

- ALLOCATE CURSOR 346
- ALTER DATABASE 348
- ALTER FUNCTION 351
- ALTER INDEX 364
- ALTER PROCEDURE 376, 384
- ALTER STOGROUP 390
- ALTER TABLE 393
- ALTER TABLESPACE
 - description 412
- ASSOCIATE LOCATORS 424
- BEGIN DECLARE SECTION 427
- CALL
 - description 428
- catalog table restrictions 915
- CLOSE 436
- COMMENT ON 438
- COMMIT 444

SQL statements *(continued)*

- CONNECT (Type 1) 449
- CONNECT (Type 2) 454
- CONNECT differences 446
- CONTINUE 845
- CREATE ALIAS 457
- CREATE AUXILIARY TABLE 459
- CREATE DATABASE 462
- CREATE DISTINCT TYPE 465
- CREATE FUNCTION 472
- CREATE FUNCTION (external scalar) 473
- CREATE FUNCTION (external table) 492
- CREATE FUNCTION (sourced) 508
- CREATE GLOBAL TEMPORARY TABLE 520
- CREATE INDEX 525
- CREATE PROCEDURE 541, 555
- CREATE STOGROUP 565
- CREATE SYNONYM 568
- CREATE TABLE 570
- CREATE TABLESPACE
 - description 597
- CREATE TRIGGER 615
- CREATE VIEW 627
- DECLARE CURSOR
 - description 634
- DECLARE GLOBAL TEMPORARY TABLE 639
- DECLARE STATEMENT 649
- DECLARE TABLE 650
- DELETE
 - description 653
- DESCRIBE 659
- DESCRIBE CURSOR 666
- DESCRIBE INPUT 668
- DESCRIBE PROCEDURE 671
- DROP
 - description 674
- END DECLARE SECTION 687
- EXECUTE 689
- EXECUTE IMMEDIATE 692
- EXPLAIN
 - description 694
- FETCH
 - description 707
- FOR 695
- FREE LOCATOR 710
- GRANT 711
- HOLD LOCATOR 738
- INCLUDE
 - description 740
 - SQLCA 887
 - SQLDA 902
- INSERT
 - description 742
- invocation 340
- LABEL ON 749
- LOCALE LC_CTYPE 814

SQL statements (*continued*)

- LOCK TABLE 751
- OPEN
 - description 753
- PREPARE 757
- RELEASE 765
- RELEASE SAVEPOINT 768
- remote execution
 - description 63
 - dynamic execution 64
 - static execution 64
- RENAME 769
- REVOKE 772
- ROLLBACK 801
- SAVEPOINT 804
- SELECT INTO 806
- SET CONNECTION 809
- SET CURRENT DEGREE 812
- SET CURRENT OPTIMIZATION HINT 816
- SET CURRENT PATH 819
- SET CURRENT PRECISION 822
- SET CURRENT RULES 823
- SET CURRENT SQLID 824
- SET host-variable assignment 826
- SET transition-variable assignment 828
- SIGNAL SQLSTATE 831
- UPDATE
 - description 833
- VALUES 842
- VALUES INTO 843
- WHENEVER 845
- SQL variable name
 - naming convention 54
- SQL_DATA_ACCESS column of SYSROUTINES catalog table 991
- SQLCA (SQL communication area)
 - contents 883
 - entry changed by UPDATE 838
 - INCLUDE statement 740
- SQLCABC field of SQLCA 883
- SQLCAID field of SQLCA 883
- SQLCODE 451
 - 752 451
 - 900 452
 - 918 451
 - +100 344, 707, 745, 753, 806, 845
 - description 344
 - field of SQLCA 883
- SQLD field of SQLDA 663, 892
- SQLDA (SQL descriptor area)
 - clause of INCLUDE statement 740
 - contents 890, 891
- SQLDABC field of SQLDA 663, 891
- SQLDAID field of SQLDA 662, 891
- SQLDATA field of SQLDA 664, 896
- SQLDATAL field of SQLDA 898
- SQLDATALEN field of SQLDA 898
- SQLDATATYPE field of SQLDA 664
- SQLDATATYPE-NAME field of SQLDA 899
- SQLERRD(3) field of SQLCA 657
- SQLERRD(n) field of SQLCA 883
- SQLERRMC field of SQLCA 883
- SQLERRML field of SQLCA 883
- SQLERROR
 - clause of WHENEVER statement 845
 - column of SYSPACKAGE catalog table 965
- SQLERP field of SQLCA 883
- SQLIND field of SQLDA 664, 896
- SQLJ 20
- SQLLEN field of SQLDA 663, 896
- SQLLONGL field of SQLDA 898
- SQLLONGLEN field of SQLDA 664, 898
- SQLN field of SQLDA
 - description 660, 892
- SQLNAME field of SQLDA 664, 897
- SQLRULES
 - column of SYSPLAN catalog table 977
- SQLSTATE 707
 - '02000' 707, 745, 753, 806, 845
 - description 345
 - field of SQLCA 883
 - signaling 831
- SQLTNAME field of SQLDA 899
- SQLTYPE field of SQLDA
 - description 896
 - values 663, 899
- SQLVAR
 - base 663
 - extended 663
- SQLVAR field of SQLDA 663
- SQLWARN6 field of SQLCA 140
- SQLWARNING clause
 - WHENEVER statement 845
- SQLWARNn field of SQLCA 883
- SQRT function 279
- SQTY column
 - SYSINDEXPART catalog table 958
 - SYSTABLEPART catalog table 1007
- standard, SQL (ANSI/ISO)
 - description 4
 - SET CONNECTION statement 809
 - SQL-style comments 171
 - STDSQL precompiler option 170
- START column of SYSSEQUENCES catalog table 996
- START_RBA column of SYSCOPY catalog table 941
- STARTDB privilege
 - GRANT statement 716
 - REVOKE statement 779
- STARTDBAUTH column of SYSDBAUTH catalog table 948

- state
 - application process 449
 - SQL connection 35
- statement
 - descriptions 19
 - See also SQL statements
 - operational form 19
 - preparation 19
 - source form 19
- STATEMENT clause of DECLARE STATEMENT statement 649
- statement table 694
- static SQL
 - description 19, 340
 - invocation of SELECT statement 343
- STATS privilege
 - GRANT statement 716
 - REVOKE statement 779
- STATSAUTH column of SYSDBAUTH catalog table 948
- STATSTIME column
 - SYSCOLDIST catalog table 931
 - SYSCOLDISTSTATS catalog table 932
 - SYSCOLSTATS catalog table 933
 - SYSCOLUMNS catalog table 938
 - SYSINDEXES catalog table 956
 - SYSINDEXPART catalog table 959
 - SYSINDEXSTATS catalog table 960
 - SYSLOBSTATS catalog table 962
 - SYSSTOGROUP catalog table 1000
 - SYSTABLEPART catalog table 1009
 - SYSTABLES catalog table 1013
 - SYSTABLESPACE catalog table 1015
 - SYSTABSTATS catalog table 1017
- STATUS column
 - SYSPACKSTMT catalog table 971
 - SYSSTMT catalog table 998
 - SYSTABLES catalog table 1011
 - SYSTABLESPACE catalog table 1014
- STAY RESIDENT clause
 - ALTER FUNCTION statement 361
 - ALTER PROCEDURE statement 381, 387
 - CREATE FUNCTION statement 488, 505
 - CREATE PROCEDURE statement 551, 561
- STAYRESIDENT column
 - SYS PROCEDURES catalog table 983
 - SYSROUTINES catalog table 991
- STD SQL LANGUAGE field of panel DSNTIP4 170
- STDDEV function 184
- STDSQL option
 - precompiler 170
- STGROUP column of SYSDATABASE catalog table 944
- STMT column of SYSPACKSTMT catalog table 971
- STMTNO column
 - SYSPACKSTMT catalog table 971
- STMTNO column (*continued*)
 - SYSSTMT catalog table 998
- STMTNOI column
 - SYSPACKSTMT catalog table 972
 - SYSSTMT catalog table 999
- STNAME column of SYSTABAUTH catalog table 1004
- STOGROUP
 - clause of ALTER DATABASE statement 349
 - clause of ALTER INDEX statement 368, 371
 - clause of ALTER STOGROUP statement 390
 - clause of ALTER TABLESPACE statement 417
 - clause of CREATE DATABASE statement 463
 - clause of CREATE INDEX statement 530, 532
 - clause of CREATE STOGROUP statement 565
 - clause of CREATE TABLESPACE statement 600, 603
 - clause of DROP statement 680
- STOGROUP privilege
 - GRANT statement 736
 - REVOKE statement 799
- STOPALL privilege
 - GRANT statement 731
 - REVOKE statement 794
- STOPALLAUTH column of SYSUSERAUTH catalog table 1020
- STOPAUTH column of SYSDBAUTH catalog table 948
- STOPDB privilege
 - GRANT statement 716
 - REVOKE statement 779
- storage group, DB2
 - altering 390
 - creating 565
 - description 26
 - dropping 680
- storage structure 26
- stored procedure
 - altering
 - ALTER PROCEDURE statement 376, 384
 - CALL statement 428
 - creating
 - CREATE PROCEDURE statement 541, 555
 - CURRENT PACKAGESET special register 818
 - dropping 679
 - invoking 428
 - name, unqualified 56
 - naming convention 53
 - privileges 720, 783
 - statements allowed 877
 - unqualified name 56
- STORES clause of CREATE AUXILIARY TABLE statement 460
- STORNAME column
 - SYSINDEXPART catalog table 958
 - SYSTABLEPART catalog table 1007

STORTYPE column
 SYSINDEXPART catalog table 958
 SYSTABLEPART catalog table 1007
 STOSPACE privilege
 GRANT statement 731
 REVOKE statement 794
 STOSPACEAUTH column of SYSUSERAUTH catalog table 1020
 string
 binary 71
 character 67
 comparison 94
 constant 102
 conversion 38
 datetime values 76
 delimiter
 COBOL 168
 controlling representation 168
 SQL 168
 description 38
 fixed-length
 description 69, 70
 graphic 70
 long
 column limitations 313
 long column
 description 69, 70, 72
 use restrictions 72
 short 69, 70
 varying-length
 description 69, 70
 string delimiter precompiler option 168
 STRIP function 272, 280
 strong typing 79
 STYPE column of SYSCOPY catalog table 942
 SUBBYTE column of SYSSTRINGS catalog table 1001
 subquery
 description 116
 HAVING clause 322
 WHERE clause 320
 subselect
 CREATE VIEW statement 311, 629
 description 311
 example 322
 INSERT statement 311, 745
 substitution byte 39
 substitution character 91
 SUBSTR function 282
 SUBTYPE column
 SYSDATATYPES catalog table 946
 SYSPARMS catalog table 974
 SUM function 185
 synonym
 defining 568
 description 58
 synonym (*continued*)
 dropping 680
 naming convention 54
 qualifying a column name 114
 SYNONYM clause
 CREATE SYNONYM statement 568
 DROP statement 680
 syntax diagrams, how to read 4
 SYSADM authority
 GRANT statement 731
 REVOKE statement 794
 SYSADMAUTH column of SYSUSERAUTH catalog table 1020
 SYSCTRL authority
 GRANT statement 731
 REVOKE statement 794
 SYSCTRLAUTH column of SYSUSERAUTH catalog table 1021
 SYSENTRIES column
 SYSPACKAGE catalog table 963
 SYSPLAN catalog table 977
 SYSIBM... catalog tables
 See catalog tables
 SYSMODENAME column of LUNAMES catalog table 924
 SYSOPR authority
 GRANT statement 731
 REVOKE statement 794
 SYSOPRAUTH column of SYSUSERAUTH catalog table 1020
 system
 limits 869
 SYSTEM column
 SYSPKSYSTEM catalog table 975
 SYSPLSYSTEM catalog table 981
 SYSTEM PATH clause
 SET CURRENT PATH statement 819

T

table
 altering
 ALTER TABLE statement 393
 auxiliary table 21
 base table 21
 column of SYSPARMS catalog table 974
 controlling changes 25
 creating
 CREATE AUXILIARY TABLE statement 459
 CREATE GLOBAL TEMPORARY TABLE statement 520
 CREATE TABLE statement 570
 DECLARE GLOBAL TEMPORARY TABLE statement 639
 description 21
 designator 115

- table (*continued*)
 - dropping
 - DROP statement 680
 - empty table 21
 - joining 319
 - obtaining information with DESCRIBE 659
 - privileges 733, 796
 - renaming
 - RENAME statement 769
 - restricting column values 25
 - result table 21, 755
 - sample table 21
 - temporary copy 755
 - temporary table 21
- TABLE
 - clause of COMMENT ON statement 442
 - clause of DECLARE TABLE statement 650
 - clause of DROP statement 680
 - clause of LABEL ON statement 749
- table check constraint 403
 - defining
 - ALTER TABLE statement 403
 - CREATE TABLE statement 586
 - deleting rows 657
 - description 25
 - inserting rows 746
 - SYSCHECKDEP catalog table 928
 - updating rows 839
- table function 125
- TABLE LIKE clause
 - CREATE FUNCTION statement 479, 497, 513
 - CREATE PROCEDURE statement 546, 559
- table name
 - naming convention 54
 - qualifying a column name 114
 - unqualified 56
- table space
 - altering
 - ALTER TABLESPACE statement 412
 - catalog table 911
 - creating
 - CREATE TABLESPACE statement 597
 - implicitly 588
 - description 26
 - dropping 680
 - naming convention 55
- TABLE_COLNO column of SYSPARMS catalog table 974
- TABLE_LOCATION function 1039
- TABLE_NAME function 1041
- TABLE_SCHEMA function 1043
- TABLESPACE clause
 - ALTER TABLESPACE statement 412
 - DROP statement 680
- TABLESPACE privilege
 - GRANT statement 736
- TABLESPACE privilege (*continued*)
 - REVOKE statement 799
- TABLESTATUS column of SYSTABLES catalog table 1013
- TAN function 284
- TANH function 285
- TBCREATOR column
 - SYSCOLUMNS catalog table 934
 - SYSFIELDS catalog table 953
 - SYSINDEXES catalog table 955
 - SYSSYNONYMS catalog table 1003
 - SYSTABLES catalog table 1012
- TBNAME column
 - SYSAUXRELS catalog table 927
 - SYSCHECKDEP catalog table 928
 - SYSCHECKS catalog table 929
 - SYSCOLDIST catalog table 931
 - SYSCOLDISTSTATS catalog table 932
 - SYSCOLSTATS catalog table 933
 - SYSCOLUMNS catalog table 934
 - SYSFIELDS catalog table 953
 - SYSFOREIGNKEYS catalog table 954
 - SYSINDEXES catalog table 955
 - SYSRELS catalog table 985
 - SYSSYNONYMS catalog table 1003
 - SYSTABLES catalog table 1012
 - SYSTRIGGERS catalog table 1018
- TBOWNER column
 - SYSAUXRELS catalog table 927
 - SYSCHECKDEP catalog table 928
 - SYSCHECKS catalog table 929
 - SYSCOLDIST catalog table 931
 - SYSCOLDISTSTATS catalog table 932
 - SYSCOLSTATS catalog table 933
 - SYSTRIGGERS catalog table 1018
- TCREATOR column of SYSTABAUTH catalog table 1004
- TEMP database
 - creating 463
- temporary
 - copy of table 755
- temporary table
 - creating 520, 639
 - description 21
- TEXT column
 - SYSSTMT catalog table 998
 - SYSTRIGGERS catalog table 1018
 - SYSVIEWS catalog table 1023
- three-part name
 - description 33
- time
 - arithmetic 141
 - data type 75
 - duration 138
 - function 286
 - strings 77, 78

TIME
 data type
 CREATE TABLE statement 577
 description 75
TIME FORMAT field of panel DSNTIP4 170
TIMEGRANTED column
 SYSCOLAUTH catalog table 930
 SYSDBAUTH catalog table 947
 SYSPLANAUTH catalog table 979
 SYSRESAUTH catalog table 986
 SYSTABAUTH catalog table 1004
 SYSUSERAUTH catalog table 1019
timestamp
 arithmetic 142
 data type 76
 duration 138
 strings 78
TIMESTAMP
 column of SYSCHECKS catalog table 929
 column of SYSCOPY catalog table 942
 column of SYSDATABASE catalog table 944
 column of SYSDBRM catalog table 950
 column of SYSPACKAGE catalog table 963
 column of SYSPACKAUTH catalog table 968
 column of SYSPACKLIST catalog table 970
 column of SYSRELS catalog table 985
 data type
 CREATE TABLE statement 577
 description 76
 function 287
TIMESTAMP_FORMAT
 function 289
TNAME column of SYSCOLAUTH catalog table 930
TO
 clause of CONNECT (Type 1) statement 449
 clause of CONNECT (Type 2) statement 454
 clause of GRANT statement 712
TO SAVEPOINT clause
 ROLLBACK statement 802
TO_CHAR
 function 299
TO_DATE
 function 289
token in SQL 47
TPN column of LOCATIONS catalog table 921
TRACE privilege
 GRANT statement 732
 REVOKE statement 795
TRACEAUTH column of SYSUSERAUTH catalog table 1020
TRACKMOD
 clause of ALTER TABLESPACE statement 421
 clause of CREATE TABLESPACE statement 606
 column of SYSTABLEPART catalog table 1009
TRANSLATE function 290
TRANSPROC column of SYSSTRINGS catalog table 1001
TRANSTAB column of SYSSTRINGS catalog table 1001
TRANSTYPE column of SYSSTRINGS catalog table 1001
TRIGEVENT column of SYSTRIGGERS catalog table 1018
trigger
 creating 615
 description 25
 dropping 681
 naming convention 55
TRIGGER clause
 COMMENT ON statement 442
 DROP statement 681
TRIGGER privilege
 GRANT statement 734
 REVOKE statement 797
TRIGGERAUTH column of SYSTABAUTH catalog table 1006
TRIGTIME column of SYSTRIGGERS catalog table 1018
TRUNCATE function 293
truncation
 numbers 86
truth table 163
truth valued logic 163
TSNAME column
 SYSCOPY catalog table 941
 SYSTABLEPART catalog table 1007
 SYSTABLES catalog table 1010
 SYSTABSTATS catalog table 1017
TTNAME column of SYSTABAUTH catalog table 1004
TYPE 2
 clause of CREATE INDEX statement 527
TYPE column
 SYSCOLDIST catalog table 931
 SYSCOLDISTSTATS catalog table 932
 SYSDATABASE catalog table 944
 SYSPACKAGE catalog table 967
 SYSTABLES catalog table 1010
 SYSTABLESPACE catalog table 1016
 USERNAMES catalog table 1025
typed parameter marker 759
TYPENAME column
 SYSCOLUMNS catalog table 939
 SYSPARMS catalog table 973
TYPESCHEMA column
 SYSCOLUMNS catalog table 939
 SYSPARMS catalog table 973

U

UCASE function 294

- unary operation 133
- unconnected state 36
- UNION clause
 - duplicate rows 327
 - fullselect 327
 - result data type 99
- UNIQUE clause
 - CREATE INDEX statement 527
 - CREATE TABLE statement 578, 583
 - SAVEPOINT statement 804
- unique index
 - description 22
- unique key 22
- UNIQUERULE column of SYSINDEXES catalog
 - table 955
- unit of recovery
 - COMMIT statement 444
 - description 28
 - ROLLBACK statement 801
- unit of work
 - closes cursors 755
 - description 29
 - dynamic caching 763
 - ending 29, 444, 801
 - initiating 29
 - persistence of prepared statements 763
 - referring to prepared statements 757
- universal time, coordinated (UTC) 105
- unqualified object names 56
- untyped parameter marker 759
- UPDATE
 - clause of TRIGGER statement 617
 - statement
 - description 833
- UPDATE privilege
 - GRANT statement 734
 - REVOKE statement 797
- update rule 23, 838
- UPDATEAUTH column of SYSTABAUTH catalog
 - table 1005
- UPDATCOLS column of SYSTABAUTH catalog
 - table 1004
- UPDATES column of SYSCOLUMNS catalog
 - table 935
- updating
 - rows in a table 833
- UPPER function 294
- USA 76
- USAGE privilege
 - GRANT statement 718
 - REVOKE statement 781
- USEAUTH column of SYSRESAUTH catalog
 - table 986
- USER clause of SET CURRENT PATH statement 819
- USER special register 113
- user-defined function
 - altering
 - ALTER FUNCTION statement 351
 - creating
 - CREATE FUNCTION statement 472, 473, 492, 508
 - dropping 677
 - privileges 720, 783
 - statements allowed 877
- user-defined function (UDF)
 - description 125
 - external
 - description 125
 - invocation 127
 - name, unqualified 56
 - naming convention 52
 - resolution 127
 - sample 1029
 - ALTDATA 1030
 - ALTTIME 1033
 - CURRENCY 1035
 - DAYNAME 1037
 - MONTHNAME 1038
 - TABLE_LOCATION 1039
 - TABLE_NAME 1041
 - TABLE_SCHEMA 1043
 - WEATHER 1045
 - sourced
 - description 125
 - table
 - description 125
 - unqualified name 56
- USERNAMES column
 - IPNAMES catalog table 920
 - LUNAMES catalog table 925
- USING clause
 - ALTER INDEX statement 367, 370
 - ALTER TABLESPACE statement 416
 - CREATE INDEX statement 529, 531
 - CREATE TABLESPACE statement 600, 603
 - DESCRIBE statement 660
 - EXECUTE statement 689
 - OPEN statement 753
 - PREPARE statement 758
- USING DESCRIPTOR clause
 - EXECUTE statement 689
 - FETCH statement 708
 - OPEN statement 754
- USING TYPE DEFAULTS clause
 - DECLARE GLOBAL TEMPORARY TABLE
 - statement 644
- UTC (universal time, coordinated) 105

V

VALID column
 SYSPACKAGE catalog table 963
 SYSPLAN catalog table 976

VALIDATE
 column of SYSPACKAGE catalog table 964
 column of SYSPLAN catalog table 976

validation procedure 402

validation routine
 VALIDPROC clause 402, 589

VALIDPROC clause
 ALTER TABLE statement 402
 CREATE TABLE statement 589

VALPROC column of SYSTABLES catalog table 1010

value
 composite 22
 SQL 66

VALUE function 99, 203, 232

VALUES clause
 ALTER INDEX statement 373
 CREATE INDEX statement 535
 INSERT statement 744
 VALUES INTO statement 843
 VALUES statement 842

VALUES INTO statement 843

VALUES statement 842

VAR function 186

VARCHAR data type
 CREATE TABLE statement 576
 description 69

VARCHAR function 295

VARCHAR_FORMAT
 function 299

VARGRAPHIC
 data type
 CREATE TABLE statement 577
 description 70

VARGRAPHIC function 301

variable
 host
 referencing 120
 SQL syntax 120

VARIANCE function 186

VARIANT clause
 ALTER PROCEDURE statement 377, 385
 CREATE FUNCTION statement 474, 493
 CREATE PROCEDURE statement 542, 556

VCAT
 clause of CREATE STOGROUP statement 566
 USING clause
 ALTER INDEX statement 368
 ALTER TABLESPACE statement 416
 CREATE INDEX statement 370, 530, 532
 CREATE TABLESPACE statement 600, 603

VCATNAME column
 SYSINDEXPART catalog table 958
 SYSSTOGROUP catalog table 1000
 SYSTABLEPART catalog table 1007

VERSION
 clause of DROP statement 679
 column of SYSDBRM catalog table 951
 column of SYSPACKAGE catalog table 965
 column of SYSPACKSTMT catalog table 971

version identificaton, current server 450, 455

version-id naming convention 55

view
 creating
 CREATE VIEW statement 627
 description 27
 dropping
 description 681
 name, unqualified 56
 naming convention 55
 privileges 733, 796
 read-only 631
 unqualified name 56
 using 631
 obtaining information with DESCRIBE 659

VIEW clause
 CREATE VIEW statement 627
 DROP statement 681

VOLID column of SYSVOLUMES catalog table 1024

VOLUMES clause of CREATE STOGROUP
 statement 565

VSAM (virtual storage access method)
 catalog 532

W

WEATHER function 1045

WEEK function 304

WHEN clause of TRIGGER statement 619

WHENEVER statement
 description 845

WHERE clause
 DELETE statement 655
 description 320
 search condition 320
 UPDATE statement 837

WHILE statement
 example 868
 SQL procedure 868

WITH CHECK OPTION clause of CREATE VIEW
 statement 629

WITH clause
 DELETE statement 656
 INSERT statement 745
 SELECT INTO statement 807
 select-statement 334

WITH COMPARISONS clause of CREATE DISTINCT
TYPE statement 468
WITH GRANT OPTION clause of GRANT
statement 712
WITH HOLD clause of DECLARE CURSOR
statement 635
WITH PROCEDURE clause of ASSOCIATE
LOCATORS statement 424
WITH RETURN clause of DECLARE CURSOR
statement 636
WITH RR|RS|CS|UR clause 807
WLM ENVIRONMENT clause
ALTER FUNCTION statement 360
ALTER PROCEDURE statement 380, 386
CREATE FUNCTION statement 487, 505
CREATE PROCEDURE statement 550, 560
WLM_ENV column clause of SYSPROCEDURES
catalog table 983
WLM_ENV_FOR_NESTED column of SYSROUTINES
catalog table 992
WLM_ENVIRONMENT column of SYSROUTINES
catalog table 992
work file database
creating 463
WORKAREA column of SYSFIELDS catalog table 953

Y

YEAR function 305

How to send your comments

DB2 Universal Database for OS/390
SQL Reference
Version 6
Publication No. SC26-9014-02

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 for OS/390 documentation. You can use any of the following methods to provide comments.

- Send your comments by e-mail to db2pubs@vnet.ibm.com and include the name of the product, the version number of the product 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, page number, or a help topic title).

- Send your comments from the Web. Visit the DB2 for OS/390 Web site at:

<http://www.ibm.com/software/db2os390>

The Web site has a feedback page that you can use to send comments.

- Complete the readers' comment form at the back of the book and return it by mail, by fax (800-426-7773 for the United States and Canada), or by giving it to an IBM representative.

Readers' Comments

**DB2 Universal Database for OS/390
SQL Reference
Version 6**

Publication No. SC26-9014-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	<input type="checkbox"/>				
Complete	<input type="checkbox"/>				
Easy to find	<input type="checkbox"/>				
Easy to understand	<input type="checkbox"/>				
Well organized	<input type="checkbox"/>				
Applicable to your tasks	<input type="checkbox"/>				
Grammatically correct and consistent	<input type="checkbox"/>				
Graphically well designed	<input type="checkbox"/>				
Overall satisfaction	<input type="checkbox"/>				

Please tell us how we can improve this book:

May we contact you to discuss your comments? Yes No

Name

Address

Company or Organization

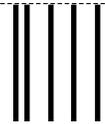
Phone No.



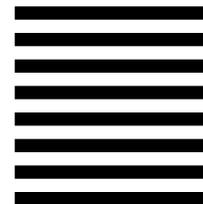
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department HHX/H3
PO Box 49023
San Jose, CA 95161-9023



Fold and Tape

Please do not staple

Fold and Tape



Program Number: 5645-DB2



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-9014-02

