

**IBM DB2 Universal Database  
SQL Reference  
Version 5.2**

Document Number S10J-8165-01





IBM DB2 Universal Database

# SQL Reference

*Version 5.2*





IBM DB2 Universal Database

# SQL Reference

*Version 5.2*

Before using this information and the product it supports, be sure to read the general information under Appendix Q, "Notices" on page 991.

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

Order publications through your IBM representative or the IBM branch office serving your locality or by calling 1-800-879-2755 in U.S. or 1-800-IBM-4YOU in Canada.

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

© **Copyright International Business Machines Corporation 1993, 1998. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Chapter 1. Introduction</b> . . . . .	1
Who Should Use This Book . . . . .	1
How To Use This Book . . . . .	1
How This Book is Structured . . . . .	1
How to Read the Syntax Diagrams . . . . .	2
Conventions Used in This Manual . . . . .	5
Error Conditions . . . . .	5
Highlighting Conventions . . . . .	5
Related Documentation for This Book . . . . .	5
<b>Chapter 2. Concepts</b> . . . . .	7
Relational Database . . . . .	7
Structured Query Language (SQL) . . . . .	7
Embedded SQL . . . . .	7
Static SQL . . . . .	7
Dynamic SQL . . . . .	8
DB2 Call Level Interface (CLI) . . . . .	8
Interactive SQL . . . . .	9
Schemas . . . . .	9
Controlling Use of Schemas . . . . .	9
Tables . . . . .	10
Views . . . . .	10
Aliases . . . . .	11
Indexes . . . . .	11
Keys . . . . .	12
Unique Keys . . . . .	12
Primary Keys . . . . .	12
Foreign Keys . . . . .	12
Partitioning Keys . . . . .	12
Constraints . . . . .	12
Unique Constraints . . . . .	13
Referential Constraints . . . . .	14
Table Check Constraints . . . . .	16
Triggers . . . . .	17
Event Monitors . . . . .	18
Queries . . . . .	18
Table Expressions . . . . .	18
Common Table Expressions . . . . .	19
Packages . . . . .	19
Catalog Views . . . . .	19
Application Processes, Concurrency, and Recovery . . . . .	19
Isolation Level . . . . .	22
Repeatable Read (RR) . . . . .	22
Read Stability (RS) . . . . .	23
Cursor Stability (CS) . . . . .	23

Uncommitted Read (UR)	24
Comparison of Isolation Levels	24
Distributed Relational Database	24
Application Servers	25
CONNECT (Type 1) and CONNECT (Type 2)	25
Remote Unit of Work	26
Application-Directed Distributed Unit of Work	30
Data Representation Considerations	34
Character Conversion	35
Character Sets and Code Pages	36
Code Page Attributes	37
Authorization and Privileges	38
Storage Structures	39
Data Partitioning Across Multiple Partitions	41
Partitioning Maps	41
Table Collocation	42
<b>Chapter 3. Language Elements</b>	<b>45</b>
Characters	45
MBCS Considerations	46
Tokens	46
MBCS Considerations	47
Identifiers	47
SQL Identifiers	47
Host Identifiers	48
Naming Conventions and Implicit Object Name Qualifications	48
Aliases	51
Authorization IDs and authorization-names	51
Data Types	54
Nulls	54
Large Objects (LOBs)	55
Character Strings	56
Graphic Strings	58
Binary String	59
Numbers	59
Datetime Values	60
DATALINK Values	63
User Defined Types	64
Promotion of Data Types	66
Casting Between Data Types	67
Assignments and Comparisons	70
Numeric Assignments	72
String Assignments	72
Datetime Assignments	75
DATALINK Assignments	75
User-defined Type Assignments	77
Reference Type Assignments	77
Numeric Comparisons	78



String Comparisons	78
Datetime Comparisons	81
User-defined Type Comparisons	81
Reference Type Comparisons	82
Rules for Result Data Types	82
Character Strings	83
Graphic Strings	83
Binary Large Object (BLOB)	83
Numeric	83
DATE	84
TIME	84
TIMESTAMP	84
DATALINK	84
User-defined Types	85
Nullable Attribute of Result	85
Rules for String Conversions	85
Partition Compatibility	87
Constants	88
Integer Constants	89
Floating-Point Constants	89
Decimal Constants	89
Character String Constants	89
Hexadecimal Constants	90
Graphic String Constants	90
Special Registers	91
CURRENT DATE	91
CURRENT DEGREE	91
CURRENT EXPLAIN MODE	92
CURRENT EXPLAIN SNAPSHOT	93
CURRENT NODE	94
CURRENT PATH	94
CURRENT QUERY OPTIMIZATION	95
CURRENT REFRESH AGE	95
CURRENT SCHEMA	96
CURRENT SERVER	96
CURRENT TIME	96
CURRENT TIMESTAMP	97
CURRENT TIMEZONE	97
USER	98
Column Names	98
Qualified Column Names	98
Correlation Names	99
Column Name Qualifiers to Avoid Ambiguity	101
Column Name Qualifiers in Correlated References	102
References to Host Variables	105
Host Variables in Dynamic SQL	105
References to BLOB, CLOB, and DBCLOB Host Variables	107
References to Locator Variables	108

References to BLOB, CLOB, and DBCLOB File Reference Variables . . . . .	108
Functions . . . . .	110
Function Resolution . . . . .	112
Function Invocation . . . . .	115
Expressions . . . . .	117
Without Operators . . . . .	117
With the Concatenation Operator . . . . .	117
With Arithmetic Operators . . . . .	120
Two Integer Operands . . . . .	121
Integer and Decimal Operands . . . . .	121
Two Decimal Operands . . . . .	121
Decimal Arithmetic in SQL . . . . .	122
Floating-Point Operands . . . . .	122
User-defined Types as Operands . . . . .	122
Datetime Operations and Durations . . . . .	123
Datetime Arithmetic in SQL . . . . .	124
Precedence of Operations . . . . .	128
CASE Expressions . . . . .	129
CAST Specifications . . . . .	131
Dereference Operations . . . . .	134
Predicates . . . . .	135
Basic Predicate . . . . .	136
Quantified Predicate . . . . .	137
BETWEEN Predicate . . . . .	140
EXISTS Predicate . . . . .	142
IN Predicate . . . . .	143
LIKE Predicate . . . . .	146
NULL Predicate . . . . .	151
TYPE Predicate . . . . .	152
Search Conditions . . . . .	153
<b>Chapter 4. Functions . . . . .</b>	<b>155</b>
Column Functions . . . . .	170
AVG . . . . .	171
COUNT . . . . .	173
COUNT_BIG . . . . .	174
GROUPING . . . . .	176
MAX . . . . .	178
MIN . . . . .	180
STDDEV . . . . .	182
SUM . . . . .	183
VARIANCE . . . . .	184
Scalar Functions . . . . .	185
ABS or ABSVAL . . . . .	186
ACOS . . . . .	187
ASCII . . . . .	188
ASIN . . . . .	189
ATAN . . . . .	190

ATAN2	191
BIGINT	192
BLOB	193
CEIL or CEILING	194
CHAR	195
CHR	200
CLOB	201
COALESCE	202
CONCAT	203
COS	204
COT	205
DATE	206
DAY	207
DAYNAME	208
DAYOFWEEK	209
DAYOFYEAR	210
DAYS	211
DBCLOB	212
DECIMAL	213
DEGREES	216
DEREF	217
DIFFERENCE	218
DIGITS	219
DLCOMMENT	220
DLINKTYPE	221
DLURLCOMPLETE	222
DLURLPATH	223
DLURLPATHONLY	224
DLURLSCHEME	225
DLURLSERVER	226
DLVALUE	227
DOUBLE	229
EVENT_MON_STATE	231
EXP	232
FLOAT	233
FLOOR	234
GENERATE_UNIQUE	235
GRAPHIC	237
HEX	238
HOUR	240
INSERT	241
INTEGER	242
JULIAN_DAY	243
LCASE	244
LEFT	245
LENGTH	246
LN	247
LOCATE	248

LOG	249
LOG10	250
LONG_VARCHAR	251
LONG_VARGRAPHIC	252
LTRIM	253
MICROSECOND	254
MIDNIGHT_SECONDS	255
MINUTE	256
MOD	257
MONTH	258
MONTHNAME	259
NODENUMBER	260
NULLIF	262
PARTITION	263
POSSTR	265
POWER	267
QUARTER	268
RADIANS	269
RAISE_ERROR	270
RAND	272
REAL	273
REPEAT	274
REPLACE	275
RIGHT	276
ROUND	277
RTRIM	278
SECOND	279
SIGN	280
SIN	281
SMALLINT	282
SOUNDEX	283
SPACE	284
SQRT	285
SUBSTR	286
TABLE_NAME	289
TABLE_SCHEMA	291
TAN	293
TIME	294
TIMESTAMP	295
TIMESTAMP_ISO	297
TIMESTAMPDIFF	298
TRANSLATE	299
TRUNC or TRUNCATE	301
TYPE_ID	302
TYPE_NAME	303
TYPE_SCHEMA	304
UCASE	305
VALUE	306

VARCHAR	307
VARGRAPHIC	308
WEEK	310
YEAR	311
User-Defined Functions	312
<b>Chapter 5. Queries</b>	<b>315</b>
subselect	316
select-clause	317
from-clause	321
table-reference	322
joined-table	326
where-clause	328
group-by-clause	329
having-clause	336
Examples of subselects	337
Examples of Joins	339
Examples of Grouping Sets, Cube, and Rollup	342
fullselect	350
Examples of a fullselect	352
select-statement	355
common-table-expression	356
order-by-clause	358
update-clause	361
read-only-clause	362
fetch-first-clause	363
optimize-for-clause	364
Examples of a select-statement	365
<b>Chapter 6. Statements</b>	<b>367</b>
How SQL Statements Are Invoked	369
Embedding a Statement in an Application Program	370
Dynamic Preparation and Execution	371
Static Invocation of a select-statement	371
Dynamic Invocation of a select-statement	372
Interactive Invocation	372
SQL Return Codes	373
SQLCODE	373
SQLSTATE	373
SQL Comments	374
ALTER BUFFERPOOL	375
ALTER NODEGROUP	377
ALTER TABLE	380
ALTER TABLESPACE	399
ALTER TYPE (Structured)	403
ALTER VIEW	405
BEGIN DECLARE SECTION	407
CALL	409

CLOSE	416
COMMENT ON	418
COMMIT	426
Compound SQL	428
CONNECT (Type 1)	432
CONNECT (Type 2)	439
CREATE ALIAS	446
CREATE BUFFERPOOL	449
CREATE DISTINCT TYPE	452
CREATE EVENT MONITOR	458
CREATE FUNCTION	467
CREATE FUNCTION (External Scalar)	468
CREATE FUNCTION (External Table)	484
CREATE FUNCTION (Sourced)	497
CREATE INDEX	504
CREATE NODEGROUP	508
CREATE PROCEDURE	511
CREATE SCHEMA	519
CREATE TABLE	522
CREATE TABLESPACE	559
CREATE TRIGGER	568
CREATE TYPE (Structured)	578
CREATE VIEW	582
DECLARE CURSOR	595
DELETE	599
DESCRIBE	604
DISCONNECT	608
DROP	611
END DECLARE SECTION	624
EXECUTE	626
EXECUTE IMMEDIATE	631
EXPLAIN	633
FETCH	637
FREE LOCATOR	640
GRANT (Database Authorities)	641
GRANT (Index Privileges)	644
GRANT (Package Privileges)	646
GRANT (Schema Privileges)	649
GRANT (Table or View Privileges)	652
INCLUDE	659
INSERT	661
LOCK TABLE	666
OPEN	668
PREPARE	673
REFRESH TABLE	682
RELEASE	683
RENAME TABLE	685
REVOKE (Database Authorities)	687

REVOKE (Index Privileges)	690
REVOKE (Package Privileges)	692
REVOKE (Schema Privileges)	695
REVOKE (Table or View Privileges)	697
ROLLBACK	702
SELECT	704
SELECT INTO	705
SET CONNECTION	707
SET CONSTRAINTS	709
SET CURRENT DEGREE	716
SET CURRENT EXPLAIN MODE	718
SET CURRENT EXPLAIN SNAPSHOT	720
SET CURRENT PACKAGESET	722
SET CURRENT QUERY OPTIMIZATION	724
SET CURRENT REFRESH AGE	727
SET EVENT MONITOR STATE	729
SET PATH	731
SET SCHEMA	733
SET transition-variable	735
SIGNAL SQLSTATE	738
UPDATE	740
VALUES	747
VALUES INTO	748
WHENEVER	750
<b>Appendix A. SQL Limits</b>	<b>753</b>
<b>Appendix B. SQL Communication Area (SQLCA)</b>	<b>759</b>
Viewing the SQLCA Interactively	759
SQLCA Field Descriptions	759
Order of Error Reporting	762
DB2 Extended Enterprise Edition Usage of the SQLCA	762
<b>Appendix C. Appendix C. SQL Descriptor Area (SQLDA)</b>	<b>763</b>
Field Descriptions	763
Fields in the SQLDA Header	764
Fields in an Occurrence of a Base SQLVAR	765
Fields in an Occurrence of a Secondary SQLVAR	766
Effect of DESCRIBE on the SQLDA	767
SQLTYPE and SQLEEN	768
Packed Decimal Numbers	770
UNRECOGNIZED AND UNSUPPORTED SQLTYPES	771
SQLEEN Field for Decimal	771
<b>Appendix D. Catalog Views</b>	<b>773</b>
Updatable Catalog Views	774
“Roadmap” to Catalog Views	774
“Roadmap” to Updatable Catalog Views	775

SYSCAT.BUFFERPOOLS	776
SYSCAT.BUFFERPOOLNODES	777
SYSCAT.CHECKS	778
SYSCAT.COLAUTH	779
SYSCAT.COLCHECKS	780
SYSCAT.COLDIST	781
SYSCAT.COLUMNS	782
SYSCAT.CONSTDEP	784
SYSCAT.DATATYPES	785
SYSCAT.DBAUTH	786
SYSCAT.EVENTMONITORS	787
SYSCAT.EVENTS	788
SYSCAT.FUNCPARMS	789
SYSCAT.FUNCTIONS	790
SYSCAT.INDEXAUTH	793
SYSCAT.INDEXES	794
SYSCAT.KEYCOLUSE	797
SYSCAT.NODEGROUPDEF	798
SYSCAT.NODEGROUPS	799
SYSCAT.PACKAGEAUTH	800
SYSCAT.PACKAGEDEP	801
SYSCAT.PACKAGES	802
SYSCAT.PARTITIONMAPS	805
SYSCAT.PROCEDURES	806
SYSCAT.PROCPARMS	807
SYSCAT.REFERENCES	808
SYSCAT.SCHEMAAUTH	809
SYSCAT.SCHEMATA	810
SYSCAT.STATEMENTS	811
SYSCAT.TABAUTH	812
SYSCAT.TABCONST	814
SYSCAT.TABLES	815
SYSCAT.TABLESPACES	818
SYSCAT.TRIGDEP	819
SYSCAT.TRIGGERS	820
SYSCAT.VIEWDEP	821
SYSCAT.VIEWS	822
SYSSTAT.COLDIST	823
SYSSTAT.COLUMNS	824
SYSSTAT.FUNCTIONS	825
SYSSTAT.INDEXES	827
SYSSTAT.TABLES	830
<b>Appendix E. Catalog Views For Use With Structured Types</b>	<b>831</b>
Updatable Catalog Views For Use With Structured Types	832
“Roadmap” to Catalog Views for Structured Types	832
“Roadmap” to Updatable Catalog Views For Structured Types	833
OBJCAT.ATTRIBUTES	835



OBJCAT.CHECKS	836
OBJCAT.COLCHECKS	837
OBJCAT.COLUMNS	838
OBJCAT.CONSTDEP	841
OBJCAT.DATATYPES	842
OBJCAT.FUNCPARMS	843
OBJCAT.FUNCTIONS	844
OBJCAT.HIERARCHIES	847
OBJCAT.INDEXES	848
OBJCAT.KEYCOLUSE	851
OBJCAT.PACKAGEDEP	852
OBJCAT.REFERENCES	853
OBJCAT.TABCONST	854
OBJCAT.TABLES	855
OBJCAT.TRIGDEP	858
OBJCAT.TRIGGERS	859
OBJCAT.VIEWDEP	860
OBJSTAT.TABLES	861

<b>Appendix F. Sample Tables</b>	863
The Sample Database	863
To Install the Sample Database	863
To Erase the Sample Database	864
CL_SCHED Table	864
DEPARTMENT Table	864
EMPLOYEE Table	865
EMP_ACT Table	867
EMP_PHOTO Table	869
EMP_RESUME Table	869
IN_TRAY Table	870
ORG Table	870
PROJECT Table	870
SALES Table	871
STAFF Table	872
STAFFG Table	874
Sample Files with BLOB and CLOB Data Type	875
Quintana Photo	875
Quintana Resume	875
Nicholls Photo	876
Nicholls Resume	876
Adamson Photo	877
Adamson Resume	878
Walker Photo	879
Walker Resume	879

<b>Appendix G. Reserved Schema Names and Reserved Words</b>	881
Reserved Schemas	881
Reserved Words	881

IBM SQL Reserved Words . . . . .	881
ISO/ANS SQL92 Reserved Words . . . . .	882
<b>Appendix H. Comparison of Isolation Levels . . . . .</b>	<b>885</b>
<b>Appendix I. Interaction of Triggers and Constraints . . . . .</b>	<b>887</b>
<b>Appendix J. Incompatibilities Between Releases . . . . .</b>	<b>891</b>
System Catalog Tables/Views . . . . .	892
System Catalog Views . . . . .	892
System Catalog Tables . . . . .	892
Unique Table Identification . . . . .	894
Application Programming . . . . .	894
NS, NW and NX Locks . . . . .	894
CREATE TABLE NOT LOGGED INITIALLY . . . . .	895
DB2 Call Level Interface (DB2 CLI) Defaults . . . . .	896
Obsolete DB2 CLI Keywords . . . . .	896
DB2 CLI SQLSTATEs . . . . .	897
DB2 CLI Mixing Embedded SQL, Without CONNECT RESET . . . . .	897
DB2 CLI Use of VARCHAR FOR BIT DATA . . . . .	897
DB2 CLI Data Conversion Values for SQLGetInfo . . . . .	898
DB2 CLI/ODBC Configuration Keyword Defaults . . . . .	898
Obsolete DB2 CLI/ODBC Configuration Keywords . . . . .	899
DB2 CLI SQLSTATEs . . . . .	899
Stored Procedure Catalog Table . . . . .	900
PREP Command - LANGLEVEL . . . . .	900
Change to SMALLINT Constants . . . . .	900
Down-level Client and Distinct Types Sourced on BIGINT . . . . .	901
Error Handling . . . . .	902
Maximum Number of Sections in a Package . . . . .	902
Bind Warnings . . . . .	903
Bind Options . . . . .	903
PREP with BINDFILE . . . . .	903
Varchar Structures in COBOL . . . . .	904
Incompatible APIs . . . . .	905
Supported Level of JDBC . . . . .	905
Calling Convention for Java Stored Procedures and UDFs . . . . .	905
Java Runtime Environment . . . . .	906
Obsolete System Monitor Requests for DB2 PE Version 1.2 . . . . .	906
SQL . . . . .	907
Updating Partitioning Key Columns . . . . .	907
Column NGNAME . . . . .	907
Node Number Temporary Space Usage . . . . .	908
Authorities for Create and Drop Nodegroups . . . . .	908
Target Map in REDISTRIBUTE NODEGROUP . . . . .	908
Node Group for Create Table . . . . .	909
Revoking CONTROL on Tables or Views . . . . .	910
High Level Qualifiers for Objects in DB2 Version 5 . . . . .	910

Inoperative VIEWS	911
Unusable VIEWS	912
SQLCODE Changes	912
WITH CHECK OPTION on CREATE VIEW	913
SQLSTATE Changes	913
FOR BIT DATA Comparisons	913
Code Page Conversion	914
Isolation Levels and Blocking All	915
ORDER BY Temporary Space Usage	915
Using Quotes in SQL Statements	916
Database Security and Tuning	916
GROUP Authorizations	916
Authentication Type	917
SYSADM Groups	917
Security Enhancements	917
Obsolete Profile Registry and Environment Variables	918
Utilities and Tools	918
Executable Name Changes	918
Backup and Restore - BUFF_SIZE Parameter	919
Backup and Restore - Changes Only Option	919
Backup and Restore - User Exits	919
Backup and Restore - Authority	920
Import - IMPORT REPLACE Option	920
LOAD TERMINATE	921
REORG - Alternate Path Option	921
Connectivity and Coexistence	921
Distributed Transaction Processing - Connect Type	921
Distributed Transaction Processing - SQLERRD Changes	922
DDCS - SQLJSETP	923
DDCS - DDCSSETP	923
DDCS - SQLJTRC.CMD	923
DDCS - SQLJBIND.CMD	924
APPC and APPN Nodes	924
Configuration Parameters	925
ADSM_PASSWORD	925
Agent Pool Size (NUM_POOLAGENTS)	925
MAXDARI and MAXCAGENTS	926
LOGFILSIZ	926
PCKCACHEFILSIZ	927
APPLHEAPSZ and APP_CTL_HEAP_SZ	928
BUFFPAGE and Multiple Buffer Pools	928
NEWLOGPATH	929
MULTIPAGE_ALLOC	929
EXTENTSIZ vs SEGPAGES	929
LOCKLIST	930
BUFFPAGE and SORTHEAP	930
Numeric Values for Database Manager Configuration Tokens	931
Numeric Values for Database Manager Configuration Tokens	931

New Generic Out-of-Range Return Codes . . . . .	932
Segments versus 4KB Pages . . . . .	933
Obsolete Database Configuration Parameters . . . . .	933
Obsolete Database Manager Configuration Parameters . . . . .	933
DB2_MMAP_READ and DB2_MMAP_WRITE . . . . .	934

<b>Appendix K. Explain Tables and Definitions . . . . .</b>	<b>935</b>
EXPLAIN_ARGUMENT Table . . . . .	935
EXPLAIN_INSTANCE Table . . . . .	939
EXPLAIN_OBJECT Table . . . . .	940
EXPLAIN_OPERATOR Table . . . . .	942
EXPLAIN_PREDICATE Table . . . . .	944
EXPLAIN_STATEMENT Table . . . . .	946
EXPLAIN_STREAM Table . . . . .	948
Table Definitions for Explain Tables . . . . .	949
EXPLAIN_ARGUMENT Table Definition . . . . .	950
EXPLAIN_INSTANCE Table Definition . . . . .	951
EXPLAIN_OBJECT Table Definition . . . . .	952
EXPLAIN_OPERATOR Table Definition . . . . .	953
EXPLAIN_PREDICATE Table Definition . . . . .	954
EXPLAIN_STATEMENT Table Definition . . . . .	955
EXPLAIN_STREAM Table Definition . . . . .	956

<b>Appendix L. Explain Register Values . . . . .</b>	<b>957</b>
------------------------------------------------------	------------

<b>Appendix M. Recursion Example: Bill of Materials . . . . .</b>	<b>961</b>
Example 1: Single Level Explosion . . . . .	961
Example 2: Summarized Explosion . . . . .	963
Example 3: Controlling Depth . . . . .	964

<b>Appendix N. Exception Tables . . . . .</b>	<b>967</b>
Rules for Creating an Exception Table . . . . .	967
Handling Rows in the Exception Tables . . . . .	969
Querying the Exception Tables . . . . .	969

<b>Appendix O. Japanese and Traditional-Chinese EUC Considerations . . . . .</b>	<b>973</b>
Language Elements . . . . .	973
Characters . . . . .	973
Tokens . . . . .	973
Identifiers . . . . .	973
Data Types . . . . .	974
Assignments and Comparisons . . . . .	974
Rules for Result Data Types . . . . .	975
Rules for String Conversions . . . . .	975
Constants . . . . .	976
Functions . . . . .	976
Expressions . . . . .	976
Predicates . . . . .	977

Functions	978
LENGTH	978
SUBSTR	978
TRANSLATE	978
VARGRAPHIC	978
Statements	979
CONNECT	979
PREPARE	979
<b>Appendix P. How the DB2 Library Is Structured</b>	<b>981</b>
SmartGuides	981
Online Help	982
DB2 Books	983
Viewing Online Books	987
Searching Online Books	988
Printing the PostScript Books	988
Ordering the Printed DB2 Books	989
Information Center	990
<b>Appendix Q. Notices</b>	<b>991</b>
Trademarks	991
Trademarks of Other Companies	992
<b>Appendix R. Contacting IBM</b>	<b>993</b>
<b>Index</b>	<b>995</b>



---

## Chapter 1. Introduction

This introductory chapter:

- Identifies this book's purpose and audience.
- Explains how to use the book and its structure.
- Explains the syntax diagram notation, the naming and highlighting conventions used throughout the manual.
- Lists related documentation.
- Presents the product family overview

---

### Who Should Use This Book

This book is intended for anyone who wants to use the Structured Query Language (SQL) to access a database. It is primarily for programmers and database administrators, but it can also be used by general users using the command line processor.

This book is a reference rather than a tutorial. It assumes that you will be writing application programs and therefore presents the full functions of the database manager.

---

### How To Use This Book

This book defines the SQL language used by DB2 Version 5.2. Use it as a reference manual for information on relational database concepts, language elements, functions, the forms of queries, and the syntax and semantics of the SQL statements. The appendixes can be used to find limitations and information on important components.

### How This Book is Structured

This book has the following sections:

- Chapter 1, "Introduction," identifies the purpose, the audience, and the use of the book.
- Chapter 2, "Concepts" on page 7, discusses the basic concepts of relational databases and SQL.
- Chapter 3, "Language Elements" on page 45, describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- Chapter 4, "Functions" on page 155, contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL column and scalar functions.
- Chapter 5, "Queries" on page 315, describes the various forms of a query.
- Chapter 6, "Statements" on page 367, contains syntax diagrams, semantic descriptions, rules, and examples of all SQL statements.
- The appendixes contain the following information:
  - Appendix A, "SQL Limits" on page 753 contains the SQL limitations
  - Appendix B, "SQL Communication Area (SQLCA)" on page 759 contains the SQLCA structure

- Appendix C, “Appendix C. SQL Descriptor Area (SQLDA)” on page 763 contains the SQLDA structure
- Appendix D, “Catalog Views” on page 773 contains the catalog views for the database
- Appendix F, “Sample Tables” on page 863 contains the sample tables used for examples
- Appendix G, “Reserved Schema Names and Reserved Words” on page 881 contains the reserved schema names and the reserved words for the IBM SQL and ISO/ANS SQL92 standards
- Appendix H, “Comparison of Isolation Levels” on page 885 contains a summary of the isolation levels.
- Appendix I, “Interaction of Triggers and Constraints” on page 887 discusses the interaction of triggers and referential constraints.
- Appendix J, “Incompatibilities Between Releases” on page 891 contains the release to release incompatibilities.
- Appendix K, “ Explain Tables and Definitions” on page 935 contains the Explain tables and how they are defined.
- Appendix L, “Explain Register Values” on page 957 describes the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values with each other and the PREP and BIND commands.
- Appendix M, “Recursion Example: Bill of Materials” on page 961 contains an example of a recursive query.
- Appendix N, “Exception Tables” on page 967 contains information on user-created tables that are used with the SET CONSTRAINTS statement.
- Appendix O, “Japanese and Traditional-Chinese EUC Considerations” on page 973 lists considerations when using EUC character sets.

---

## How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined as follows:

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

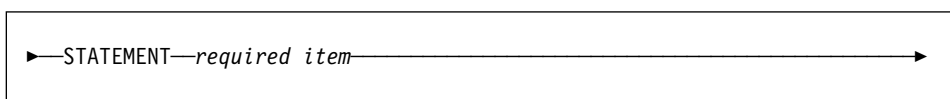
The ► symbol indicates the beginning of a statement.

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

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

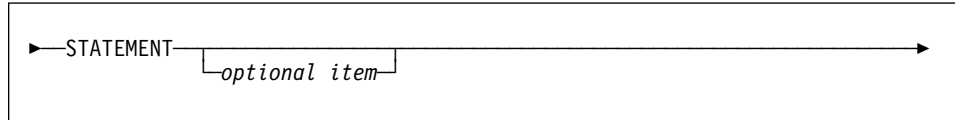
The →◄ symbol indicates the end of a statement.

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

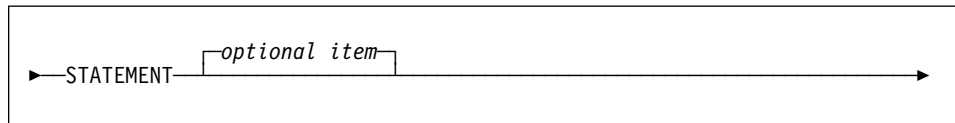




Optional items appear below the main path.

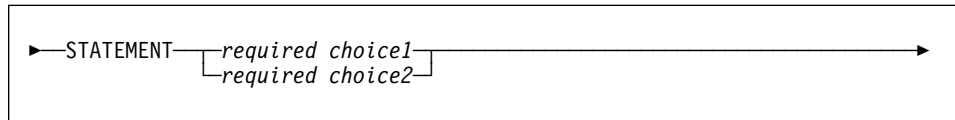


If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

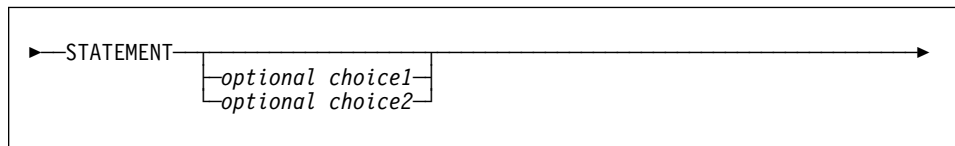


If you can choose from two or more items, they appear in a stack.

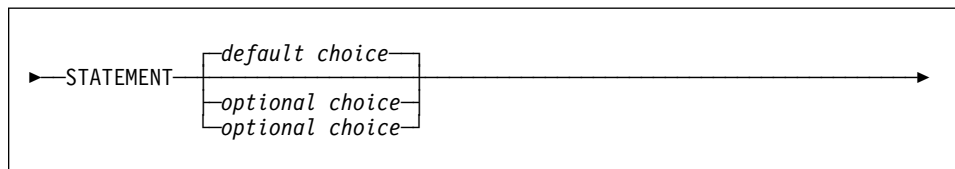
If you *must* choose one of the items, one item of the stack appears on the main path.



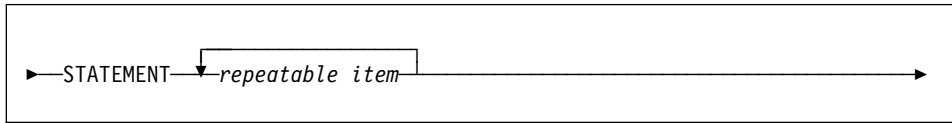
If choosing none of the items is an option, the entire stack appears below the main path.



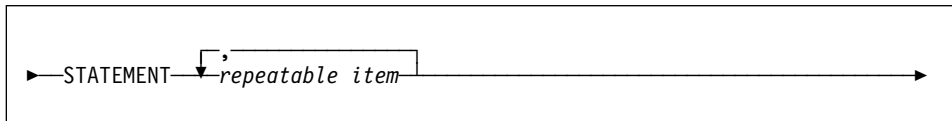
If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.



An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.



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

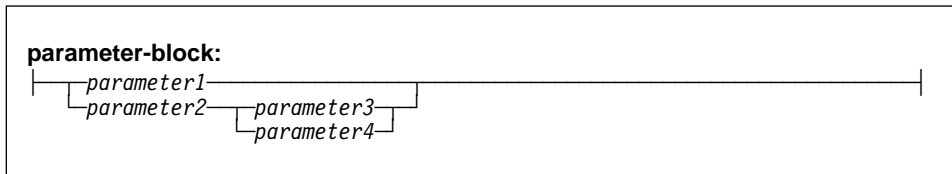
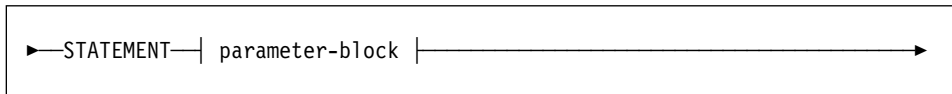


A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

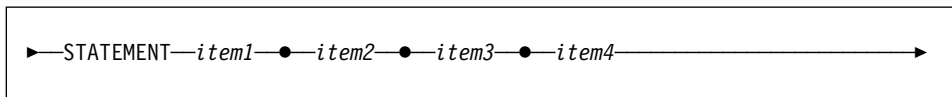
Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in lowercase (for example, column-name). They represent user-supplied names or values in the syntax.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sometimes a single variable represents a set of several parameters. For example, in the following diagram, the variable parameter-block can be replaced by any of the interpretations of the diagram that is headed **parameter-block**:



Adjacent segments occurring between "fat bullets" (●) may be specified in any sequence.



The above diagram shows that item2 and item3 may be specified in either order. Both of the following are valid:

```
STATEMENT item1 item2 item3 item4
STATEMENT item1 item3 item2 item4
```

---

## Conventions Used in This Manual

This section specifies some conventions which are used consistently throughout this manual.

### Error Conditions

An error condition is indicated within the text of the manual by listing the SQLSTATE associated with the error in brackets. For example: A duplicate signature raises an SQL error (SQLSTATE 42723).

### Highlighting Conventions

The following conventions are used in this book.

---

<b>Bold</b>	Indicates commands, keywords, and other items whose names are predefined by the system.
<i>Italics</i>	Indicates one of the following: <ul style="list-style-type: none"><li>• Names or values (variables) that must be supplied by the user</li><li>• General emphasis</li><li>• The introduction of a new term</li><li>• A reference to another source of information.</li></ul>
Monospace	Indicates one of the following: <ul style="list-style-type: none"><li>• Files and directories</li><li>• Information that you are instructed to type at a command prompt or in a window</li><li>• Examples of specific data values</li><li>• Examples of text similar to what may be displayed by the system</li><li>• Examples of system messages.</li></ul>

---

---

## Related Documentation for This Book

The following publications may prove useful in preparing applications:

*Administration Guide*

Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.

*Embedded SQL Programming Guide*

Discusses the application development process and how to code, compile, and execute application programs that use embedded SQL and APIs to access the database.

*IBM SQL Reference SC26-8416*

This manual contains all the common elements of SQL that span across IBM's library of database products. It provides limits and rules that assist in preparing portable programs using IBM databases. It provides a list of SQL extensions

and incompatibilities among the following standards and products: SQL92E, XPGG4-SQL, IBM-SQL and the IBM relational database products.

*American National Standard X3.135-1992, Database Language SQL*

Contains the ANSI standard definition of SQL.

*ISO/IEC 9075:1992, Database Language SQL*

Contains the ISO standard definition of SQL.

---

## Chapter 2. Concepts

The chapter provides an overview of the concepts commonly used in the Structured Query Language (SQL). The intent of the chapter is to provide a high-level view of the concepts. The reference material that follows provides a more detailed view.

---

### Relational Database

A *relational database* is a database that can be perceived as a set of tables and manipulated in accordance with the relational model of data. It contains a set of objects used to store, manage, and access data. Examples of such objects are tables, views, indexes, functions, triggers, and packages.

A *partitioned* relational database is a relational database where the data is managed across multiple partitions (also called nodes). This partitioning of data across partitions is transparent to users of most SQL statements. However, some DDL statements take partition information into consideration (e.g. create nodegroup).

---

### Structured Query Language (SQL)

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 base tables.

SQL statements are executed by a database manager. One of the functions of the database manager is to transform the specification of a result table into a sequence of internal operations that optimize data retrieval. The transformation occurs in two phases: preparation and 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.

---

### Embedded SQL

Embedded SQL statements are SQL statements written within application programming languages such as C and preprocessed by an SQL preprocessor before the application program is compiled. There are two types of embedded SQL: static and dynamic.

### 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 containing static SQL statements must be processed by an SQL pre-compiler before it is compiled. The precompiler turns the SQL statements into host language comments, and generates host language statements to invoke the database manager. The syntax of the SQL statements is checked during the binding process.

The preparation of an SQL application program includes precompilation, the binding of its static SQL statements to the target database, and compilation of the modified source program. The steps are specified in the *Embedded SQL Programming Guide*.

## Dynamic SQL

Programs containing embedded dynamic SQL statements must be precompiled like those containing static SQL, but unlike static SQL, the dynamic SQL statements are constructed and prepared at run time. The SQL statement text is prepared and executed using either the PREPARE and EXECUTE statements, or the EXECUTE IMMEDIATE statement. The statement can also be executed with the cursor operations if it is a SELECT statement.

---

## DB2 Call Level Interface (CLI)

The DB2 Call Level Interface is an application programming interface in which functions are provided to application programs to process dynamic SQL statements. Through the interface, applications use procedure calls at execution time to connect to databases, to issue SQL statements, and to get returned data and status information. Unlike using embedded SQL, no precompilation is required. Applications developed using this interface may be executed on a variety of databases without being compiled against each of the databases.

The DB2 CLI interface provides many features not available in embedded SQL. A few of these are:

- CLI provides function calls which support a consistent way to query and retrieve database system catalog information across the DB2 family of database management systems. This reduces the need to write database server specific catalog queries.
- Application programs written using CLI can have multiple concurrent connections to the same database.
- CLI provides support for scrollable cursors.
- Stored procedures called from application programs written using CLI can return result sets to those programs.

For a comparison between the features of CLI and Embedded Dynamic SQL, see *Road Map to DB2 Programming*.

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

---

## Interactive SQL

*Interactive* SQL statements are entered by a user through an interface like the command line processor or the command center. These statements are processed as dynamic SQL statements. For example, an interactive SELECT statement can be processed dynamically using the DECLARE CURSOR, PREPARE, DESCRIBE, OPEN, FETCH, and CLOSE statements.

The *Command Reference* lists the commands that can be issued using the command line processor or similar facilities and products.

---

## Schemas

A *schema* is a collection of named objects. Schemas provide a logical classification of objects in the database. Some of the objects that a schema may contain include tables, views, triggers, functions and packages.

A schema is also an object in the database. It is explicitly created using the CREATE SCHEMA statement with a user recorded as owner. It can also be implicitly created when another object is created, provided the user has IMPLICIT\_SCHEMA authority.

A *schema name* is used as the high-order part of a two-part object name. An object that is contained in a schema is assigned to a schema when the object is created. The schema to which it is assigned is determined by the name of the object if specifically qualified with a schema name or by the default schema name if not qualified.

For example, a user with DBADM authority creates a schema called C for user A.

```
CREATE SCHEMA C AUTHORIZATION A
```

User A can then issue the following statement to create a table called X in schema C:

```
CREATE TABLE C.X (COL1 INT)
```

## Controlling Use of Schemas

When a database is created, all users have IMPLICIT\_SCHEMA authority. This allows any user to create objects in any schema that does not already exist. An implicitly created schema allows any user to create other objects in this schema.<sup>1</sup>

If IMPLICIT\_SCHEMA authority is revoked from PUBLIC, schemas are either explicitly created using the CREATE SCHEMA statement or implicitly created by users (such as those with DBADM authority) who are granted IMPLICIT\_SCHEMA authority. While revoking IMPLICIT\_SCHEMA authority from PUBLIC increases control over the use of schema names, it may result in authorization errors in existing applications when they attempt to create objects.

---

<sup>1</sup> The default privileges on an implicitly created schema provide upward compatibility with previous versions. Alias, distinct type, function and trigger creation is extended to implicitly created schemas.

There are also privileges associated with a schema that control which users have the privilege to create, alter and drop objects in the schema. A schema owner is initially given all of these privileges on a schema with the ability to grant them to others. An implicitly created schema is owned by the system and all users are initially given the privilege to create objects in such a schema. A user with DBADM or SYSADM authority can change the privileges held by users on any schema. Therefore, access to create, alter and drop objects in any schema (even one that is implicitly created) can be controlled.

---

## Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows. The rows are not necessarily ordered within a table (order is determined by the application program). 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 data 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.

A *base table* is created with the CREATE TABLE statement and is used to hold persistent user data. A *result table* is a set of rows that the database manager selects or generates from one or more base tables to satisfy a query.

A summary table is a table that is defined by a query that is also used to determine the data in the table. Summary tables can be used to improve the performance of queries. If the database manager determines that a portion of a query could be resolved using a summary table, the query may be rewritten by the database manager to use the summary table. This decision is based on certain settings such as CURRENT REFRESH AGE and CURRENT QUERY OPTIMIZATION special registers.

A table can have the data type of each column defined separately, or have the types for the columns based on the attributes of a user-defined structured type. This is called a *typed table*. A user-defined structured type may be part of a type hierarchy. A *subtype* is said to inherit attributes from its *supertype*. Similarly, a typed table can be part of a table hierarchy. A *subtable* is said to inherit columns from its *supertable*. Note that the term *subtype* applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy. Similarly the term *subtable* applies to a typed table and all typed tables that are below it in the table hierarchy. A *proper subtable* of a table T is a table below T in the table hierarchy.

---

## 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 a SELECT statement that is executed whenever the view is referenced in an SQL statement. Thus, a view can be thought of as having columns and rows just like a base table. For retrieval, all views can be used just like base tables. Whether a view can be used in an insert,



update, or delete operation depends on its definition as explained in the description of CREATE VIEW. (See “CREATE VIEW” on page 582 for more information.)

When the column of a view is directly derived from a column of a base table, that column inherits any constraints that apply to the 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. Also, if the base table of a view is a parent table, DELETE and UPDATE operations using that view are subject to the same rules as DELETE and UPDATE operations on the base table.

A view can have the data type of each column derived from the result table, or have the types for the columns based on the attributes of a user-defined structured type. This is called a *typed view*. Similar to a typed table, a typed view can be part of a view hierarchy. A *subview* is said to inherit columns from its *superview*. The term *subview* applies to a typed view and all typed views that are below it in the view hierarchy. A *proper subview* of a view V is a view below V in the typed view hierarchy.

A view may become inoperative, in which case it is no longer available for SQL statements.

---

## Aliases

An *alias* is an alternate name for a table or view. It can be used to reference a table or view in those cases where an existing table or view can be referenced.<sup>2</sup> Like tables and views, an alias may be created, dropped, and have comments associated with it. Unlike tables, aliases may refer to each other in a process called chaining. Aliases are publicly referenced names so no special authority or privilege is required to use an alias. Access to the tables and views referred to by the alias, however, still require the appropriate authorization for the current context.

In addition to table aliases, there are other types of aliases such as a database alias or a network alias.

---

## Indexes

An *index* is an ordered set of pointers to rows of a base table. Each index is based on the values of data in one or more table columns. An index is an object that is separate from the data in the table. When an index is created, the database manager builds this structure and maintains it automatically.

Indexes are used by the database manager to:

- Improve performance. In most cases, access to data is faster than without an index.

---

<sup>2</sup> An alias cannot be used in all contexts. For example, it cannot be used in the check condition of a check constraint.

An index cannot be created for a view. However, an index created for a table on which a view is based may improve the performance of operations on the view.

- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

---

## Keys

A *key* is a set of columns that can be used to identify or access a particular row or rows. The key is identified in the description of a table, index, or referential constraint. The same column can be part of more than one key.

A key composed of more than one column is called a *composite key*. In a table with a composite key, the ordering of the columns within the composite key 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 primary key” means that each component of the value of the foreign key must be equal to the corresponding component of the value of the primary key.

## Unique Keys

A *unique key* is a key that is constrained so that no two of its values are equal. The columns of a unique key cannot contain null values. The constraint is enforced by the database manager during the execution of any operation that changes data values, such as INSERT or UPDATE. The mechanism used to enforce the constraint is called 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. See “Unique Constraints” on page 13 for a more detailed description.

## Primary Keys

A *primary key* is a special case of a unique key. A table cannot have more than one primary key. See “Unique Keys” for a more detailed description.

## Foreign Keys

A *foreign key* is a key that is specified in the definition of a referential constraint. See “Referential Constraints” on page 14 for a more detailed description.

## Partitioning Keys

A *partitioning key* is a key that is part of the definition of a table in a partitioned database. The partitioning key is used to determine the partition on which the row of data is stored. If a partitioning key is defined, unique keys and primary keys must include the same columns as the partitioning key (they may have more columns).

---

## Constraints

A *constraint* is a rule that the database manager enforces.

There are three types of constraints:

- A *unique constraint* is a rule that forbids duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, a unique constraint could be defined on the supplier identifier in the supplier table to ensure that the same supplier identifier is not given to two suppliers.
- A *referential constraint* is a logical rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's name changes. A referential constraint could be defined stating that the ID of the supplier in a table must match a supplier id in the supplier information. This constraint prevents inserts, updates or deletes that would otherwise result in missing supplier information.
- A *table check constraint* sets restrictions on data added to a specific table. For example, it could define the salary level for an employee to never be less than \$20,000.00 when salary data is added or updated in a table containing personnel information.

Referential and table check constraints may be turned on or off. Loading large amounts of data into the database is typically a time to turn off checking the enforcement of a constraint. The details of setting constraints on or off are discussed in "SET CONSTRAINTS" on page 709.

## Unique Constraints

A *unique constraint* is the rule that the values of a key are valid only if they are unique within the table. Unique constraints are optional and can be defined in the CREATE TABLE or ALTER TABLE statement using the PRIMARY KEY clause or the UNIQUE clause. The columns specified in a unique constraint must be defined as NOT NULL. A unique index is used by the database manager to enforce the uniqueness of the key during changes to the columns of the unique constraint.

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

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

When a unique constraint is defined in a CREATE TABLE statement, a unique index is automatically created by the database manager and designated as a primary or unique system-required index.

When a unique constraint is defined in an ALTER TABLE statement and an index exists on the same columns, that index is designated as unique and system-required. If such an index does not exist, the unique index is automatically created by the database manager and designated as a primary or unique system-required index.

Note that there is a distinction between defining a unique constraint and creating a unique index. Although both enforce uniqueness, a unique index allows nullable columns and generally cannot be used as a parent key.

## Referential Constraints

*Referential integrity* is the state of a database in which all values of all foreign keys are valid. A *foreign key* is a column or set of columns in a table whose values are required to match at least one primary key or unique key value of a row of its parent table. A *referential constraint* is the rule that the values of the foreign key are valid only if:

- they appear as values of a parent key, or
- some component of the foreign key is null.

The table containing the parent key is called the *parent table* of the referential constraint, and the table containing the foreign key is said to be a *dependent* of that table.

Referential constraints are optional and can be defined in CREATE TABLE statements and ALTER TABLE statements. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, DELETE, ALTER TABLE ADD CONSTRAINT, and SET CONSTRAINTS statements. The enforcement is effectively performed at the completion of the statement.

Referential constraints with a delete or update rule of RESTRICT are enforced before all other referential constraints. Referential constraints with a delete or update rule of NO ACTION behave like RESTRICT in most cases. However, in certain SQL statements there can be a difference.

Note that referential integrity, check constraints and triggers can be combined in execution. For further information on the combination of these elements, see Appendix I, “Interaction of Triggers and Constraints” on page 887.

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

<b>Parent key</b>	A primary key or unique key of a referential constraint.
<b>Parent row</b>	A row that has at least one dependent row.
<b>Parent table</b>	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. A parent table can also be a dependent table.
<b>Dependent table</b>	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.
<b>Descendent table</b>	A table is a descendent of table T if it is a dependent of T or a descendent of a dependent of T.
<b>Dependent row</b>	A row that has at least one parent row.
<b>Descendent row</b>	A row is a descendent of row p if it is a dependent of p or a descendent of a dependent of p.
<b>Referential cycle</b>	A set of referential constraints such that each table in the set is a descendent of itself.

<b>Self-referencing row</b>	A row that is a parent of itself.
<b>Self-referencing table</b>	A table that is a parent and a dependent in the same referential constraint. The constraint is called a <i>self-referencing constraint</i> .

### Insert Rule

The insert rule of a referential constraint is that a non-null insert value of the foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is null if any component of the value is null. This rule is implicit when a foreign key is specified.

### Update Rule

The update rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION and RESTRICT. The update rule applies when a row of the parent or a row of the dependent table is updated.

In the case of a parent row, when a value in a column of the parent key is updated:

- if any row in the dependent table matched the original value of the key, the update is rejected when the update rule is RESTRICT
- if any row in the dependent table does not have a corresponding parent key when the update statement is completed (excluding after triggers), the update is rejected when the update rule is NO ACTION.

In the case of a dependent row, the update rule that is implicit when a foreign key is specified is NO ACTION. NO ACTION means that a non-null update value of a foreign key must match some value of the parent key of the parent table when the update statement is completed.

The value of a composite foreign key is null if any component of the value is null.

### Delete Rule

The delete rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION, RESTRICT, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

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 (defined below) 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 dependents of p in D
- SET NULL, each nullable column of the foreign key of each dependent 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 may affect rows of these tables:

- If table 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 may 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 may 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 may 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.

---

## Table Check Constraints

A *table check constraint* is a rule that specifies the values allowed in one or more columns of every row of a table. They are optional and can be defined using the SQL statements CREATE TABLE and ALTER TABLE. The specification of table check constraints is a restricted form of a search condition. One of the restrictions is that a column name in a table check constraint on table *T* must identify a column of *T*.

A table can have an arbitrary number of table check constraints. They are enforced when:

- a row is inserted into the table
- a row of the table is updated.

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

When one or more table check constraints are defined in the ALTER TABLE statement for a table with existing data, the existing data is checked against the new condition before the ALTER TABLE statement succeeds. The table can be placed in *check pending state* which will allow the ALTER TABLE statement to succeed without checking the data. The SET CONSTRAINT statement is used to place the table into

check pending state. It is also used to resume the checking of each row against the constraint.

---

## Triggers

A *trigger* defines a set of actions that are executed at, or triggered by, a delete, insert, or update operation 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 can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts.

Triggers are a useful mechanism to define and enforce *transitional* business rules which are rules that involve different states of the data (for example, salary cannot be increased by more than 10 percent). For rules that do not involve more than one state of the data, check and referential integrity constraints should be considered.

Using triggers places the logic to enforce the business rules in the database and relieves the applications using the tables from having to enforce it. Centralized logic enforced on all the tables means easier maintenance, since no application program changes are required when the logic changes.

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

There are a number of criteria that are defined when creating a trigger which are used to determine when a trigger should be activated.

- The *subject table* defines the table for which the trigger is defined.
- The *trigger event* defines a specific SQL operation that modifies the subject table. The operation could be delete, insert or update.
- The *trigger activation time* defines whether the trigger should be activated before or after the trigger event is performed on the subject table.

The statement that causes a trigger to be activated will include a *set of affected rows*. These are the rows of the subject table that are being deleted, inserted or updated. The *trigger granularity* defines whether the actions of the trigger will be performed once for the statement or once for each of the rows in the set of affected rows.

The *triggered action* consists of an optional search condition and a set of SQL statements that are executed whenever the trigger is activated. The SQL statements are only executed if the search condition evaluates to true. When the trigger activation time is before the trigger event, triggered action can include statements that select, set transition variables, and signal sqlstates. When the trigger activation time is after the trigger event, triggered action can include statements that select, update, insert, delete, and signal sqlstates.

The triggered action may refer to the values in the set of affected rows. This is supported through the use of *transition variables*. 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 (prior to the update) or the new value (after the update). The new value can also be changed using the SET transition-variable statement in before update or insert triggers. Another means of referring to the values in the set of affected rows is using *transition tables*. Transition tables also use the names 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. As with transition variables, a transition table can be defined for the old values and the new values but only in after triggers.

Multiple triggers can be specified for a combination of table, event, or activation time. The order in which the triggers are activated is the same as the order in which they were created. Thus, the most recently created trigger will be the last trigger activated.

The activation of a trigger may cause *trigger cascading*. This is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions may also cause updates as a result of the original modification, or as a result of referential integrity delete rules which may result in the activation of additional triggers. With trigger cascading, a significant chain of triggers and referential integrity delete rules may be activated causing significant change to the database as a result of a single delete, insert or update statement.

---

## Event Monitors

An *event monitor* tracks specific data as the result of an event. For example, starting the database might be an event that causes an event monitor to track the number of users on the system by taking an hourly snapshot of authorization IDs using the database.

Event monitors are activated or deactivated by a statement (SET EVENT MONITOR STATE). A function (EVENT\_MON\_STATE) can be used to find the current state of an event monitor; that is, if it is active or not active.

---

## Queries

A *query* is a component of certain SQL statements that specifies a (temporary) result table.

---

## Table Expressions

A *table expression* creates a (temporary) result table from a simple query. Clauses further refine the result table. For example, a table expression could be a query that selects all the managers from several departments and further specifies that they have over 15 years of working experience and are located at the New York branch office.



## Common Table Expressions

A *common table expression* is like a temporary view within a complex query, and can be referenced in other places within the query; for example, in place of a view, to avoid creating the view. Each use of a specific common table expression within a complex query shares the same temporary view.

Recursive use of a common table expression within a query can be used to support applications such as bill of materials (BOM), airline reservation systems, and network planning. A set of examples from a BOM application is contained in Appendix M, “Recursion Example: Bill of Materials” on page 961.

---

## Packages

A *package* is an object that contains control structures (called sections) used to execute SQL statements. Packages are produced during program preparation. The sections created for static SQL can be thought of as the bound or operational form of SQL statements. The sections created for dynamic SQL can be thought of as placeholder control structures which are used at execution time. All sections in a package are derived from the SQL statements embedded in a single source file. See the *Embedded SQL Programming Guide* for more information on packages.

---

## Catalog Views

The database manager maintains a set of views and base tables that contain information about the data under its control. These views and base tables are collectively known as the *catalog*. They contain information about objects in the database such as tables, views, indexes, packages and functions.

The catalog views are like any other database views. SQL statements can be used to look at the data in the catalog views in the same way that data is retrieved from any other view in the system. The database manager ensures that the catalog contains accurate descriptions of the objects in the database at all times. A set of updatable catalog views can be used to modify certain values in the catalog (see “Updatable Catalog Views” on page 774).

Statistical information is also contained in the catalog. The statistical information is updated by utilities executed by an administrator, or through update statements by appropriately authorized users.

The catalog views are listed in Appendix D, “Catalog Views” on page 773.

---

## Application Processes, Concurrency, and Recovery

All SQL programs execute as part of an *application process* or agent. An application process involves the execution of one or more programs, and is the unit to which the database manager allocates resources and locks. Different application processes may involve the execution of different programs, or different executions of the same program.

More than one application process may request access to the same data at the same time. *Locking* is the mechanism used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously.

The database manager acquires locks in order to prevent uncommitted changes made by one application process from being accidentally perceived by any other. The database manager releases all locks it has acquired and retained on behalf of an application process when that process ends, but an application process itself should explicitly request that locks be released sooner. This operation is called *commit* and it writes the changes to the database.

The database manager provides a means of *backing 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* or lock timeout situation. An application process itself, however, can explicitly request that its database changes be backed out. This operation is called *rollback*.

A *unit of work* is a recoverable sequence of operations within an application process. At any time, an application process has a single unit of work<sup>3</sup>, but the life of an application process may involve many units of work as a result of commit or 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 termination of the application process. A unit of work is ended by a commit operation, a 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 and they can be backed out.<sup>4</sup> Once committed, these database changes are accessible by other application processes and can no longer be backed out by a rollback.

Locks acquired by the database manager on behalf of an application process are held until the end of a unit of work. The exception to this rule is with a read stability or cursor stability isolation level, or an uncommitted read level, in which case the lock is released as the cursor moves from row to row (see “Isolation Level” on page 22).

The initiation and termination of a unit of work define points of consistency within an application process. 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

---

<sup>3</sup> DB2 CLI supports a connection mode called *concurrent transactions* which supports multiple connections, each of which is an independent transaction. Furthermore, an application can have multiple concurrent connections to the same database, which is not possible at all with DB2 embedded SQL.

<sup>4</sup> Except for isolation level uncommitted read, described in “Uncommitted Read (UR)” on page 24.

commit operation can be used to end the unit of work, thereby making the changes available to other application processes.

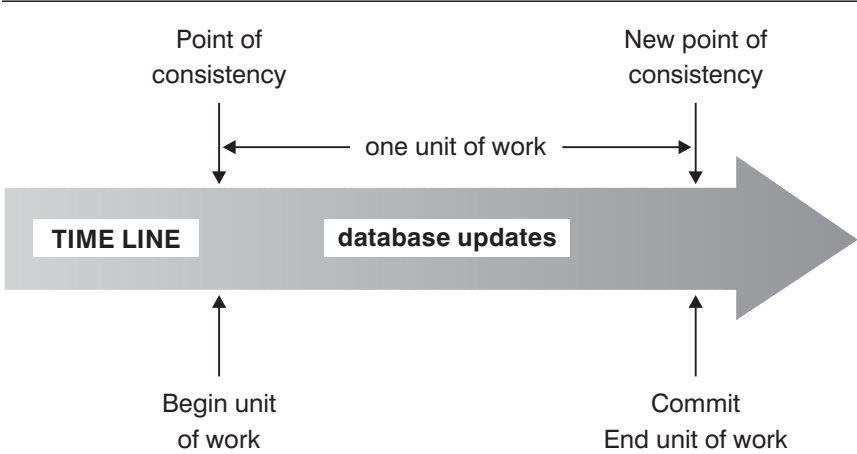


Figure 1. Unit of Work with a Commit Statement

If a failure occurs before the unit of work ends, the database manager will roll back uncommitted changes to restore the data consistency that it assumes existed when the unit of work was initiated.

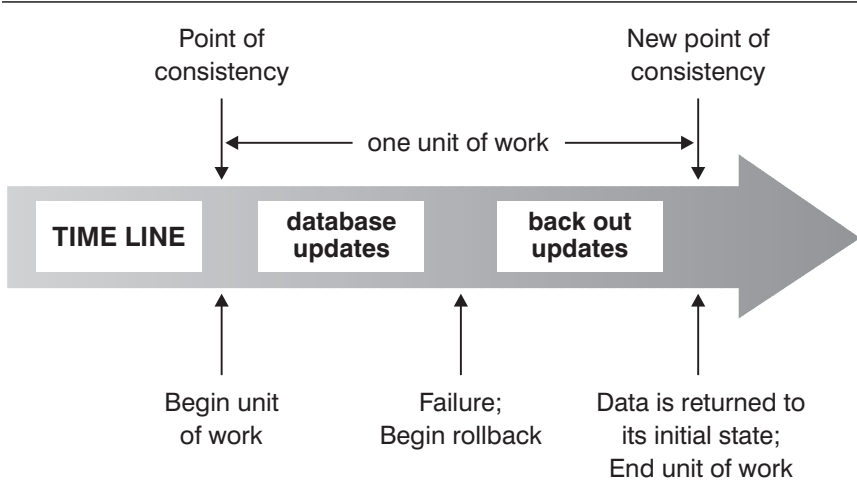


Figure 2. Unit of Work with a Rollback Statement

**Note:** An application process is never prevented from performing operations because of its own locks.

---

## Isolation Level

The *isolation level* associated with an application process defines the degree of isolation of that application process from other concurrently executing application processes. The isolation level of an application process, P, therefore specifies:

- The degree to which rows read and updated by P are available to other concurrently executing application processes
- The degree to which update activity of other concurrently executing application processes can affect P.

Isolation level is specified as an attribute of a package and applies to the application processes that use the package. The isolation level is specified in the program preparation process. Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. The database manager supports three types of locks:

- |                  |                                                                                                                                                                                                                                                      |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Share</b>     | Limits concurrent application processes to read-only operations on the data.                                                                                                                                                                         |
| <b>Exclusive</b> | Prevents concurrent application processes from accessing the data in any way except for application processes with an isolation level of <i>uncommitted read</i> , which can read but not modify the data. (See “Uncommitted Read (UR)” on page 24.) |
| <b>Update</b>    | Limits concurrent application processes to read-only operations on the data providing these processes have not declared they might update the row. The database manager assumes the process looking at the row presently may update the row.         |

The following descriptions of isolation levels refer to locking data in row units. Logically, locking occurs at the base table row. The database manager, however, can escalate a lock to a higher level. An application process is guaranteed at least the minimum requested lock level.

The DB2 Universal Database database manager supports four isolation levels. Regardless of the isolation level, the database manager places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed by this application process during a unit of work is not changed by any other application processes until the unit of work is complete. The isolation levels are:

### Repeatable Read (RR)

Level RR ensures that:

- Any row read during a unit of work <sup>5</sup> is not changed by other application processes until the unit of work is complete. <sup>6</sup>

---

<sup>5</sup> The rows must be read in the same unit of work as the corresponding OPEN statement. See WITH HOLD in “DECLARE CURSOR” on page 595.

- Any row changed by another application process cannot be read until it is committed by that application process.

RR does not allow phantom rows (see Read Stability) to be seen.

In addition to any exclusive locks, an application process running at level RR acquires at least share locks on all the rows it references. Furthermore, the locking is performed so that the application process is completely isolated <sup>6</sup> from the effects of concurrent application processes.

## Read Stability (RS)

Like level RR, level RS ensures that:

- Any row read during a unit of work <sup>5</sup> is not changed by other application processes until the unit of work is complete <sup>7</sup>
- Any row changed by another application process cannot be read until it is committed by that application process.

Unlike RR, RS does not completely isolate the application process from the effects of concurrent application processes. At level RS, application processes that issue the same query more than once might see additional rows. These additional rows are called *phantom rows*.

For example, a phantom row can occur in the following situation:

1. Application process P1 reads the set of rows *n* that satisfy some search condition.
2. Application process P2 then INSERTs one or more rows that satisfy the search condition and COMMITs those INSERTs.
3. P1 reads the set of rows again with the same search condition and obtains both the original rows and the rows inserted by P2.

In addition to any exclusive locks, an application process running at level RS acquires at least share locks on all the qualifying rows.

## Cursor Stability (CS)

Like the RR level:

- CS ensures that any row that was changed by another application process cannot be read until it is committed by that application process.

Unlike the RR level:

---

<sup>6</sup> Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable read and phantoms no longer apply to any previously accessed rows if the cursor is reopened.

<sup>7</sup> Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable read no longer apply to any previously accessed rows if the cursor is reopened.

- CS only ensures that the current row of every updatable cursor is not changed by other application processes. Thus, the rows that were read during a unit of work can be changed by other application processes.

In addition to any exclusive locks, an application process running at level CS has at least a share lock for the current row of every cursor.

### Uncommitted Read (UR)

For a SELECT INTO, FETCH with a read-only cursor, fullselect used in an INSERT, row fullselect in an UPDATE, or scalar fullselect (wherever used), level UR allows:

- Any row that is read during the unit of work to be changed by other application processes.
- Any row that was changed by another application process to be read even if the change has not been committed by that application process.

For other operations, the rules of level CS apply.

### Comparison of Isolation Levels

A comparison of the four isolation levels can be found on Appendix H, “Comparison of Isolation Levels” on page 885.

---

## Distributed Relational Database

A *distributed relational database* consists of a set of tables and other objects that are spread across different but interconnected computer systems. Each computer system has a relational database manager to manage the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a given database manager to execute SQL statements on another computer system.

Distributed relational databases are built on formal requester-server protocols and functions. An *application requester* supports the application end of a connection. It transforms a database request from the application into communication protocols suitable for use in the distributed database network. These requests are received and processed by an *application server* at the other end of the connection. Working together, the application requester and application server handle the communication and location considerations so that the application is isolated from these considerations and can operate as if it were accessing a local database. A simple distributed relational database environment is illustrated in Figure 3 on page 25.

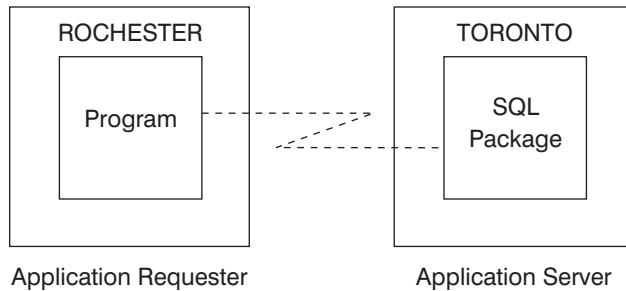


Figure 3. A Distributed Relational Database Environment

For more information on Distributed Relational Database Architecture (DRDA) communication protocols, see *Distributed Relational Database Architecture Reference* SC26-4651.

## Application Servers

An application process must be connected to the application server of a database manager before SQL statements that reference tables or views can be executed. A CONNECT statement establishes a connection between an application process and its server. An application process has only one server at any time<sup>3</sup>, but the server can change when a CONNECT statement is executed.

The application server can be local to or remote from the environment where the process is initiated. (An application server is present, even when not using distributed relational databases.) This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement. For a description of local directories, see the *Administration Guide*.

To execute a static SQL statement that references tables or views, the application server uses the bound form of the statement. This bound statement is taken from a package that the database manager previously created through a bind operation.

For the most part, an application can use the statements and clauses that are supported by the database manager of the application server to which it is currently connected, even though that application might be running via the application requester of a database manager that does not support some of those statements and clauses.

See the *IBM SQL Reference* for information on considerations for using a distributed relational database on DB2 application servers on other platforms.

## CONNECT (Type 1) and CONNECT (Type 2)

There are two types of CONNECT statements:

- CONNECT (Type 1) supports the single database per unit of work (Remote Unit of Work) semantics. See "CONNECT (Type 1)" on page 432.

- CONNECT (Type 2) supports the multiple database per unit of work (Application-Directed Distributed Unit of Work) semantics. See “CONNECT (Type 2)” on page 439.

## Remote Unit of Work

The *remote unit of work* facility provides for the remote preparation and execution of SQL statements. An application process at computer system A can connect to an application server at computer system B and, within one or more units of work, execute any number of static or dynamic SQL statements that reference objects at B. After ending a unit of work at B, the application process can connect to an application server at computer system C, and so on.

Most SQL statements can be remotely prepared and executed with the following restrictions:

- All objects referenced in a single SQL statement must be managed by the same application server
- All of the SQL statements in a unit of work must be executed by the same application server

## Remote Unit of Work Connection Management

This section outlines the connection states that an application process may enter.

### **Connection States:**

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

- Connectable and connected
- Unconnectable and connected
- Connectable and unconnected
- Implicitly connectable (if implicit connect is available).

If implicit connect is available (see Figure 4 on page 28), the application process is initially in the *implicitly connectable* state. If implicit connect is not available (see Figure 5 on page 29), the application process is initially in the *connectable and unconnected* state.

Availability of implicit connect is determined by installation options, environment variables, and authentication settings. See the *Quick Beginnings* for information on setting implicit connect on installation and the *Administration Guide* for information on environment variables and authentication settings.

### **The implicitly connectable state:**

If implicit connect is available, this is the initial state of an application process. The CONNECT RESET statement causes a transition to this state. Issuing a COMMIT or ROLLBACK statement in the unconnectable and connected state followed by a DISCONNECT statement in the connectable and connected state also results in this state.

### **The connectable and connected state:**



An application process is connected to an application server and CONNECT statements can be executed.

If implicit connect is available:

- The application process enters this state when a CONNECT TO statement or a CONNECT without operands statement is successfully executed from the connectable and unconnected state.
- The application process may also enter this state from the implicitly connectable state if any SQL statement other than CONNECT RESET, DISCONNECT, SET CONNECTION, or RELEASE is issued.

Whether or not implicit connect is available, this state is entered when:

- A CONNECT TO statement is successfully executed from the connectable and unconnected state.
- A COMMIT or ROLLBACK statement is successfully issued or a forced rollback occurs from the unconnectable and connected state.

***The unconnectable and connected state:***

An application process is connected to an application server, but a CONNECT TO statement cannot be successfully executed to change application servers. The process enters this state from the connectable and connected state when it executes any SQL statement other than the following statements: CONNECT TO, CONNECT with no operand, CONNECT RESET, DISCONNECT, SET CONNECTION, RELEASE, COMMIT or ROLLBACK.

***The connectable and unconnected state:***

An application process is not connected to an application server. The only SQL statement that can be executed is CONNECT TO, otherwise an error (SQLSTATE 08003) is raised.

Whether or not implicit connect is available:

- The application process enters this state if an error occurs when a CONNECT TO statement is issued or an error occurs in a unit of work which causes the loss of a connection and a rollback. An error caused because the application process is not in the connectable state or the server-name is not listed in the local directory does not cause a transition to this state.

If implicit connect is not available:

- the CONNECT RESET and DISCONNECT statements cause a transition to this state.

***State Transitions*** are shown in the following diagrams.

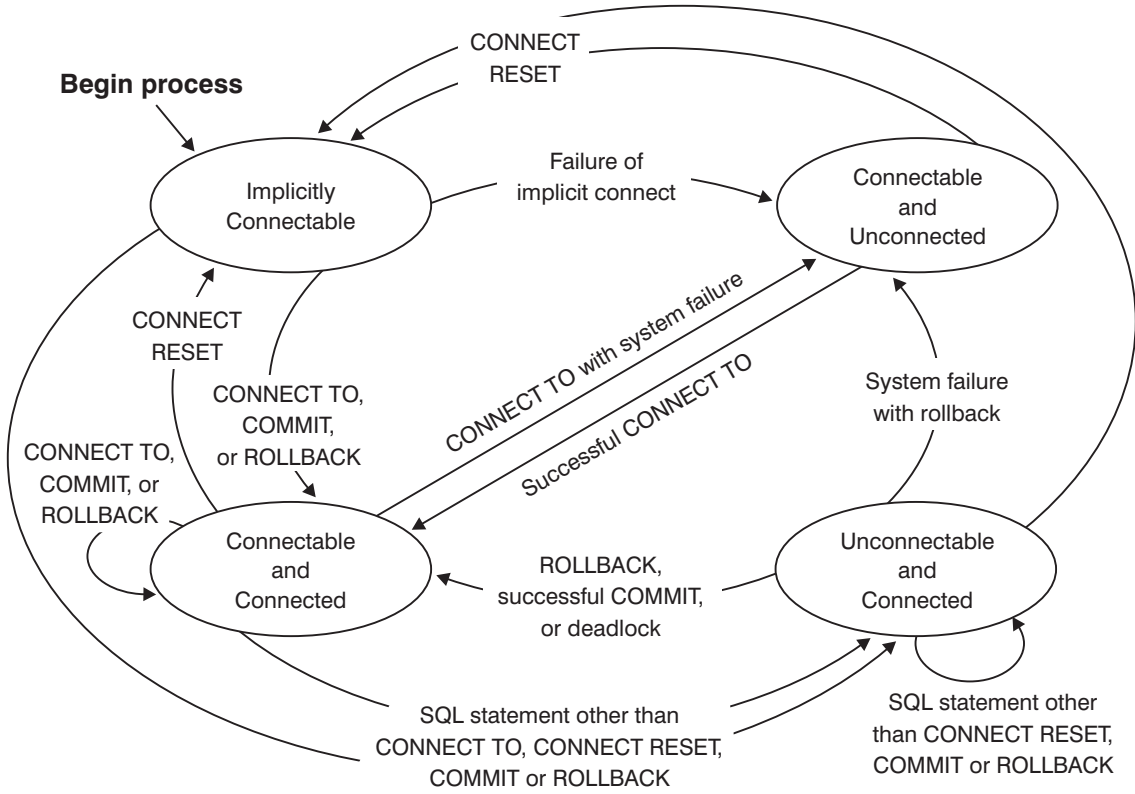


Figure 4. Connection State Transitions If Implicit Connect Is Available

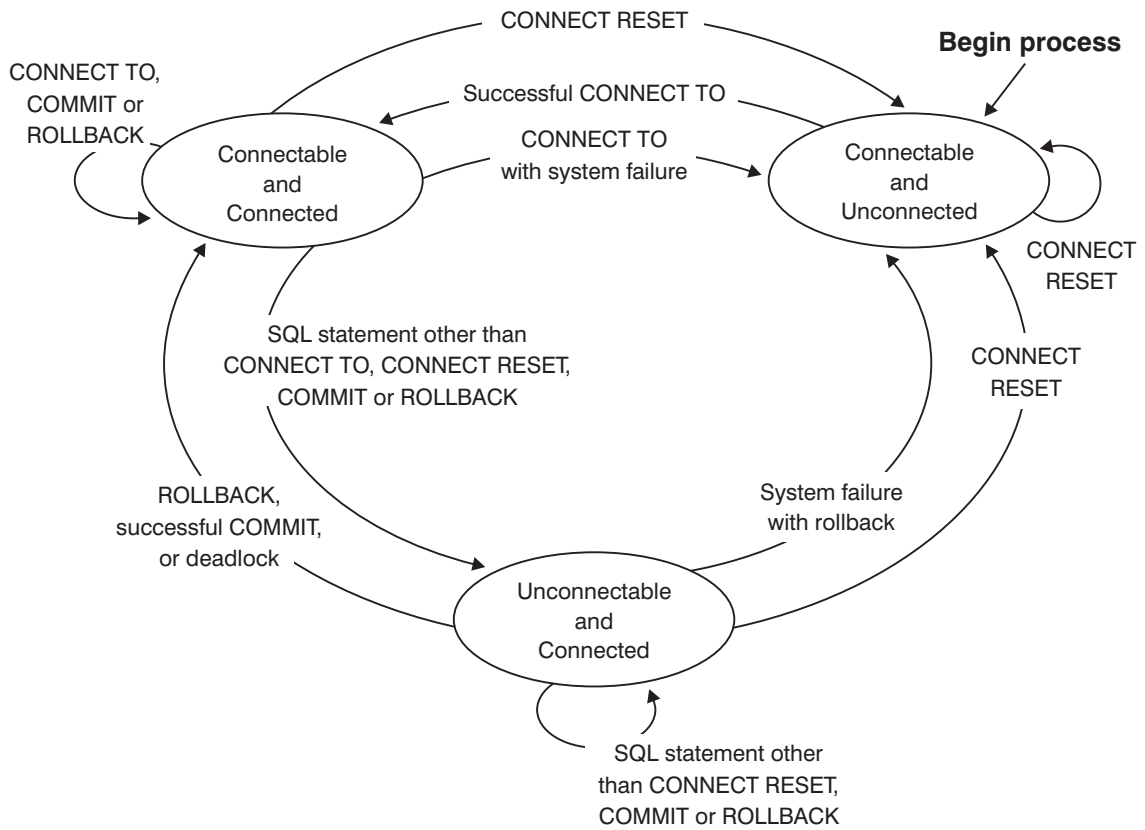


Figure 5. Connection State Transitions If Implicit Connect Is Not Available

**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 consecutive CONNECT RESET statements.
- It is an error to execute any SQL statement other than CONNECT TO, CONNECT RESET, CONNECT with no operand, SET CONNECTION, RELEASE, COMMIT, or ROLLBACK, and then execute a CONNECT TO statement. To avoid the error, a CONNECT RESET, DISCONNECT (preceded by a COMMIT or ROLLBACK statement), COMMIT, or ROLLBACK statement should be executed before executing the CONNECT TO.

## Application-Directed Distributed Unit of Work

The *application-directed distributed unit of work facility* also provides for the remote preparation and execution of SQL statements in the same fashion as remote unit of work. An application process at computer system A can connect to an application server at computer system B by issuing a `CONNECT` or `SET CONNECTION` statement. The application process can then execute any number of static and dynamic SQL statements that reference objects at B before ending the unit of work. All objects referenced in a single SQL statement must be managed by the same application server. However, unlike remote unit of work, any number of application servers can participate in the same unit of work. A commit or rollback operation ends the unit of work.

## Application-Directed Distributed Unit of Work Connection Management

An application-directed distributed unit of work uses a Type 2 connection. A Type 2 connection connects an application process to the identified application server and establishes the rules for application-directed distributed unit of work.

## Overview of Application Process and Connection States

At any time a type 2 application process:

- Is always connectable
- Is in the *connected* state or *unconnected* state.
- Has a set of zero or more connections.

Each connection of an application process is uniquely identified by the database alias of the application server of the connection.

At any time an individual connection has one of the following sets of connection states:

- *current* and *held*
- *current* and *release-pending*
- *dormant* and *held*
- *dormant* and *release-pending*

**Initial States and State Transitions:** A type 2 application process is initially in the *unconnected state* and does not have any connections.

A connection initially is in the *current* and *held state*.

The following diagram shows the state transitions:

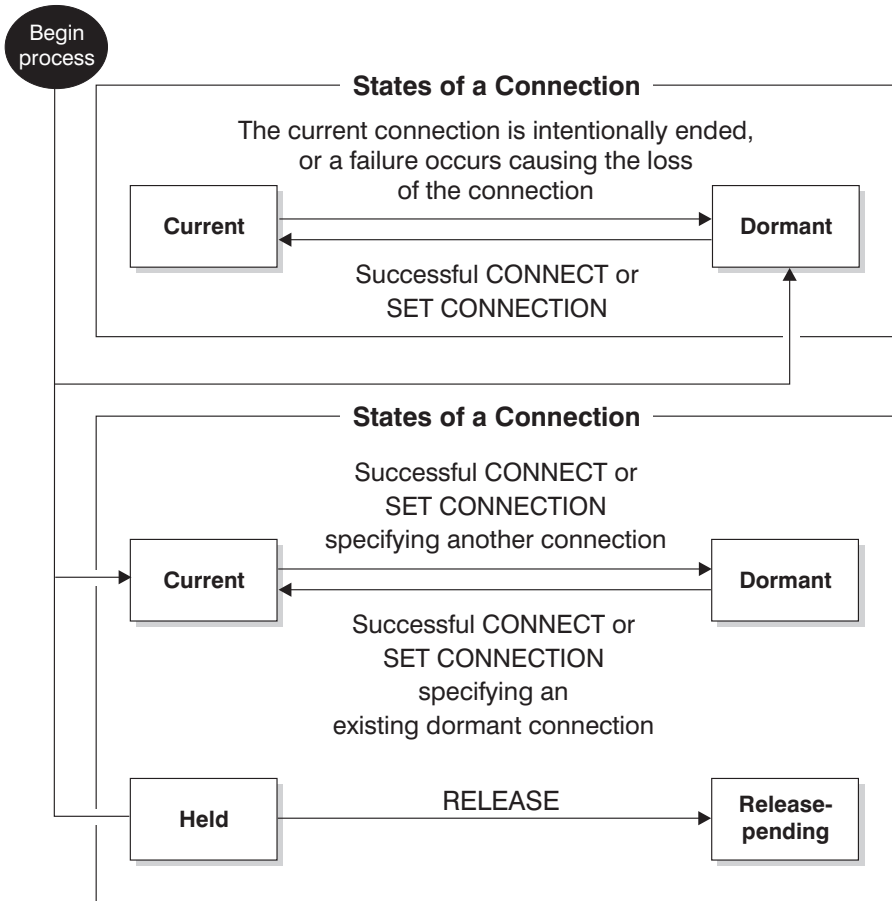


Figure 6. Application-Directed Distributed Unit of Work Connection and Application Process Connection State Transitions

**Application Process Connection States:** A different application server can be established by the explicit or implicit execution of a CONNECT statement.<sup>8</sup> The following rules apply:

- An application process cannot have more than one connection to the same application server at the same time. Note that DB2 CLI does not have this restriction. (DB2 CLI has its own connection type. DB2 CLI also can support a Type 2 connection as discussed, in which case this restriction does apply.) See *Administration Guide* for information on support of multiple connections to the same DB2 Universal Database at the same time.

<sup>8</sup> Note that a Type 2 implicit connection is more restrictive than a Type 1. See "CONNECT (Type 2)" on page 439 for details.

- When an application process executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections of the application process.
- When an application process executes a CONNECT statement, and the SQLRULES(STD) option is in effect the specified server name must not be an existing connection in the set of connections of the application process. See “Options that Govern Distributed Unit of Work Semantics” on page 33 for a description of the SQLRULES option.

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

An application process in the unconnected state enters the connected state when it successfully executes a CONNECT or SET CONNECTION statement. If there is no connection in the application but SQL statements are issued, an implicit connect will be made provided the DB2DBDFT environment variable has been defined with a default database.

**If an application process does not have a current connection,** the application process is in the *unconnected* state. The only SQL statements that can be executed are CONNECT, DISCONNECT ALL, DISCONNECT specifying a database, SET CONNECTION, RELEASE, COMMIT and ROLLBACK.

An application process in the *connected state* enters the *unconnected state* when its current 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 connection. Connections are intentionally ended either by the successful execution of a DISCONNECT statement or by the successful execution of a commit operation when the connection is in the *release-pending state*. Different options specified in the DISCONNECT precompiler option affect intentionally ending a connection. If set to AUTOMATIC, then all connections are ended. If set to CONDITIONAL, then all connections that do not have open WITH HOLD cursors are ended.

**States of a Connection:** If an application process executes a CONNECT statement and the server name is known to the application requester and is not in the set of existing connections of the application process, then:

- the current connection is placed into the *dormant state*, and
- the server name is added to the set of connections, and
- the new connection is placed into both the *current state* and the *held state*.

If the server name is already in the set of existing connections of the application process and the application is precompiled with the option SQLRULES(STD), an error (SQLSTATE 08002) is raised.

- **Held and Release-pending States:** The RELEASE statement controls whether a connection is in the held or release-pending state. A *release-pending state* means

that a disconnect is to occur for the connection at the next successful commit operation (a rollback has no effect on connections). A *held state* means that a connection is not to be disconnected at the next operation. All connections are initially in the held state and may be moved into the release-pending state using the RELEASE statement. Once in the release-pending state, a connection cannot be moved back to the held state. A connection will remain in a release-pending state across unit of work boundaries if a ROLLBACK statement is issued or if an unsuccessful commit operation results in a rollback operation.

Even if a connection is not explicitly marked for release, it may still be disconnected by a commit operation if the commit operation satisfies the conditions of the DISCONNECT precompiler option.

- **Current and Dormant States:** Regardless of whether a connection is in the *held state* or the *release-pending state*, a connection can also be in the *current state* or *dormant state*. A *current state* means that the connection is the one used for SQL statements that are executed while in this state. A *dormant state* means that the connection is not current. The only SQL statements which can flow on a dormant connection are COMMIT and ROLLBACK; or DISCONNECT and RELEASE, which can specify either ALL (for all connections) or a specific database name. The SET CONNECTION and CONNECT statements change the connection for the named server into the *current state* while any existing connections are either placed or remain in the *dormant state*. At any point in time, only one connection can be in the *current state*. When a dormant connection becomes current in the same unit of work, the state of all locks, cursors, and prepared statements will remain the same and reflect their last use when the connection was current.

**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. For example, if the application process executes a RELEASE statement, any open cursors will be closed when the connection is ended during the next commit operation.

A connection can also be ended because of a communications failure. The application process is placed in the unconnected state if the connection ended was the current one.

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

## Options that Govern Distributed Unit of Work Semantics

The semantics of type 2 connection management are determined by a set of precompiler options. These are summarized briefly below with the defaults indicated by bold and underlined text. For details refer to the *Command Reference* or *API Reference* manuals.

- CONNECT (1 | 2)  
Specifies whether CONNECT statements are to be processed as type 1 or type 2.
- SQLRULES (DB2 | STD)

Specifies whether type 2 CONNECTs should be processed according to the DB2 rules which allow CONNECT to switch to a dormant connection, or the SQL92 Standard (STD) rules which do not allow this.

- DISCONNECT (**EXPLICIT** | CONDITIONAL | AUTOMATIC)

Specifies what database connections are disconnected when a commit operation occurs. They are either:

- those which had been explicitly marked for release by the SQL RELEASE statement (EXPLICIT), or
- those that have no open WITH HOLD cursors as well as those marked for release (CONDITIONAL)<sup>9</sup>, or
- all connections (AUTOMATIC).

- SYNCPOINT (**ONEPHASE** | TWOPHASE | NONE)

Specifies how commits or rollbacks are to be coordinated among multiple database connections.

**ONEPHASE** Updates can only occur on one database in the unit of work, all other databases are read-only. Any update attempts to other databases raise an error (SQLSTATE 25000).

**TWOPHASE** A Transaction Manager (TM) will be used at run time to coordinate two phase commits among those databases that support this protocol.

**NONE** Does not use any TM to perform two phase commit and does not enforce single updater, multiple reader. When a COMMIT or ROLLBACK statement is executed, individual COMMITs or ROLLBACKs are posted to all databases. If one or more rollbacks fails an error (SQLSTATE 58005) is raised. If one or more commits fails an error (SQLSTATE 40003) is raised.

Any of the above options can be overridden at run time using a special SET CLIENT application programming interface (API). Their current settings can be obtained using the special QUERY CLIENT API. Note that these are not SQL statements; they are APIs defined in the various host languages and in the Command Line Processor. These are defined in the *Command Reference* and *API Reference* manuals.

## Data Representation Considerations

Different systems represent data in different ways. When data is moved from one system to another, data conversion sometimes must be performed. Products supporting DRDA will automatically perform any necessary conversions at the receiving system. With numeric data, the information needed to perform the conversion is the data type of the data and how that data type is represented by the sending system. With character data, additional information is needed to convert character strings. String conversion

---

<sup>9</sup> The CONDITIONAL option will not work properly with downlevel servers prior to Version 2. A disconnection will occur in these cases regardless of the presence of WITH HOLD cursors



depends on both the coded character set of the data and the operation that is to be performed with that data. Character conversions are performed in accordance with the IBM Character Data Representation Architecture (CDRA). For more information on character conversion, refer to *Character Data Representation Architecture Reference* SC09-1390.

---

## Character Conversion

A *string* is a sequence of bytes that may represent characters. Within a string, all the characters are represented by a common coding representation. In some cases, it might be necessary to convert these characters to a different coding representation. The process of conversion is known as *character conversion*.<sup>10</sup>

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 application server
- The values of result columns sent from the application 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.

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

<b>character set</b>	A defined set of characters. For example, the following character set appears in several code pages: <ul style="list-style-type: none"><li>• 26 non-accented letters A through Z</li><li>• 26 non-accented letters a through z</li><li>• digits 0 through 9</li><li>• . , ; ? ( ) ' " / - _ &amp; + % * = &lt; &gt;</li></ul>
<b>code page</b>	A set of assignments of characters to code points. In the ASCII encoding scheme for code page 850, for example, 'A' is assigned code point X'41' and 'B' is assigned code point X'42'. Within a code page, each code point has only one specific meaning. A code page is an attribute of the database. When an application program connects to the data-

---

<sup>10</sup> Character conversion, when required, is automatic and is transparent to the application when it is successful. A knowledge of conversion is therefore unnecessary when all the strings involved in a statement's execution are represented in the same way. This is frequently the case for *stand-alone* installations and for networks within the same country. Thus, for many readers, character conversion may be irrelevant.

base, the database manager determines the code page of the application.

**code point**

A unique bit pattern that represents a character.

**encoding scheme**

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

- Single-Byte ASCII
- Single-Byte EBCDIC
- Double-Byte ASCII
- Mixed Single- and Double-Byte ASCII.

### 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.

code page: pp1 (ASCII)

code page: pp2 (EBCDIC)

	0	1	2	3	4	5		E	F
0				0	@	P		Â	
1				1	A	Q		À	α
2			"	2	B	R		Å	β
3				3	C	S		Á	γ
4				4	D	T		Ã	δ
5			%	5	E	U		Ä	ε
E			.	>	N			5/8	Ö
F			/	*	0			®	

	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	{	

code point: 2F

character set ss1  
(in code page pp1)

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 and bit data. *Mixed* data is a mixture of single-byte, double-byte, or multi-byte characters. *Bit* data (columns defined as FOR BIT DATA or BLOBs, or binary strings) is not associated with any character set.

## Code Page Attributes

The database manager determines code page attributes for all character strings when an application is bound to a database. The potential code page attributes are:

- |                                  |                                                                                                                                                                                                                                                                           |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>The Database Code Page</b>    | The database code page stored in the database configuration files. This code page value is determined when the database is created and cannot be altered.                                                                                                                 |
| <b>The Application Code Page</b> | The code page under which the application is executed. Note that this is not necessarily the same code page under which the application was bound. (See the <i>Embedded SQL Programming Guide</i> for further information on binding and executing application programs.) |
| <b>Code Page 0</b>               | This represents a string that is derived from an expression that contains a FOR BIT DATA or BLOB value.                                                                                                                                                                   |

## String Code Page Attributes

Character string code page attributes are as follows:

- Columns may be in the database code page or code page 0 (if defined as character FOR BIT DATA or BLOB).
- Constants and special registers (for example, USER, CURRENT SERVER) are in the database code page. Note that constants are converted to the database code page when an SQL statement is bound to the database.
- Input host variables are in the application code page.

A set of rules is used to determine the code page attributes for operations that combine string objects, such as the results of scalar operations, concatenation, or set operations. At execution time, code page attributes are used to determine any requirements for code page conversions of strings.

For more details on character conversion, see:

- “Conversion Rules for String Assignments” on page 73 for rules on string assignments
- “Rules for String Conversions” on page 85 for rules on conversions when comparing or combining character strings.

---

## Authorization and Privileges

An *authorization* allows a user or group to perform a general task such as connecting to a database, creating tables, or administering a system. A *privilege* gives a user or group the right to access one specific database object in a specified way.

The database manager requires that a user be specifically authorized, either implicitly or explicitly,<sup>11</sup> to use each database function needed by that user to perform a specific task. Thus to create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table; and so on.

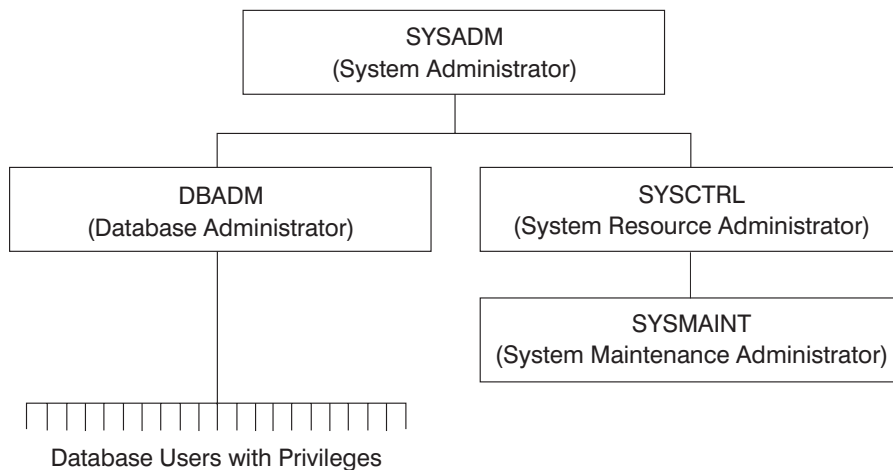


Figure 7. Hierarchy of Authorities and Privileges

The person or persons with administrative authority have the task of controlling the database manager and are responsible for the safety and integrity of the data. They control who will have access to the database manager and to what extent each user has access.

The database manager provides two administrative authorities:

**SYSADM** System administrator authority  
**DBADM** Database administrator authority

and two system control authorities:

**SYSCTRL** System control authority  
**SYSMANT** System maintenance authority

SYSADM authority is the highest level of authority and has control over all the resources created and maintained by the database manager. SYSADM authority

---

<sup>11</sup> Explicit authorities or privileges are granted to the user (GRANTEETYPE of U). Implicit authorities or privileges are granted to a group to which the user belongs (GRANTEETYPE of G).

includes all the privileges of DBADM, SYSCTRL, and SYSMANT, and the authority to grant or revoke DBADM authorities.

DBADM authority is the administrative authority specific to a single database. This authority includes privileges to create objects, issue database commands, and access the data in any of its tables through SQL statements. DBADM authority also includes the authority to grant or revoke CONTROL and individual privileges.

SYSCTRL authority is the higher level of system control authority and applies only to operations affecting system resources. It does not allow direct access to data. This authority includes privileges to create, update, or drop a database; quiesce an instance or database; and drop or create a table space.

SYSMANT authority is the second level of system control authority. A user with SYSMANT authority can perform maintenance operations on all databases associated with an instance. It does not allow direct access to data. This authority includes privileges to update database configuration files, backup a database or table space, restore an existing database, and monitor a database.

Database authorities apply to those activities that an administrator has allowed a user to perform within the database that do not apply to a specific instance of a database object. For example, a user may be granted the authority to create packages but not create tables.

Privileges apply to those activities that an administrator or object owner has allowed a user to perform on database objects. Users with privileges can create objects, though they face some constraints, unlike a user with an authority like SYSADM or DBADM. For example, a user may have the privilege to create a view on a table but not a trigger on the same table. Users with privileges have access to the objects they own, and can pass on privileges on their own objects to other users by using the GRANT statement.

CONTROL privilege allows the user to access a specific database object as desired and to GRANT and REVOKE privileges to and from other users on that object. DBADM authority is required to grant CONTROL privilege.

Individual privileges and database authorities allow a specific function but do not include the right to grant the same privileges or authorities to other users. The right to grant table, view or schema privileges to others can be extended to other users using the WITH GRANT OPTION on the GRANT statement.

---

## Storage Structures

Storage structures contain the objects of the database. The basic storage structures managed by the database manager are table spaces. A *table space* is a storage structure containing tables, indexes, large objects, and data defined with a LONG data type. There are two types of table spaces:

### **Database Managed Space (DMS) Table Space**

A table space which has its space managed by the database manager.

## System Managed Space (SMS) Table Space

A table space which has its space managed by the operating system.

All table spaces consist of containers. A *container* describes where objects, such as some tables, are stored. For example, a subdirectory in a file system could be a container.

For more information on table spaces and containers, see “CREATE TABLESPACE” on page 559 or the *Administration Guide*.

Data that is read from table space containers is placed in an area of memory called a *buffer pool*. A buffer pool is associated with a table space allowing control over which data shares the same memory areas for data buffering. For more information on buffer pools, see “CREATE BUFFERPOOL” on page 449 or the *Administration Guide*.

A partitioned database allows data to be spread across different database partitions. The partitions included are determined by the *nodegroup* assigned to the table space. A nodegroup is a group of one or more partitions that are defined as part of the database. A table space includes one or more containers for each partition in the nodegroup. A *partitioning map* is associated with each nodegroup. The partitioning map is used by the database manager to determine which partition from the nodegroup will store a given row of data. For more information on nodegroups and data partitioning, see “Data Partitioning Across Multiple Partitions” on page 41, “CREATE NODEGROUP” on page 508 or the *Administration Guide*.

A table can also include columns that register links to data stored in external files. The mechanism for this is the DATALINK data type. A DATALINK value which is recorded in a regular table points to a file stored in an external file server.

The DB2 File Manager, which is installed on a fileserver, works in conjunction with DB2 to provide the following optional functionality:

- Referential integrity to insure that files currently linked to DB2 are not deleted or renamed.
- Security to insure that only those with suitable SQL privileges on the DATALINK column can read the files linked to that column.
- Coordinated backup and recovery of the file.

The DataLinker product comprises the following facilities:

### DataLinks File Manager

Registers all the files in a particular file server that are linked to DB2.

### DataLinks Filter

Filters file system commands to insure that registered files are not deleted or renamed. Optionally also filters commands to insure that proper access authority exists.

---

### Data Partitioning Across Multiple Partitions

DB2 allows great flexibility in spreading data across multiple partitions (nodes) of a partitioned database. Users can choose how to partition their data by declaring partitioning keys and can determine which and how many partitions their table data can be spread across by selecting the nodegroup and table space in which the data should be stored. In addition, a partitioning map (which can be user-updatable) specifies the mapping of partitioning key values to partitions. This makes it possible for flexible workload parallelization across a partitioned database for large tables, while allowing smaller tables to be stored on one or a small number of partitions if the application designer chooses. Each local partition may have local indexes on the data it stores in order to provide high performance local data access.

A partitioned database supports a partitioned storage model, in which the partitioning key is used to partition table data across a set of database partitions. Index data is also partitioned with its corresponding tables, and stored locally at each partition.

Before partitions can be used to store database data, they must be defined to the database manager. Partitions are defined in a file called `db2nodes.cfg`. See the *Administration Guide* for more details about defining partitions.

The partitioning key for a table in a table space on a partitioned nodegroup is specified in the CREATE TABLE statement (or ALTER TABLE statement). If not specified, a partitioning key for a table is created by default from the first column of the primary key. If no primary key is specified, the default partitioning key is the first column defined in that table that has a data type other than a LONG or LOB data type. Partitioned tables must have at least one column that is neither a LONG nor a LOB data type. A table in a table space on a single-partition nodegroup will only have a partitioning key if it is explicitly specified.

*Hash partitioning* is used to place a row on a partition as follows.

1. A hashing algorithm (partitioning function) is applied to the partitioning key (all the columns), which results in a partitioning map index being generated.
2. This partitioning map index is used as an index into the partitioning map. The partition number at that index in the partitioning map is the partition where the row is stored.
3. Partitioning maps are associated with nodegroups, and tables are created in table spaces which are on nodegroups.

DB2 supports *partial declustering*, which means that the table can be partitioned across a subset of partitions in the system (that is, a nodegroup). Tables do not have to be partitioned across all the partitions in the system.

### Partitioning Maps

Each nodegroup is associated with a *partitioning map*, which is an array of 4 096 partition numbers. The partitioning map index produced by the partitioning function for each

## Data Partitioning Across Multiple Partitions

row of a table is used as an index into the partitioning map to determine partition on which a row is stored.

Figure 8 shows how the row with the partitioning key value (c1, c2, c3) is mapped to partitioning map index 2, which, in turn, references partition p5.

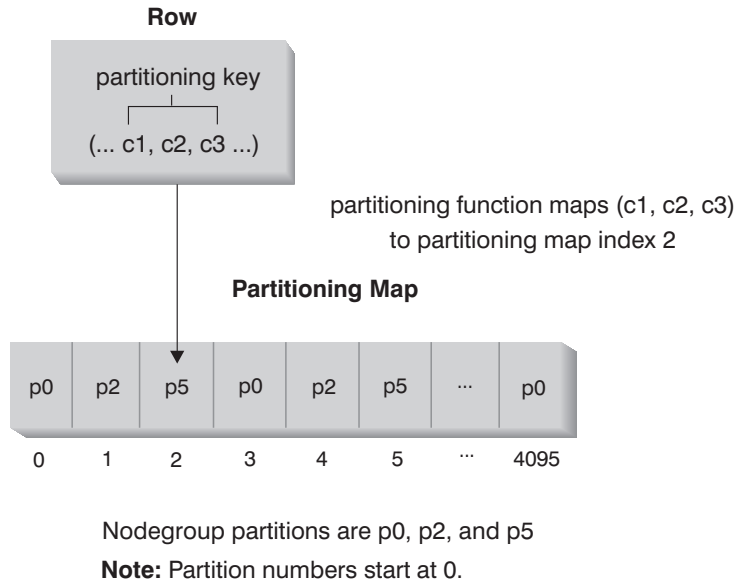


Figure 8. Data Distribution

The partitioning map can be changed, allowing the data distribution to be changed without modifying the partitioning key or the actual data. The new partitioning map is specified as part of the REDISTRIBUTE NODEGROUP command or API which uses it to redistribute the tables in the nodegroup. See *Command Reference* or *API Reference* for further information.

## Table Collocation

DB2 has the capability of recognizing when the data accessed for a join or subquery is located at the same partition in the same nodegroup. When this happens DB2 can choose to perform the join or subquery processing at the partition where the data is stored, which often has significant performance advantages. This situation is called table collocation. To be considered collocated tables, the tables must:

- be in the same nodegroup (that is not being redistributed <sup>12</sup>)
- have partitioning keys with the same number of columns

---

<sup>12</sup> While redistributing a nodegroup, tables in the nodegroup may be using different partitioning maps - they are not collocated.



## Data Partitioning Across Multiple Partitions

- have the corresponding columns of the partitioning key be partition compatible (see “Partition Compatibility” on page 87).

OR

- be in a single partition nodegroup defined on the same partition.

Rows in collocated tables with the same partitioning key values will be located on the same partition.

## Data Partitioning Across Multiple Partitions

---

## Chapter 3. Language Elements

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

Subject	Page
Characters	45
Tokens	46
Identifiers	47
Naming Conventions and Implicit Object Name Qualifications	48
Aliases	51
Authorization IDs and authorization-names	51
Data Types	54
Promotion of Data Types	66
Casting Between Data Types	67
Assignments and Comparisons	70
Rules for Result Data Types	82
Constants	88
Special Registers	91
Column Names	98
References to Host Variables	105
Functions	110
Expressions	117
Predicates	135
Search Conditions	153

---

### Characters

The basic symbols of keywords and operators in the SQL language are single-byte characters that are part of all IBM character sets. Characters of the language are classified as letters, digits, or special characters.

A *letter* is any of the 26 uppercase (A through Z) and 26 lowercase (a through z) letters plus the three characters (\$, #, and @), which are included for compatibility with host database products (for example, in code page 850, \$ is at X'24' # is at X'23', and @ is at X'40'). Letters also include the alphabets from the extended character sets. Extended character sets contain additional alphabetic characters; for example, those with diacritical (eg.,  $\tilde$ ) marks. The available characters depend on the code page in use.

A *digit* is any of the characters 0 through 9.

## Tokens

A *special character* is any of the characters listed below:

	blank	-	minus sign
"	quotation mark or double-quote	.	period
%	percent	/	slash
&	ampersand	:	colon
'	apostrophe or single quote	;	semicolon
(	left parenthesis	<	less than
)	right parenthesis	=	equals
*	asterisk	>	greater than
+	plus sign	?	question mark
,	comma	_	underline or underscore
	vertical bar	^	caret
!	exclamation mark		

## MBCS Considerations

All multi-byte characters are treated as letters, except for the double-byte blank which is a special character.

---

## Tokens

The basic syntactical units of the language are called *tokens*. A token is a sequence of one or more characters. A token cannot contain blank characters, unless it is a string constant or delimited identifier, which may contain blanks. (These terms are defined later.)

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 under "PREPARE" on page 673.

*Examples*

,            'string'            "fld1"            =            .

**Spaces:** A space is a sequence of one or more blank characters. Tokens other than string constants and delimited identifiers must not include a space. Any token may be followed by a space. Every ordinary token must be followed by a space or a delimiter token if allowed by the syntax.

**Comments:** Static SQL statements may include host language comments or SQL comments. Either type of comment may be specified wherever a space may be specified, except within a delimiter token or between the keywords EXEC and SQL. SQL comments are introduced by two consecutive hyphens (--) and ended by the end of the line. For more information, see “SQL Comments” on page 374.

**Uppercase and Lowercase:** Any token may include lowercase letters, but a lowercase letter in an ordinary token is folded to uppercase, except for host variables in the C language, which has case-sensitive identifiers. Delimiter tokens are never folded to uppercase. Thus, the statement:

```
select * from EMPLOYEE where lastname = 'Smith';
```

is equivalent, after folding, to:

```
SELECT * FROM EMPLOYEE WHERE LASTNAME = 'Smith';
```

### MBCS Considerations

Multi-byte alphabetic letters are not folded to uppercase. Single-byte characters, a to z, are folded to uppercase.

---

## Identifiers

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

### SQL Identifiers

There are two types of SQL identifiers: *ordinary identifiers* and *delimited identifiers*.

- An *ordinary identifier* is a letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier should not be identical to a reserved word (see Appendix G, “Reserved Schema Names and Reserved Words” on page 881 for information on reserved words).
- A *delimited identifier* is a sequence of one or more characters enclosed within quotation marks ("). Two consecutive quotation marks are used to represent one quotation mark within the delimited identifier. In this way an identifier can include lowercase letters.

*Example:*

```
WKLYSAL    WKLY_SAL    "WKLY_SAL"    "WKLY SAL"    "UNION"    "wkly_sal"
```

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 quotation marks surrounding the delimited identifier.

Character conversions between identifiers created on a double-byte code page but used by an application or database on a multi-byte code page may require special consideration. After conversion to multi-byte, it is possible that such identifiers may exceed

## Naming Conventions

the length limit for an identifier (see Appendix O, “Japanese and Traditional-Chinese EUC Considerations” on page 973 for details).

### 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. A host identifier should not be greater than 30 characters and should not begin with 'SQL'.

---

## Naming Conventions and Implicit Object Name Qualifications

The rules for forming a name depend on the type of the object designated by the name. Database object names may be made up of a single identifier or they may be schema qualified objects made up of two identifiers. Schema qualified object names may be specified without the schema name. In such cases, a schema name is implicit.

In dynamic SQL statements, a schema qualified object name implicitly uses the CURRENT SCHEMA special register value as the qualifier for unqualified object name references. By default it is set to the current authorization ID. See “SET SCHEMA” on page 733 for details.

In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified database object names. By default it is set to authorization ID of the binder. See the *Command Reference* for details.

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

**alias-name** A schema qualified name that designates an alias. The unqualified form of an alias-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier.

**attribute-name** A long identifier that designates an attribute of a structured data type.

**authorization-name** A short identifier that designates a user or group. Note the following restrictions on the characters that can be used:

- The underscore character ( \_ ) is not valid.
- The name must not begin with the characters 'SYS', 'IBM', or 'SQL'.
- The name must not be: ADMINS, GUESTS, LOCAL, PUBLIC, or USERS.
- A delimited authorization ID must not contain lowercase letters.
- Letters from the extended character set are not allowed.

**bufferpool-name** A long identifier that designates a bufferpool.

**column-name** A qualified or unqualified name that designates a column of a table or view. The unqualified form of a column-name is a long identifier. The qualified form is a qualifier followed by a period

## Naming Conventions

	and a long identifier. The qualifier is a table-name, a view-name, or a correlation-name.
<b>constraint-name</b>	A long identifier that designates a referential constraint, primary key constraint, unique constraint or a table check constraint.
<b>correlation-name</b>	A long identifier that designates a table or a view.
<b>cursor-name</b>	A long identifier that designates an SQL cursor.
<b>descriptor-name</b>	A colon followed by a host identifier that designates an SQL descriptor area (SQLDA). See “References to Host Variables” on page 105 for a description of a host identifier. Note that a descriptor-name never includes an indicator variable.
<b>distinct-type-name</b>	A qualified or unqualified name that designates a distinct type-name. The unqualified form of a distinct-type-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier. An unqualified distinct-type-name in an SQL statement is implicitly qualified by the database manager, depending on context.
<b>event-monitor-name</b>	A long identifier that designates an event monitor.
<b>function-name</b>	A qualified or unqualified name that designates a function. The unqualified form of a function-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier. An unqualified function-name in an SQL statement is implicitly qualified by the database manager, depending on context.
<b>host-variable</b>	A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, as explained in “References to Host Variables” on page 105.
<b>index-name</b>	A schema qualified name that designates an index. The unqualified form of an index-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier.
<b>nodegroup-name</b>	A long identifier that designates a nodegroup.
<b>package-name</b>	A schema qualified name that designates a package. The unqualified form of a package-name is a short identifier. The qualified form is a short identifier followed by a period and a short identifier.
<b>procedure-name</b>	A qualified or unqualified name that designates a procedure. The unqualified form of a procedure-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier. An unqualified procedure-name in an SQL statement is implicitly qualified by the database manager, depending on context.

## Naming Conventions

<b>schema-name</b>	<p>A short identifier that provides a logical grouping for SQL objects. A schema-name used as a qualifier of the name of an object may be implicitly determined:</p> <ul style="list-style-type: none"><li>• from the value of the CURRENT SCHEMA special register</li><li>• from the value of the QUALIFIER precompile/bind option</li><li>• based on a resolution algorithm that uses the CURRENT PATH special register</li><li>• based on the schema name of another object in the same SQL statement.</li></ul>
<b>server-name</b>	<p>A long identifier that designates an application server.</p>
<b>specific-name</b>	<p>A qualified or unqualified name that designates specific-name. The qualified form is a schema-name followed by a period and a long identifier. A specific-name can be used to source a function, to drop and to comment on a procedure or a function. It can never be used to invoke a function or procedure. An unqualified specific-name in an SQL statement is implicitly qualified by the database manager, depending on context.</p>
<b>statement-name</b>	<p>A long identifier that designates a prepared SQL statement.</p>
<b>supertype-name</b>	<p>A qualified or unqualified name that designates a the supertype of a type-name. The unqualified form of a supertype-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier. An unqualified supertype-name in an SQL statement is implicitly qualified by the database manager, depending on context.</p>
<b>table-name</b>	<p>A schema qualified name that designates a table. The unqualified form of a table-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier.</p>
<b>tablespace-name</b>	<p>A long identifier that designates a table space.</p>
<b>trigger-name</b>	<p>A schema qualified name that designates a trigger. The unqualified form of a trigger-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier.</p>
<b>type-name</b>	<p>A qualified or unqualified name that designates a type-name. The unqualified form of a type-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier. An unqualified type-name in an SQL statement is implicitly qualified by the database manager, depending on context.</p>
<b>typed-table-name</b>	<p>A schema qualified name that designates a typed table. The unqualified form of a typed-table-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier.</p>



## Authorization IDs and authorization-names

<b>typed-view-name</b>	A schema qualified name that designates a typed view. The unqualified form of a typed-view-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier.
<b>view-name</b>	A schema qualified name that designates a view. The unqualified form of a view-name is a long identifier. The qualified form is a short identifier followed by a period and a long identifier.

---

### Aliases

A table alias can be thought of as an alternative name for a table or view. A table or view, therefore, can be referred to in an SQL statement by its name or by a table alias.

An alias can be used wherever a table or view name can be used. An alias can be created even though the object does not exist (though it must exist by the time a statement referring to it is compiled). It can refer to another alias if no circular or repetitive references are made along the chain of aliases. An alias can only refer to a table, view, or alias within the same database. An alias name cannot be used where a new table or view name is expected, such as in the CREATE TABLE or CREATE VIEW statements; for example, if an alias name of PERSONNEL is created then a subsequent statement such as CREATE TABLE PERSONNEL... will cause an error.

The option of referring to a table or view by an alias is not explicitly shown in the syntax diagrams or mentioned in the description of the SQL statement.

A new unqualified alias cannot have the same fully-qualified name as an existing table, view, or alias.

The effect of using an alias in an SQL statement is similar to that of text substitution. The alias, which must be defined when the SQL statement is compiled, is replaced at statement compilation time by the qualified base table or view name. For example, if PBIRD.SALES is an alias for DSPN014.DIST4\_SALES\_148, then at compilation time:

```
SELECT * FROM PBIRD.SALES
```

effectively becomes

```
SELECT * FROM DSPN014.DIST4_SALES_148
```

For syntax toleration of existing DB2 for MVS/ESA applications, SYNONYM can be used in place of ALIAS in the CREATE ALIAS and DROP ALIAS statements.

---

### Authorization IDs and authorization-names

An *authorization ID* is a character string that is obtained by the database manager when a connection is established between the database manager and either an application process or a program preparation process. It designates a set of privileges. It may also designate a user or a group of users, but this property is not controlled by the database manager.

## Authorization IDs and authorization-names

Authorization IDs are used by the database manager to provide:

- Authorization checking of SQL statements
- Default value for the QUALIFIER precompile/bind option and the CURRENT SCHEMA special register. Also, the authorization ID is included in the default CURRENT PATH special register and FUNCPATH precompile/bind option.

An authorization ID applies to every SQL statement. The authorization ID that applies to a static SQL statement is the authorization ID that is used during program binding. The authorization ID that applies to a dynamic SQL statement is the authorization ID that was obtained by the database manager when a connection was established between the database manager and the process. This is called the *run-time authorization ID*.

An *authorization-name* specified in an SQL statement should not be confused with the authorization ID of the statement. An authorization-name is an identifier that is used within various SQL statement. An authorization-name is used in a CREATE SCHEMA statement to designate the owner of the schema. An authorization-name is used in GRANT and REVOKE statements to designate a target of the grant or revoke. Note that the premise of a grant of privileges to *X* is that *X* or a member of the group *X* will subsequently be the authorization ID of statements which require those privileges.

*Examples:*

- Assume SMITH is the userid and the authorization ID that the database manager obtained when the connection was established with the application process. The following statement is executed interactively:

```
GRANT SELECT ON TDEPT TO KEENE
```

SMITH is the authorization ID of the statement. Hence, in a dynamic SQL statement the default value of the CURRENT SCHEMA special register and in static SQL the default QUALIFIER precompile/bind option is SMITH. Thus, the authority to execute the statement is checked against SMITH and SMITH is the *table-name* implicit qualifier based on qualification rules described in “Naming Conventions and Implicit Object Name Qualifications” on page 48.

KEENE is an authorization-name specified in the statement. KEENE is given the SELECT privilege on SMITH.TDEPT.

- Assume SMITH has administrative authority and is the authorization ID of the following dynamic SQL statements with no SET SCHEMA statement issued during the session:

```
DROP TABLE TDEPT
```

Removes the SMITH.TDEPT table.

```
DROP TABLE SMITH.TDEPT
```

Removes the SMITH.TDEPT table.

```
DROP TABLE KEENE.TDEPT
```

Removes the KEENE.TDEPT table. Note that KEENE.TDEPT and SMITH.TDEPT are different tables.

## Authorization IDs and authorization-names

```
CREATE SCHEMA PAYROLL AUTHORIZATION KEENE
```

KEENE is the authorization-name specified in the statement which creates a schema called PAYROLL. KEENE is the owner of the schema PAYROLL and is given CREATEIN, ALTERIN, and DROPIN privileges with the ability to grant them to others.

# Data Types

---

## Data Types

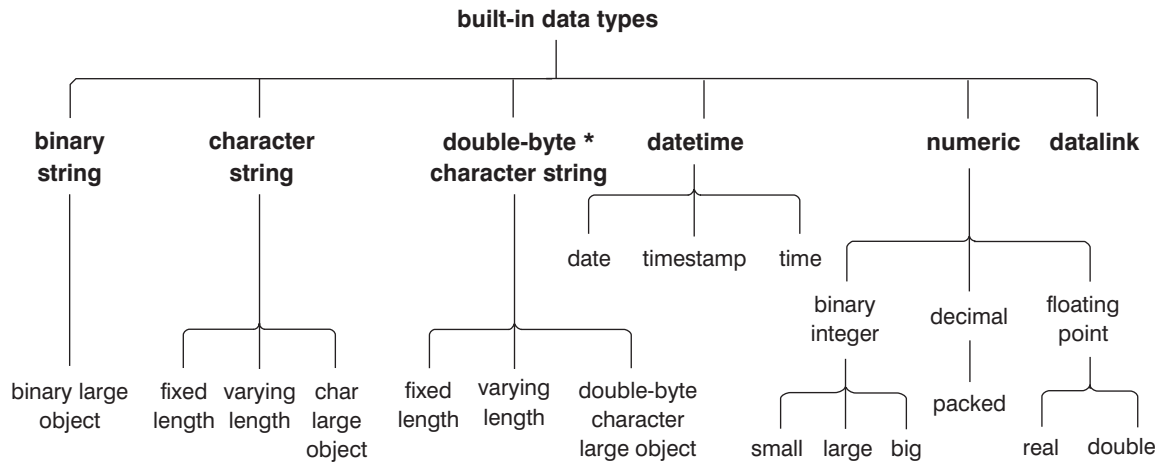
For information about specifying the data types of columns, see “CREATE TABLE” on page 522.

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
- Host variables
- Functions
- Expressions
- Special registers.

DB2 supports a number of built-in datatypes, which are described in this section. It also provides support for user-defined data types. See “User Defined Types” on page 64 for a description of user-defined data types.

Figure 9 illustrates the supported built-in data types.



\* Double-byte character is usually referred to as graphic character.

---

Figure 9. Supported Built-in Data Types

## Nulls

All data types include the null value. The null value is a special value that is distinct from all non-null values and thereby denotes the absence of a (non-null) value. Although all data types include the null value, columns defined as NOT NULL cannot contain null values.

### Large Objects (LOBs)

The term *large object* and the generic acronym *LOB* are used to refer to any BLOB, CLOB, or DBCLOB data type. LOB values are subject to the restrictions that apply to LONG VARCHAR values as specified in “Restrictions Using Varying-Length Character Strings” on page 57. For LOB strings, these restrictions apply even when the length attribute of the string is 254 bytes or less.

#### Character Large Object (CLOB) Strings

A *Character Large Object (CLOB)* is a varying-length string measured in bytes that can be up to 2 gigabytes (2 147 483 647 bytes) long. A *CLOB* is used to store large SBCS or mixed (SBCS and MBCS) character-based data such as documents written with a single character set (and, therefore, has an SBCS or mixed code page associated with it). Note that a CLOB is considered to be a character string.

#### Double-Byte Character Large Object (DBCLOB) Strings

A *Double-Byte Character Large Object (DBCLOB)* is a varying-length string of double-byte characters that can be up to 1 073 741 823 characters long. A *DBCLOB* is used to store large DBCS character based data such as documents written with a single character set (and, therefore has a DBCS CCSID associated with it). Note that a DBCLOB is considered to be a graphic string.

#### Binary Large Objects (BLOBs)

A *Binary Large Object (BLOB)* is a varying-length string measured in bytes that can be up to 2 gigabytes (2 147 483 647 bytes) long. A *BLOB* is primarily intended to hold non-traditional data such as pictures, voice, and mixed media. Another use is to hold structured data for exploitation by user-defined types and user-defined functions. As with FOR BIT DATA character strings, BLOB strings are not associated with a character set.

#### Manipulating Large Objects (LOBs) with Locators

Since LOB values can be very large, the transfer of these values from the database server to client application program host variables can be time consuming. However, it is also true that application programs typically process LOB values a piece at a time, rather than as a whole. For those cases where an application does not need (or want) the entire LOB value to be stored in application memory, the application can reference a LOB value via a large object locator (LOB locator).

A *large object locator* or LOB locator is a host variable with a value that represents a single LOB value in the database server. LOB locators were developed to provide users with a mechanism by which they could easily manipulate very large objects in application programs without requiring them to store the entire LOB value on the client machine where the application program may be running.

For example, when selecting a LOB value, an application program could select the entire LOB value and place it into an equally large host variable (which is acceptable if the application program is going to process the entire LOB value at once), or it could instead select the LOB value into a LOB locator. Then, using the LOB locator, the appli-

## Data Types

cation program can issue subsequent database operations on the LOB value (such as applying the scalar functions SUBSTR, CONCAT, VALUE, LENGTH, doing an assignment, searching the LOB with LIKE or POSSTR, or applying UDFs against the LOB) by supplying the locator value as input. The resulting output of the locator operation, for example the amount of data assigned to a client host variable, would then typically be a small subset of the input LOB value.

LOB locators may also represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

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

For normal host variables in an application program, when a null value is selected into that host variable, the indicator variable is set to -1, signifying that the value is null. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Since a locator host variable itself can never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value — the server does not track null values with valid locators.

It is important to understand that a LOB locator represents a value, not a row or location in the database. Once a value is selected into a locator, there is no operation that one can perform on the original row or table that will affect the value which is referenced by the locator. The value associated with a locator is valid until the transaction ends, or until the locator is explicitly freed, whichever comes first. Locators do not force extra copies of the data in order to provide this function. Instead, the locator mechanism stores a description of the base LOB value. The materialization of the LOB value (or expression, as shown above) is deferred until it is actually assigned to some location — either into a user buffer in the form of a host variable or into another record's field value in the database.

A LOB locator is only a mechanism used to refer to a LOB value during a transaction; it does not persist beyond the transaction in which it was created. Also, it is not a database type; it is never stored in the database and, as a result, cannot participate in views or check constraints. However, since a locator is a client representation of a LOB type, there are SQLTYPEs for LOB locators so that they can be described within an SQLDA structure that is used by FETCH, OPEN and EXECUTE statements.

## 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*. This value should not be confused with the null value.

### Fixed-Length Character Strings

All values of a fixed-length string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 254, inclusive.

### Varying-Length Character Strings

Varying-length character strings are of three types: VARCHAR, LONG VARCHAR, and CLOB.

- VARCHAR types are varying-length strings of up to 4000 bytes.
- LONG VARCHAR types are varying-length strings of up to 32,700 bytes.
- CLOB types are varying-length strings of up to 2 gigabytes.

**Restrictions Using Varying-Length Character Strings:** Special restrictions apply to an expression resulting in a varying-length string data type whose maximum length is greater than 254 bytes; such expressions are not permitted in:

- A SELECT DISTINCT statement's SELECT list
- A GROUP BY clause
- An ORDER BY clause
- A column function with DISTINCT
- A subselect of a set operator other than UNION ALL.

In addition to the restrictions listed above, expressions resulting in LONG VARCHAR or CLOB data types are not permitted in:

- A Basic, Quantified, BETWEEN, or IN predicate
- A column function
- VARGRAPHIC, TRANSLATE, and datetime scalar functions
- The pattern operand in a LIKE predicate or the search string operand in a POSSTR function
- The string representation of a datetime value

### NUL-Terminated Character Strings

NUL-terminated character strings found in C are handled differently, depending on the standards level of the precompile option. See the C language specific section in the *Application Programming Guide* for more information on the treatment of NUL-terminated character strings.

This data type cannot be created in a table. It can only be used to insert data into and retrieve data from the database.

### Character Subtypes

Each character string is further defined as one of:

<b>Bit data</b>	Data that is not associated with a coded character set.
<b>SBCS data</b>	Data in which every character is represented by a single byte.
<b>Mixed data</b>	Data that may contain a mixture of characters from a single-byte character set (SBCS) and a multi-byte character set (MBCS).

**SBCS and MBCS Considerations:** SBCS data is supported only in a SBCS database. Mixed data is only supported in an MBCS database.

## Data Types

### Graphic Strings

A *graphic string* is a sequence of bytes which represents double-byte character data. The length of the string is the number of double-byte characters in the sequence. If the length is zero, the value is called the empty string. This value should not be confused with the null value.

Graphic strings are not validated to ensure that their values contain only double-byte character code points.<sup>13</sup> Rather, the database manager assumes that double-byte character data is contained within graphic data fields. The database manager checks that a graphic string value is an even number of bytes in length.

A graphic string data type may be fixed length or varying length; the semantics of fixed length and varying length are analogous to those defined for character string data types.

#### Fixed-Length Graphic Strings

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

#### Varying-Length Graphic Strings

Varying-length graphic strings are of three types: VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB.

- VARGRAPHIC types are varying-length strings of up to 2000 double-byte characters.
- LONG VARGRAPHIC types are varying-length strings of up to 16,350 double-byte characters.
- DBCLOB types are varying-length strings of up to 1 073 741 823 double-byte characters.

Special restrictions apply to an expression resulting in a varying-length graphic string data type whose maximum length is greater than 127. Those restrictions are the same as specified in “Restrictions Using Varying-Length Character Strings” on page 57.

#### NUL-Terminated Graphic Strings

NUL-terminated graphic strings found in C are handled differently, depending on the standards level of the precompile option. See the C language specific section in the *Application Programming Guide* for more information on the treatment of NUL-terminated graphic strings.

This data type cannot be created in a table. It can only be used to insert data into and retrieve data from the database.

---

<sup>13</sup> The exception to this rule is an application precompiled with the WCHARTYPE CONVERT option. In this case, validation does occur. See “Programming in C and C++” in the *Application Programming Guide* for details.



## Binary String

A *binary string* is a sequence of bytes. Unlike a character string which usually contains text data, a binary string is used to hold non-traditional data such as pictures. Note that character strings of the 'bit data' subtype may be used for similar purposes, but the two data types are not compatible. The BLOB scalar function can be used to cast a character for bit string to a binary string. The length of a binary string is the number of bytes. It is not associated with a coded character set. Binary strings have the same restrictions as character strings (see “Restrictions Using Varying-Length Character Strings” on page 57 for details).

## Numbers

All numbers have a sign and a precision. The *precision* is the number of bits or digits excluding the sign. The sign is considered positive if the value of a number is zero.

### Small Integer (SMALLINT)

A *small integer* is a two byte integer with a precision of 5 digits. The range of small integers is -32768 to 32767.

### Large Integer (INTEGER)

A *large integer* is a four byte integer with a precision of 10 digits. The range of large integers is -2147483648 to +2147483647.

### Big Integer (BIGINT)

A *big integer* is an eight byte integer with a precision of 19 digits. The range of big integers is -9223372036854775808 to +9223372036854775807.

### Single-Precision Floating-Point (REAL)

A *single-precision floating-point* number is a 32 bit approximation of a real number. The number can be zero or can range from -3.402E+38 to -1.175E-37, or from 1.175E-37 to 3.402E+38.

### Double-Precision Floating-Point (DOUBLE or FLOAT)

A *double-precision floating-point* number is a 64 bit approximation of a real number. The number can be zero or can range from -1.79769E+308 to -2.225E-307, or from 2.225E-307 to 1.79769E+308.

### Decimal (DECIMAL or NUMERIC)

A *decimal* value is a packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits. For information on packed decimal representation, see “Packed Decimal Numbers” on page 770.

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 the absolute value of  $n$  is the largest number that can be represented with the applicable precision and scale. The maximum range is  $-10^{**31}+1$  to  $10^{**31}-1$ .

## Data Types

### Datetime Values

The datetime data types are described below. Although datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, they are neither strings nor numbers.

#### Date

A *date* is a three-part value (year, month, and day). 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 *x*, where *x* depends on the month.

The internal representation of a date is a string of 4 bytes. Each byte consists of 2 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 SQLDA, is 10 bytes, which is the appropriate length for a character string representation of the value.

#### Time

A *time* is a three-part value (hour, minute, and second) designating a time of day under a 24-hour clock. The range of the hour part is 0 to 24; while the range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications will be zero.

The internal representation of a time is a string of 3 bytes. Each byte is 2 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 SQLDA, is 8 bytes, which is the appropriate length for a character string representation of the value.

#### Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time as defined above, 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 2 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 SQLDA, is 26 bytes, which is the appropriate length for the character string representation of the value.

### 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 SQL user. Dates, times, and timestamps can, however, also be represented by character strings, and these representations directly concern the SQL user since there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Thus, to be retrieved, a datetime value must be assigned to a character string variable. Note that the CHAR function can be used to

change a datetime value to a string representation. The character string representation is normally the default format of datetime values associated with the country code of the database, unless overridden by specification of the *DATETIME* option when the program is precompiled or bound to the database.

No matter what its length, a large object string or LONG VARCHAR cannot be used as the string that represents a datetime value; otherwise an error is raised (SQLSTATE 42884).

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp before the operation is performed. The following sections define the valid string representations of datetime values.

### 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 may be included; leading zeros may be omitted from the month and day portions.

Valid string formats for dates are listed in Table 1. Each format is identified by name and includes an associated abbreviation and an example of its use.

Format Name	Abbreviation	Date Format	Example
International Standards Organization	ISO	yyyy-mm-dd	1991-10-27
IBM USA standard	USA	mm/dd/yyyy	10/27/1991
IBM European standard	EUR	dd.mm.yyyy	27.10.1991
Japanese Industrial Standard Christian era	JIS	yyyy-mm-dd	1991-10-27
Site-defined (see <i>DataLinker Quick Beginnings</i> )	LOC	Depends on database country code	—

### 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 may be included; a leading zero may be omitted from the hour part of the time and seconds may be omitted entirely. If seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13.30 is equivalent to 13.30.00.

Valid string formats for times are listed in Table 2 on page 62. Each format is identified by name and includes an associated abbreviation and an example of its use.

## Data Types

Format Name	Abbreviation	Time Format	Example
International Standards Organization <sup>2</sup>	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
Site-defined (see <i>DataLinker Quick Beginnings</i> )	LOC	Depends on database country code	—

### Notes:

1. In ISO, EUR and JIS format, .ss (or :ss) is optional.
2. The International Standards Organization recently changed the time format so that it is identical with the Japanese Industrial Standard Christian Era. Therefore, use JIS format if an application requires the current International Standards Organization format.
3. In the case of the USA time string format, the minutes specification may be omitted, indicating an implicit specification of 00 minutes. Thus 1 PM is equivalent to 1:00 PM.
4. In the USA time format, the hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM. There is a single space before the AM and PM. 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 corresponds to 00.01.00 through 00.59.00.
  - 01:00 AM through 11:59 AM corresponds to 01.00.00 through 11.59.00.
  - 12:00 PM (noon) through 11:59 PM corresponds to 12.00.00 through 23.59.00.
  - 12:00 AM (midnight) corresponds to 24.00.00 and 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 may be included. Leading zeros may be omitted from the month, day, and hour part of the timestamp, and microseconds may be truncated or entirely omitted. If any trailing zero digits are omitted in the microseconds portion, an implicit specification of 0 is assumed for the missing digits. Thus, 1991-3-2-8.30.00 is equivalent to 1991-03-02-08.30.00.000000.

SQL statements also support the ODBC string representation of a timestamp as an input value only. The ODBC string representation of a timestamp has the form *yyyy-*

*mm-dd hh:mm:ss.nnnnnn*. See the *CLI Guide and Reference* for more information on ODBC.

### MBCS Considerations

Date, time and timestamp strings must contain only single-byte characters and digits.

### DATALINK Values

A DATALINK value is an encapsulated value that contains a logical reference from the database to a file stored outside the database. The attributes of this encapsulated value are as follows:

#### link type

The currently supported type of link is a URL (Uniform Resource Locator).

#### scheme

For URLs this is a value such as HTTP or FILE. The value, no matter what case it is entered in, is stored in the database in upper case. If a value is not specified, FILE is included in the DATALINK value.

#### file server name

The complete address of the file server. The value, no matter what case it is entered in, is stored in the database in upper case. If a value is not specified, the file server name of the database server is selected as default and included in the DATALINK value.

#### file path

The identity of the file within the server. The value is case sensitive and therefore it is not converted to upper case when stored in the database.

#### access control token

When appropriate, the access token is embedded within the file path. It is generated dynamically and is not a permanent part of the DATALINK value that is stored in the database.

#### comment

Up to 254 bytes of descriptive information. This is intended for application specific uses such as further or alternative identification of the location of the data.

The characters used in a DATALINK value are limited to the set defined for a URL. These characters include the uppercase (A through Z) and lower case (a through z) letters, the digits (0 through 9) and a subset of special characters (\$, -, \_, @, ., &, +, !, \*, ", ', (, ), =, :, /, #, ?, ;, space, and comma).

The first four attributes are collectively known as the *linkage attributes*. It is possible for a DATALINK value to have only a comment attribute and no linkage attributes. Such a value may even be stored in a column but, of course, no file will be linked to such a column.

It should be noted that DATALINKs cannot be exchanged with a DRDA server.

## Data Types

It is important to distinguish between these DATALINK references to files and the LOB file reference variables described in the section entitled "References to BLOB, CLOB, and DBCLOB File Reference Variables". The similarity is that they both contain a representation of a file. However:

- DATALINKs are retained in the database and both the links and the data in the linked files can be considered as a natural extension of data in the database.
- File reference variables exist temporarily on the client and they can be considered as an alternative to a host program buffer.

Built-in scalar functions are provided to build a DATALINK value (DLVALUE) and to extract the encapsulated values from a DATALINK value (DLCOMMENT, DLLINKTYPE, DLURLCOMPLETE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLSERVER).

## User Defined Types

### Distinct Types

A *distinct type* is a user-defined data type that shares its internal representation with an existing type (its "source" type), but is considered to be a separate and incompatible type for most operations. For example, one might want to define a picture type, a text type, and an audio type, all of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

The following example illustrates the creation of 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 considered to be a separate type that is not comparable to a BLOB or to any other type. This allows the creation of functions written specifically for AUDIO and assures that these functions will not be applied to any other type (pictures, text, etc.).

Distinct types are identified by qualified identifiers. If the schema name is not used to qualify the distinct type name when used in other than the CREATE DISTINCT TYPE, DROP DISTINCT TYPE, or COMMENT ON DISTINCT TYPE statements, the *SQL path* is searched in sequence for the first schema with a distinct type that matches. The SQL path is described in "CURRENT PATH" on page 94.

Distinct types support strong typing by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. For this reason, a distinct type does not automatically acquire the functions and operators of its source type, since these may not be meaningful. (For example, the LENGTH function of the AUDIO type might return the length of its object in seconds rather than in bytes.)

Distinct types sourced on LONG VARCHAR, LONG VARGRAPHIC, LOB types, or DATALINK are subject to the same restrictions as their source type.

However, certain functions and operators of the source type can be explicitly specified to apply to the distinct type by defining user-defined functions that are sourced on func-

tions defined on the source type of the distinct type (see “User-defined Type Comparisons” on page 81 for examples). The comparison operators are automatically generated for user-defined distinct types, except those using LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, or DATALINK as the source type. In addition, functions are generated to support casting from the source type to the distinct type and from the distinct type to the source type.

### Structured Types

A *structured type* is a user-defined data type that has a structure that is defined in the database. It contains a sequence of named *attributes*, each of which has a data type. A structured type may be defined as a *subtype* of another structured type, called its *supertype*. A subtype inherits all the attributes of its supertype and may have additional attributes defined. The set of structured types that are related to a common supertype is called a *type hierarchy* and the supertype that does not have any supertype is called the *root type* of the type hierarchy.

A structured type may be used as the type of a table or a view. The names and data types of the attributes of the structured type become the names and data types of the columns of this *typed table* or *typed view*. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type.

A structured type cannot be used as the data type of a column of a table or a view. There is also no support for retrieving a structured type into a host variable in an application program.

### Reference (REF) Types

A *reference type* is a companion type to a structured type. Similar to a distinct type, a reference type is a scalar type that shares a common representation with one of the built-in data types. This same representation is shared for all types in the type hierarchy. The reference type representation is defined when the root type of a type hierarchy is created. When using a reference type, a structured type is specified as a parameter of the type. This parameter is called the *target type* of the reference.

The target of a reference is always a row in a typed table or view. When a reference type is used, it may have a *scope* defined. The scope identifies a table (called the *target table*) or view (called the *target view*) that contains the target row of a reference value. The target table or view must have the same type as the target type of the reference type. An instance of a scoped reference type uniquely identifies a row in a typed table or typed view, called the *target row*.

## Promotion of Data Types

---

### Promotion of Data Types

Data types can be classified into groups of related data types. Within such groups, a precedence order exists where one data type is considered to precede another data type. This precedence is used to allow the *promotion* of one data type to a data type later in the precedence ordering. For example, the data type CHAR can be promoted to VARCHAR; INTEGER can be promoted to DOUBLE PRECISION; but CLOB is NOT promotable to VARCHAR.

Promotion of data types is used when:

- performing function resolution (see “Function Resolution” on page 112)
- casting user-defined types (see “Casting Between Data Types” on page 67)
- assigning user-defined types to built-in data types (see “User-defined Type Assignments” on page 77).

Table 3 shows the precedence list (in order) for each data type and can be used to determine the data types to which a given data type can be promoted. The table shows that the best choice is always the same data type instead of choosing to promote to another data type.

---

*Table 3 (Page 1 of 2). Data Type Precedence Table*

<b>Data Type</b>	<b>Data Type Precedence List (in best-to-worst order)</b>
CHAR	CHAR, VARCHAR, LONG VARCHAR, CLOB
VARCHAR	VARCHAR, LONG VARCHAR, CLOB
LONG VARCHAR	LONG VARCHAR, CLOB
GRAPHIC	GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB
VARGRAPHIC	VARGRAPHIC, LONG VARGRAPHIC, DBCLOB
LONG VARGRAPHIC	LONG VARGRAPHIC, DBCLOB
BLOB	BLOB
CLOB	CLOB
DBCLOB	DBCLOB
SMALLINT	SMALLINT, INTEGER, BIGINT, decimal, real, double
INTEGER	INTEGER, BIGINT, decimal, real, double
BIGINT	BIGINT, decimal, real, double
decimal	decimal, real, double
real	real, double
double	double
DATE	DATE
TIME	TIME

---



## Casting Between Data Types

Table 3 (Page 2 of 2). Data Type Precedence Table

Data Type	Data Type Precedence List (in best-to-worst order)
TIMESTAMP	TIMESTAMP
DATALINK	DATALINK
udt	udt (same name)
REF(T)	REF(S) (provided that S is a supertype of T)

**Note:**

The lower case types above are defined as follows:

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

udt = a user-defined type

Shorter and longer form synonyms of the data types listed are considered to be the same as the synonym listed.

## Casting Between Data Types

There are many occasions where a value with a given data type needs to be *cast* to a different data type or to the same data type with a different length, precision or scale. Data type promotion (as defined in “Promotion of Data Types” on page 66) is one example where the promotion of one data type to another data type requires that the value is cast to the new data type. A data type that can be cast to another data type is *castable* from the source data type to the target data type.

Casting between data types can be done explicitly using the CAST specification (see “CAST Specifications” on page 131) but may also occur implicitly during assignments involving a user-defined types (see “User-defined Type Assignments” on page 77). Also, when creating sourced user-defined functions (see “CREATE FUNCTION” on page 467), the data types of the parameters of the source function must be castable to the data types of the function that is being created.

The supported casts between built-in data types are shown in Table 4 on page 69.

The following casts involving distinct types are supported:

- cast from distinct type *DT* to its source data type *S*
- cast from the source data type *S* of distinct type *DT* to distinct type *DT*
- cast from distinct type *DT* to the same distinct type *DT*
- cast from a data type *A* to distinct type *DT* where *A* is promotable to the source data type *S* of distinct type *DT* (see “Promotion of Data Types” on page 66)
- cast from an INTEGER to distinct type *DT* with a source data type SMALLINT
- cast from a DOUBLE to distinct type *DT* with a source data type REAL

## Casting Between Data Types

- cast from a VARCHAR to distinct type *DT* with a source data type CHAR
- cast from a VARGRAPHIC to distinct type *DT* with a source data type GRAPHIC.

When a user-defined data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name. The SQL path is described further in “CURRENT PATH” on page 94.

The following casts involving reference types are supported:

- cast from reference type *RT* to its representation data type *S*
- cast from the representation data type *S* of reference type *RT* to reference type *RT*
- cast from reference type *RT* with target type *T* to a reference type *RS* with target type *S* where *S* is a supertype of *T*.
- cast from a data type *A* to reference type *RT* where *A* is promotable to the representation data type *S* of reference type *RT* (see “Promotion of Data Types” on page 66).

When the target type of a reference data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name. The SQL path is described further in “CURRENT PATH” on page 94.

## Casting Between Data Types

Table 4. Supported Casts between Built-in Data Types

Target Data Type →	SMALLINT		INTEGER		BIGINT		DECIMAL		REAL		DOUBLE		CHAR		VARCHAR		LONG VARCHAR		CLOB		GRAPHIC		VARGRAPHIC		LONG VARG		DBCLOB		DATE		TIME		TIMESTAMP		BLOB				
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
DECIMAL	Y	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	Y	Y	-	-	-	-	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
VARCHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y	-	Y	-	-	Y	Y	Y	Y	-	Y	Y	Y	Y	-	-	-	-	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
LONG VARCHAR	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y
CLOB	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y
GRAPHIC	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y
VARGRAPHIC	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y
LONG VARG	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y
DBCLOB	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y
DATE	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-	-	-	-	-	-	-	-
TIME	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-	-	-	-	-	-	-
TIMESTAMP	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y

**Notes**

- See the description preceding the table for information on supported casts involving user-defined types and reference types.
- Only a DATALINK type can be cast to a DATALINK type.

## Assignments and Comparisons

---

### Assignments and Comparisons

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of INSERT, UPDATE, FETCH, SELECT INTO, VALUES INTO and SET transition-variable statements. Arguments of functions are also assigned when invoking a function. 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 the data type of the operands involved must be compatible. The compatibility rule also applies to set operations (see “Rules for Result Data Types” on page 82). The compatibility matrix is as follows.

## Assignments and Comparisons

Table 5. Data Type Compatibility for Assignments and Comparisons

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp	Binary String	UDT
Binary Integer	Yes	Yes	Yes	No	No	No	No	No	No	2
Decimal Number	Yes	Yes	Yes	No	No	No	No	No	No	2
Floating Point	Yes	Yes	Yes	No	No	No	No	No	No	2
Character String	No	No	No	Yes	No	1	1	1	No <sup>3</sup>	2
Graphic String	No	No	No	No	Yes	No	No	No	No	2
Date	No	No	No	1	No	Yes	No	No	No	2
Time	No	No	No	1	No	No	Yes	No	No	2
Timestamp	No	No	No	1	No	No	No	Yes	No	2
Binary String	No	No	No	No <sup>3</sup>	No	No	No	No	Yes	2
UDT	2	2	2	2	2	2	2	2	2	Yes

**Note:**

- 1 The compatibility of datetime values and character strings 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 75.
  - A valid string representation of a date can be assigned to a date column or compared with a date.
  - A valid string representation of a time can be assigned to a time column or compared with a time.
  - A valid string representation of a timestamp can be assigned to a timestamp column or compared with a timestamp.
- 2 A user-defined type (UDT) value is only comparable to a value defined with the same UDT. In general, assignments are supported between a distinct type value and its source data type. For additional information see “User-defined Type Assignments” on page 77.
- 3 Note that this means that character strings defined with the FOR BIT DATA attribute are also not compatible with binary strings.
- 4 A DATALINK operand can only be assigned to another DATALINK operand. The DATALINK value can only be assigned to a column if the column is defined with NO LINK CONTROL or the file exists and is not already under file link control.
- 5 For information on assignment and comparison of reference types see “Reference Type Assignments” on page 77 and “Reference Type Comparisons” on page 82.

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. (See “References to Host Variables” on page 105 for a discussion of indicator variables.)

## Assignments and Comparisons

### Numeric Assignments

The basic rule for numeric assignments is that the whole part of a decimal or integer number is never truncated. If the scale of the target number is less than the scale of the assigned number the excess digits in the fractional part of a decimal number are truncated.

#### Decimal or Integer to Floating-Point

Floating-point numbers are approximations of real numbers. Hence, when a decimal or integer number is assigned to a floating-point column or variable, the result may not be identical to the original number.

#### Floating-Point or Decimal to Integer

When a floating-point or decimal number is assigned to an integer column or variable, the fractional part of the number is lost.

#### 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 appended or eliminated, and, in the fractional part of the number, the necessary number of trailing zeros is appended, 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, or 19,0 for a big integer.

#### Floating-Point to Decimal

When a floating-point number is converted to decimal, the number is first converted to a temporary decimal number of precision 31, and then, if necessary, truncated to the precision and scale of the target. In this conversion, the number is rounded (using floating-point arithmetic) to a precision of 31 decimal digits. As a result, a number less than  $0.5 \cdot 10^{-31}$  is reduced to 0. The scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

### String Assignments

There are two types of assignments:

- *storage assignment* is when a value is assigned to a column or parameter of a function
- *retrieval assignment* is when a value is assigned to a host variable.

The rules for string assignment differ based on the assignment type.

### Storage Assignment

The basic rule is that the length of the string assigned to a column or function parameter must not be greater than the length attribute of the column or the function parameter. When the length of the string is greater than the length attribute of the column or the function parameter, the following actions may occur:

- the string is assigned with trailing blanks truncated (from all string types except long strings) to fit the length attribute of the target column or function parameter
- an error is returned (SQLSTATE 22001) when:
  - non-blank characters would be truncated from other than a long string
  - any character (or byte) would be truncated from a long string.

When a string is assigned to a fixed-length column 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 columns defined with the FOR BIT DATA attribute.

### Retrieval Assignment

The length of a string assigned to a host variable may be longer than the length attribute of the host variable. When a string is assigned to a host variable and the length of the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters (or bytes). When this occurs, a warning is returned (SQLSTATE 01004) and the value 'W' is assigned to the SQLWARN1 field of the SQLCA.

Furthermore, if an indicator variable is provided, and the source of the value is not a LOB, the indicator variable is set to the original length of the string.

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.

Retrieval assignment of C NUL-terminated host variables is handled based on options specified with the PREP or BIND command. See the section on programming in C and C++ in the *Embedded SQL Programming Guide* for details.

### Conversion Rules for String Assignments

A character string or graphic string assigned to a column or host variable is first converted, if necessary, to the coded character set of the target. Character conversion is necessary only if all of the following are true:

- The code pages are different.
- The string is neither null nor empty.

---

<sup>14</sup> When acting as a DRDA application server, input host variables are converted to the code page of the application server, even if being assigned, compared or combined with a FOR BIT DATA column. If the SQLDA has been modified to identify the input host variable as FOR BIT DATA, conversion is not performed.

## Assignments and Comparisons

- Neither string has a code page value of 0 (FOR BIT DATA).<sup>14</sup>

### MBCS Considerations for Character String Assignments

There are several considerations when assigning character strings that could contain both single and multi-byte characters. These considerations apply to all character strings, including those defined as FOR BIT DATA.

- Blank padding is always done using the single-byte blank character (X'20').
- Blank truncation is always done based on the single-byte blank character (X'20'). The double-byte blank character is treated as any other character with respect to truncation.
- Assignment of a character string to a host variable may result in fragmentation of MBCS characters if the target host variable is not large enough to contain the entire source string. If an MBCS character is fragmented, each byte of the MBCS character fragment in the target is set to a single-byte blank character (X'20'), no further bytes are moved from the source, and SQLWARN1 is set to 'W' to indicate truncation. Note that the same MBCS character fragment handling applies even when the character string is defined as FOR BIT DATA.

### DBCS Considerations for Graphic String Assignments

Graphic string assignments are processed in a manner analogous to that for character strings. Graphic string data types are compatible only with other graphic string data types, and never with numeric, character string, or datetime data types.

If a graphic string value is assigned to a graphic string column, the length of the value must not be greater than the length of the column.

If a graphic string value (the 'source' string) is assigned to a fixed length graphic string data type (the 'target', which can be a column or host variable), and the length of the source string is less than that of the target, the target will contain a copy of the source string which has been padded on the right with the necessary number of double-byte blank characters to create a value whose length equals that of the target.

If a graphic string value is assigned to a graphic string host variable and the length of the source string is greater than the length of the host variable, the host variable will contain a copy of the source string which has been truncated on the right by the necessary number of double-byte characters to create a value whose length equals that of the host variable. (Note that for this scenario, truncation need not be concerned with bisection of a double-byte character; if bisection were to occur, either the source value or target host variable would be an ill-defined graphic string data type.) The warning flag SQLWARN1 in the SQLCA will be set to 'W'. The indicator variable, if specified, will contain the original length (in double-byte characters) of the source string. In the case of DBCLOB, however, the indicator variable does not contain the original length.

Retrieval assignment of C NUL-terminated host variables (declared using `wchar_t`) is handled based on options specified with the PREP or BIND command. See the section on programming in C and C++ in the *Embedded SQL Programming Guide* for details.



### Datetime Assignments

The basic rule for datetime assignments is that a DATE, TIME, or TIMESTAMP value may only be assigned to a column with a matching data type (whether DATE, TIME, or TIMESTAMP) or to a fixed- or varying-length character string variable or string column. The assignment must not be to a LONG VARCHAR, BLOB, or CLOB variable or column.

When a datetime value is assigned to a character string variable or string column, conversion to a string representation is automatic. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target will vary, depending on the format of the string representation. If the length of the target is greater than required, and the target is a fixed-length string, 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.

When the target is a host variable, the following rules apply:

**For a DATE:** If the variable length is less than 10 bytes, an error occurs.

**For a TIME:** If the USA format is used, the length of the variable must not be less than 8; in other formats the length must not be less than 5.

If ISO or JIS formats are used, and if the length of the host variable is less than 8, the seconds part of the time is omitted from the result and assigned to the indicator variable, if provided. The SQLWARN1 field of the SQLCA is set to indicate the omission.

**For a TIMESTAMP:** If the host variable is less than 19 bytes, an error occurs. If the length is less than 26, but greater than or equal to 19 bytes, trailing digits of the microseconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

For further information on string lengths for datetime values, see “Datetime Values” on page 60.

### DATALINK Assignments

The assignment of a value to a DATALINK column results in the establishment of a link to a file unless the linkage attributes of the value are empty or the column is defined with NO LINK CONTROL. In cases where a linked value already exists in the column, that file is unlinked. Assigning a null value where a linked value already exists also unlinks the file associated with the old value.

If the application provides the same data location as already exists in the column, the link is retained. There are two reasons that this might be done:

- the comment is being changed
- if the table is placed in Datalink Reconcile Not Possible (DRNP) state, the links in the table can be reinstated by providing linkage attributes identical to the ones in the column.

A DATALINK value may be assigned to a column in any of the following ways:

## Assignments and Comparisons

- The DLVALUE scalar function can be used to create a new DATALINK value and assign it to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.
- A DATALINK value can be constructed in a CLI parameter using the CLI function SQLBuildDataLink. This value can then be assigned to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.

When assigning a value to a DATALINK column, the following error conditions return SQLSTATE 428D1:

- Data Location (URL) format is invalid (reason code 21).
- File server is not registered with this database (reason code 22).
- Invalid link type specified (reason code 23).
- Invalid length of comment or URL (reason code 27).

Note that the size of a URL parameter or function result is the same on both input or output and is bound by the length of the DATALINK column. However, in some cases the URL value returned has an access token attached. In situations where this is possible, the output location must have sufficient storage space for the access token and the length of the DATALINK column. Hence, the actual length of the comment and URL in its fully expanded form, including any default URL scheme or default hostname, provided on input should be restricted to accommodate the output storage space. If the restricted length is exceeded, this error is raised.

When the assignment is also creating a link, the following errors can occur:

- File server not currently available (SQLSTATE 57050).
- File does not exist (SQLSTATE 428D1, reason code 24).
- Referenced file cannot be accessed for linking (reason code 26).
- File already linked to another column (SQLSTATE 428D1, reason code 25).

Note that this error will be raised even if the link is to a different database.

In addition, when the assignment removes an existing link, the following errors can occur:

- File server not currently available (SQLSTATE 57050).
- File with referential integrity control is not in a correct state according to the DB2 DataLinks File Manager (SQLSTATE 58004).

A DATALINK value may be retrieved from the database in either of the following ways:

- Portions of a DATALINK value can be assigned to host variables by use of scalar functions (such as DLLINKTYPE or DLURLPATH).

## Assignments and Comparisons

Note that usually no attempt is made to access the file server at retrieval time.<sup>15</sup> It is therefore possible that subsequent attempts to access the file server through file system commands might fail.

When retrieving a DATALINK, the registry of file servers at the database server is checked to confirm that the file server is still registered with the database server (SQLSTATE 55022). In addition, a warning may be returned when retrieving a DATALINK value because the table is in reconcile pending or reconcile not possible state (SQLSTATE 01627).

### User-defined Type Assignments

With user-defined types, different rules are applied for assignments to host variables than are used for all other assignments.

Assignment to host variables is done based on the source type of the distinct type. That is, it follows the rule:

A value of a distinct type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the source type of this distinct type is assignable to this host variable.

If the target of the assignment is a column, the source data type must be castable to the target data type as described in “Casting Between Data Types” on page 67 for user-defined types.

### Reference Type Assignments

A reference type with a target type of  $T$  can be assigned to a reference type column that is also a reference type with target type of  $S$  where  $S$  is a supertype of  $T$ . If an assignment is made to a scoped reference column or variable, no check is performed to ensure that the actual value being assigned exists in the target table or view defined by the scope.

Assignment to host variables is done based on the representation type of the reference type. That is, it follows the rule:

A value of a reference type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the representation type of this reference type is assignable to this host variable.

If the target of the assignment is a column, and the right hand side of the assignment is a host variable, the host variable must be explicitly cast to the reference type of the target column.

---

<sup>15</sup> It may be necessary to access the file server to determine the prefix name associated with a path. This can be changed at the file server when the mount point of a file system is moved. First access of a file on a server will cause the required values to be retrieved from the file server and cached at the database server for the subsequent retrieval of DATALINK values for that file server. An error is returned if the file server cannot be accessed (SQLSTATE 57050).

## Assignments and Comparisons

### 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 floating-point and the other is integer or decimal, the comparison is made with a temporary copy of the other number, which has been converted to double-precision floating-point.

Two floating-point numbers are equal only if the bit configurations of their normalized forms are identical.

### String Comparisons

Character strings are compared according to the collating sequence specified when the database was created, except those with a FOR BIT DATA attribute which are always compared according to their bit values.

When comparing character strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with single-byte blanks sufficient to extend its length to that of the longer string. This logical extension is done for all character strings including those tagged as FOR BIT DATA.

Character strings (except character strings tagged as FOR BIT DATA) are compared according to the collating sequence specified when the database was created (see the *Administration Guide* for more information on collating sequences specified at database creation time). For example, the default collating sequence supplied by the database manager may give lowercase and uppercase versions of the same character the same weight. The database manager performs a two-pass comparison to ensure that only identical strings are considered equal to each other. In the first pass, strings are compared according to the database collating sequence. If the weights of the characters in the strings are equal, a second "tie-breaker" pass is performed to compare the strings on the basis of their actual code point values.

Two strings are equal if they are both empty or if all corresponding bytes are equal. If either operand is null, the result is unknown.

Long strings and LOB strings are not supported in any comparison operations that use the basic comparison operators ( $=$ ,  $<>$ ,  $<$ ,  $>$ ,  $<=$ , and  $>=$ ). They are supported in comparisons using the LIKE predicate and the POSSTR function. See "LIKE Predicate" on page 146 and see "POSSTR" on page 265 for details.

Portions of long strings and LOB strings of up to 4000 bytes can be compared using the SUBSTR and VARCHAR scalar functions. For example, given the columns:

## Assignments and Comparisons

```
MY_SHORT_CLOB  CLOB(300)
MY_LONG_VAR    LONG VARCHAR
```

then the following is valid:

```
WHERE VARCHAR(MY_SHORT_CLOB) > VARCHAR(SUBSTR(MY_LONG_VAR,1,300))
```

Examples:

For these examples, 'A', 'B', 'a', and 'b', have the code point values X'41', X'42', X'61', and X'62' respectively.

Consider a collating sequence where the characters 'A', 'B', 'a', 'b' have weights 75, 101, 74, and 100. Then:

```
'a' < 'A' < 'b' < 'B'
```

and

```
'aa' < 'aA' < 'ab' < 'aB' < 'Aa' < 'AA' < 'Ab' < 'AB'.
```

However, if the values being compared have the FOR BIT DATA attribute, the collating sequence is ignored, and:

```
'A' < 'B' < 'a' < 'b'
```

and

```
'AA' < 'AB' < 'Aa' < 'Ab' < 'aA' < 'aB' < 'aa' < 'ab'.
```

Now consider a collating sequence where the characters 'A', 'B', 'a', 'b' have (non-unique) weights 74, 75, 74, and 75. Then:

```
'A' < 'a' < 'B' < 'b'
```

and

```
'AA' < 'Aa' < 'aA' < 'aa' < 'AB' < 'Ab' < 'aB' < 'ab'.
```

### Conversion Rules for Comparison

When two strings are compared, one of the strings is first converted, if necessary, to the coded character set of the other string. For details, see “Rules for String Conversions” on page 85.

### Ordering of Results

Results that require sorting are ordered based on the string comparison rules discussed in “String Comparisons” on page 78. The comparison is performed at the database server. On returning results to the client application, code page conversion may be performed. This subsequent code page conversion does not affect the order of the server-determined result set.

### MBCS Considerations for String Comparisons

Mixed SBCS/MBCS character strings are compared according to the collating sequence specified when the database was created. For databases created with default (SYSTEM) collation sequence, all single-byte ASCII characters are sorted in correct

## Assignments and Comparisons

order, but double-byte characters are not necessarily in code point sequence. For databases created with IDENTITY sequence, all double-byte characters are correctly sorted in their code point order, but single-byte ASCII characters are sorted in their code point order as well. For databases created with COMPATIBILITY sequence, a compromise order is used that sorts properly for most double-byte characters, and is almost correct for ASCII. This was the default collation table in DB2 Version 2.

Mixed character strings are compared byte-by-byte. This may result in unusual results for multi-byte characters that occur in mixed strings, because each byte is considered independently.

Example:

For this example, 'A', 'B', 'a', and 'b' double-byte characters have the code point values X'8260', X'8261', X'8281', and X'8282', respectively.

Consider a collating sequence where the code points X'8260', X'8261', X'8281', and X'8282' have weights 96, 65, 193, and 194. Then:

`'B' < 'A' < 'a' < 'b'`

and

`'AB' < 'AA' < 'Aa' < 'Ab' < 'aB' < 'aA' < 'aa' < 'ab'`

Graphic string comparisons are processed in a manner analogous to that for character strings.

Graphic string comparisons are valid between all graphic string data types except LONG VARGRAPHIC. LONG VARGRAPHIC and DBCLOB data types are not allowed in a comparison operation.

For graphic strings, the collating sequence of the database is not used. Instead, graphic strings are always compared based on the numeric (binary) values of their corresponding bytes.

Using the previous example, if the literals were graphic strings, then:

`'A' < 'B' < 'a' < 'b'`

and

`'AA' < 'AB' < 'Aa' < 'Ab' < 'aA' < 'aB' < 'aa' < 'ab'`

When comparing graphic strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with double-byte blank characters sufficient to extend its length to that of the longer string.

Two graphic values are equal if they are both empty or if all corresponding graphics are equal. If either operand is null, the result is unknown. If two values are not equal, their relation is determined by a simple binary string comparison.

## Assignments and Comparisons

As indicated in this section, comparing strings on a byte by byte basis can produce unusual results; that is, a result that differs from what would be expected in a character by character comparison. The examples shown here assume the same MBCS code page, however, the situation can be further complicated when using different multi-byte code pages with the same national language. For example, consider the case of comparing a string from a Japanese DBCS code page and a Japanese EUC code page.

### Datetime Comparisons

A DATE, TIME, or TIMESTAMP value may 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 farther 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 is implied.

Comparisons involving TIMESTAMP values are chronological without regard to representations that might be considered equivalent.

Example:

```
TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
```

### User-defined Type Comparisons

Values with a user-defined type can only be compared with values of exactly the same user-defined type. The user-defined type must have been defined using the WITH COMPARISONS clause.

Example:

Given the following YOUTH distinct type and CAMP\_DB2\_ROSTER table:

```
CREATE DISTINCT TYPE YOUTH AS INTEGER WITH COMPARISONS
```

```
CREATE TABLE CAMP_DB2_ROSTER  
( NAME VARCHAR(20),  
  ATTENDEE_NUMBER INTEGER NOT NULL,  
  AGE YOUTH,  
  HIGH_SCHOOL_LEVEL YOUTH)
```

The following comparison is valid:

```
SELECT * FROM CAMP_DB2_ROSTER  
WHERE AGE > HIGH_SCHOOL_LEVEL
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER  
WHERE AGE > ATTENDEE_NUMBER
```

## Rules for Result Data Types

However, AGE can be compared to ATTENDEE\_NUMBER by using a function or CAST specification to cast between the distinct type and the source type. The following comparisons are all valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE INTEGER(AGE) > ATTENDEE_NUMBER
```

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE CAST( AGE AS INTEGER) > ATTENDEE_NUMBER
```

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > YOUTH(ATTENDEE_NUMBER)
```

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > CAST(ATTENDEE_NUMBER AS YOUTH)
```

## Reference Type Comparisons

Reference type values can be compared only if their target types have a common supertype. The appropriate comparison function will only be found if the schema name of the common supertype is included in the function path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

---

## Rules for Result Data Types

The data types of a result are determined by rules which are applied to the operands in an operation. This section explains those rules.

These rules apply to:

- Corresponding columns in fullselects of set operations (UNION, INTERSECT and EXCEPT)
- Result expressions of a CASE expression
- Arguments of the scalar function COALESCE (or VALUE)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

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

The rules involving various data types follow. In some cases, a table is used to show the possible result data types.

These tables identify the data type of the result, including the applicable length or precision and scale. The result type is determined by considering the operands. If there is more than one pair of operands, start by considering the first pair. This gives a result type which is considered 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



## Rules for Result Data Types

for the operation. Processing of operations is done from left to right so that the intermediate result types are important when operations are repeated. For example, consider a situation involving:

`CHAR(2) UNION CHAR(4) UNION VARCHAR(3)`

The first pair results in a type of `CHAR(4)`. The result values always have 4 characters. The final result type is `VARCHAR(4)`. Values in the result from the first `UNION` operation will always have a length of 4.

### Character Strings

Character strings are compatible with other character strings. Character strings include data types `CHAR`, `VARCHAR`, `LONG VARCHAR`, and `CLOB`.

If one operand is...	And the other operand is...	The data type of the result is...
<code>CHAR(x)</code>	<code>CHAR(y)</code>	<code>CHAR(z)</code> where $z = \max(x,y)$

The code page of the result character string will be derived based on the “Rules for String Conversions” on page 85.

### Graphic Strings

Graphic strings are compatible with other graphic strings. Graphic strings include data types `GRAPHIC`, `VARGRAPHIC`, `LONG VARGRAPHIC`, and `DBCLOB`.

If one operand is...	And the other operand is...	The data type of the result is...
<code>GRAPHIC(x)</code>	<code>GRAPHIC(y)</code>	<code>GRAPHIC(z)</code> where $z = \max(x,y)$

The code page of the result graphic string will be derived based on the “Rules for String Conversions” on page 85.

### Binary Large Object (BLOB)

A `BLOB` is compatible only with another `BLOB` and the result is a `BLOB`. The `BLOB` scalar function should be used to cast from other types if they should be treated as `BLOB` types (see “`BLOB`” on page 193). The length of the result `BLOB` is the largest length of all the data types.

### Numeric

Numeric types are compatible with other numeric types. Numeric types include `SMALLINT`, `INTEGER`, `BIGINT`, `DECIMAL`, `REAL` and `DOUBLE`.

## Rules for Result Data Types

If one operand is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
BIGINT	BIGINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	SMALLINT	BIGINT
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where $p = x + \max(w-x, 5)$ <sup>1</sup>
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where $p = x + \max(w-x, 11)$ <sup>1</sup>
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where $p = x + \max(w-x, 19)$ <sup>1</sup>
DECIMAL(w,x)	DECIMAL(y,z)	DECIMAL(p,s) where $p = \max(x,z) + \max(w-x, y-z)$ <sup>1</sup> $s = \max(x,z)$
REAL	REAL	REAL
REAL	DECIMAL, BIGINT, INTEGER, or SMALLINT	DOUBLE
DOUBLE	any numeric	DOUBLE

**Note:**

1. Precision cannot exceed 31.

### DATE

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

### TIME

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

### TIMESTAMP

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

### DATALINK

A datalink is compatible with another datalink. The data type of the result is DATALINK. The length of the result DATALINK is the largest length of all the data types.

### User-defined Types

#### Distinct Types

A user-defined distinct type is compatible only with the same user-defined distinct type. The data type of the result is the user-defined distinct type.

#### Reference Types

A reference type is compatible with another reference type provided that their target types have a common supertype. The data type of the result is a reference type having the common supertype as the target type. If all operands have the identical scope table, the result has that scope table. Otherwise the result is unscoped.

### Nullable Attribute of Result

With the exception of INTERSECT and EXCEPT, the result allows nulls unless both operands do not allow nulls.

- For INTERSECT, if either operand does not allow nulls the result does not allow nulls (the intersection would never be null).
- For EXCEPT, if the first operand does not allow nulls the result does not allow nulls (the result can only be values from the first operand).

---

## Rules for String Conversions

The code page used to perform an operation is determined by rules which are applied to the operands in that operation. This section explains those rules.

These rules apply to:

- Corresponding string columns in fullselects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE expression
- Arguments of the scalar function COALESCE (and VALUE)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In each case, the code page of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the coded character set identified by that code page. A character that has no valid conversion is mapped to the substitution character for the character set and SQLWARN10 is set to 'W' in the SQLCA.

The code page of the result is determined by the code pages of the operands. The code pages of the first two operands determine an intermediate result code page, this code page and the code page of the next operand determine a new intermediate result

## Rules for String Conversions

code page (if applicable), and so on. The last intermediate result code page and the code page of the last operand determine the code page of the result string or column. For each pair of code pages, the result is determined by the sequential application of the following rules:

- If the code pages are equal, the result is that code page.
- If either code page is BIT DATA (code page 0), the result code page is BIT DATA.  
14
- Otherwise, the result code page is determined by Table 6. An entry of 'first' in the table means the code page from the first operand is selected and an entry of 'second' means the code page from the second operand is selected.

*Table 6. Selecting the Code Page of the Intermediate Result*

First Operand	Second Operand				
	Column Value	Derived Value	Constant	Special Register	Host Variable
Column Value	first	first	first	first	first
Derived Value	second	first	first	first	first
Constant	second	second	first	first	first
Special Register	second	second	first	first	first
Host Variable	second	second	second	second	first

An intermediate result is considered to be a derived value operand. An expression that is not a single column value, constant, special register, or host variable is also considered a derived value operand. There is an exception to this if the expression is a CAST specification (or a call to a function that is equivalent). In this case, the *kind* for the first operand is based on the first argument of the CAST specification.

View columns are considered to have the operand type of the object on which they are ultimately based. For example, a view column defined on a table column is considered to be a column value, whereas a view column based on a string expression (for example, A CONCAT B) is considered to be a derived value.

Conversions to the coded character set of the result are performed, if necessary, for:

- An operand of the concatenation operator
- The selected argument of the COALESCE (or VALUE) scalar function
- The selected result expression of the CASE expression
- The expressions of the in list of the IN predicate
- The corresponding expressions of a multiple row VALUES clause
- The corresponding columns involved in set operations.

Character conversion is necessary if all of the following are true:

- The code pages are different
- Neither string is BIT DATA
- The string is neither null nor empty

- The code page conversion selection table indicates that conversion is necessary.

### Examples

*Example 1:* Given the following:

Expression	Type	Code Page
COL_1	column	850
HV_2	host variable	437

When evaluating the predicate:

```
COL_1 CONCAT :HV_2
```

The result code page of the two operands is 850, since the dominant operand is the column COL\_1.

*Example 2:* Using the information from the previous example, when evaluating the predicate:

```
COALESCE(COL_1, :HV_2:NULLIND,)
```

The result code page is 850. Therefore the result code page for the COALESCE scalar function will be the code page 850.

---

## Partition Compatibility

*Partition compatibility* is defined between the base data types of corresponding columns of partitioning keys. Partition compatible data types have the property that two variables, one of each type, with the same value, are mapped to the same partitioning map index by the same partitioning function.

Table 7 on page 88 shows the compatibility of data types in partitions.

Partition compatibility has the following characteristics:

- Internal formats are used for DATE, TIME, and TIMESTAMP. They are not compatible with each other, and none are compatible with CHAR.
- Partition compatibility is not affected by columns with NOT NULL or FOR BIT DATA definitions.
- NULL values of compatible data types are treated identically. Different results might be produced for NULL values of non-compatible data types.
- Base datatype of the UDT is used to analyze partition compatibility.
- Decimals of the same value in the partitioning key are treated identically, even if their scale and precision differ.
- Trailing blanks in character strings (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) are ignored by the system-provided hashing function.

## Constants

- CHAR or VARCHAR of different lengths are compatible data types.
- REAL or DOUBLE values that are equal are treated identically even though their precision differs.

Table 7. Partition Compatibilities

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp	UDT
Binary Integer	Yes	No	No	No	No	No	No	No	1
Decimal Number	No	Yes	No	No	No	No	No	No	1
Floating Point	No	No	Yes	No	No	No	No	No	1
Character String <sup>3</sup>	No	No	No	Yes <sup>2</sup>	No	No	No	No	1
Graphic String <sup>3</sup>	No	No	No	No	Yes	No	No	No	1
Date	No	No	No	No	No	Yes	No	No	1
Time	No	No	No	No	No	No	Yes	No	1
Timestamp	No	No	No	No	No	No	No	Yes	1
UDT	1	1	1	1	1	1	1	1	1

**Note:**

- <sup>1</sup> A user-defined type (UDT) value is partition compatible with the source type of the UDT or any other UDT with a partition compatible source type.
- <sup>2</sup> The FOR BIT DATA attribute does not affect the partition compatibility.
- <sup>3</sup> Note that data types LONG VARCHAR, LONG VARGRAPHIC, CLOB, DBCLOB, and BLOB are not applicable for partition compatibility since they are not supported in partitioning keys.

## Constants

A *constant* (sometimes 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.

All constants have the attribute NOT NULL.

A negative zero value in a numeric constant (-0) is the same value as a zero without the sign (0).

## Integer Constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is a large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of large integer but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Note that the smallest literal representation of an integer constant is `-2147483647` and not `-2147483648`, which is the limit for integer values. Similarly, the smallest literal representation of a big integer constant is `-9,223,372,036,854,775,807` and not `-9,223,372,036,854,775,808` which is the limit for big integer values.

### Examples

```
64      -15      +100      32767      720176      12345678901
```

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 may include a sign and a decimal point; the second number may include a sign but not a decimal point. The data type of a floating-point constant is double precision. 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.

### Examples

```
15E1      2.E5      2.2E-1      +5.E+2
```

## Decimal Constants

A *decimal constant* is a signed or unsigned number that consists of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. It must be in the range of decimal numbers. The precision is the total number of digits (including leading and trailing zeros); the scale is the number of digits to the right of the decimal point (including trailing zeros).

### Examples

```
25.5      1000.      -15.      +37589.3333333333
```

## Character String Constants

A *character string constant* specifies a varying-length character string and consists of a sequence of characters that starts and ends with an apostrophe ('). This form of string constant specifies the character string contained between the string delimiters. The length of the character string must not be greater than 4000 bytes. Two consecutive string delimiters are used to represent one string delimiter within the character string.

## Constants

### Examples

```
'12/14/1985'  
'32'  
'DON'T CHANGE'
```

### Unequal Code Page Considerations

The constant value is always converted to the database code page when it is bound to the database. It is considered to be in the database code page. Therefore, if used in an expression that combines a constant with a FOR BIT DATA column, of which the result is FOR BIT DATA, the constant value will *not* be converted from its database code page representation when used.

## Hexadecimal Constants

A *hexadecimal constant* specifies a varying-length character string with the code page of the application server.

The format of a hexadecimal string constant is an X followed by a sequence of characters that starts and ends with an apostrophe (single quote). The characters between the apostrophes must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 4000, otherwise an error is raised (SQLSTATE -54002). A hexadecimal digit represents 4 bits. It is specified as a digit or any of the letters A through F (uppercase or lowercase) where A represents the bit pattern '1010', B the bit pattern '1011', etc. If a hexadecimal constant is improperly formatted (e.g. it contains an invalid hexadecimal digit or an odd number of hexadecimal digits), an error is raised (SQLSTATE 42606).

### Examples

```
X'FFFF'      representing the bit pattern '1111111111111111'  
  
X'4672616E6B' representing the VARCHAR pattern of the ASCII string 'Frank'
```

## Graphic String Constants

A *graphic string constant* specifies a varying-length graphic string and consists of a sequence of double-byte characters that starts and ends with a single-byte apostrophe (') and is preceded by a single-byte G or N. This form of string constant specifies the graphic string contained between the string delimiters. The length of the graphic string must be an even number of bytes and must not be greater than 4000 bytes.

### Examples

```
G'double-byte character string'  
N'double-byte character string'
```

### MBCS Considerations

The apostrophe must not appear as part of an MBCS character to be considered a delimiter.



**Using Constants with User-defined Types:** User-defined types have strong typing. This means that a user-defined type is only compatible with its own type. A constant, however, has a built-in type. Therefore, an operation involving a user-defined type and a constant is only possible if the user-defined type has been cast to the constant's built-in type or the constant has been cast to the user-defined type (see “CAST Specifications” on page 131 for information on casting). For example, using the table and distinct type in “User-defined Type Comparisons” on page 81, the following comparisons with the constant 14 are valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > CAST(14 AS YOUTH)

SELECT * FROM CAMP_DB2_ROSTER
WHERE CAST(AGE AS INTEGER) > 14
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER
WHERE AGE > 14
```

---

## Special Registers

A *special register* is a storage area that is defined for an application process by the database manager and is used to store information that can be referenced in SQL statements. Special registers are in the database code page.

### CURRENT DATE

The CURRENT DATE special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

#### Example

Using the PROJECT table, set the project end date (PRENDATE) of the MA2111 project (PROJNO) to the current date.

```
UPDATE PROJECT
SET PRENDATE = CURRENT DATE
WHERE PROJNO = 'MA2111'
```

### CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of intra-partition parallelism for the execution of dynamic SQL statements.<sup>16</sup> The data type of the register is CHAR(5). Valid values are 'ANY ' or the string representation of an integer between 1 and 32767, inclusive.

---

<sup>16</sup> For static SQL, the DEGREE bind option provides the same control.

## Special Registers

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intra-partition parallelism.

If the value of CURRENT DEGREE represented as an integer is greater than 1 and less than or equal to 32767 when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism with the specified degree.

If the value of CURRENT DEGREE is 'ANY' when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism using a degree determined by the database manager.

The actual runtime degree of parallelism will be the lower of:

- Maximum query degree (max\_querydegree) configuration parameter
- Application runtime degree
- SQL statement compilation degree

If the intra\_parallel database manager configuration parameter is set to NO, the value of the CURRENT DEGREE special register will be ignored for the purpose of optimization, and the statement will not use intra-partition parallelism.

See the *Administration Guide* for a description of parallelism and a list of restrictions.

The value can be changed by executing the SET CURRENT DEGREE statement (see "SET CURRENT DEGREE" on page 716 for information on this statement).

The initial value of CURRENT DEGREE is determined by the dft\_degree database configuration parameter. See the *Administration Guide* for a description of this configuration parameter.

## CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a CHAR(8) value which controls the behaviour of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables (for more information see the *Administration Guide*). This information does not include the Explain snapshot.

The possible values are YES, NO, and EXPLAIN.<sup>17</sup>

**YES** Enables the explain facility and causes explain information for a dynamic SQL statement to be captured when the statement is compiled.

**EXPLAIN** Enables the facility like YES, however, the dynamic statements are not executed.

**NO** Disables the Explain facility.

---

<sup>17</sup> For static SQL, the EXPLAIN bind option provides the same control. In the case of the PREP and BIND commands, the EXPLAIN option values are: YES, NO and ALL.

The initial value is NO.

Its value can be changed by the SET CURRENT EXPLAIN MODE statement (see “SET CURRENT EXPLAIN MODE” on page 718 for information on this statement).

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked (see Table 108 on page 957 for details). The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option (see Table 109 on page 958 for details).

*Example:* Set the host variable EXPL\_MODE (char(8)) to the value currently in the CURRENT EXPLAIN MODE special register.

```
VALUES CURRENT EXPLAIN MODE
INTO :EXPL_MODE
```

### CURRENT EXPLAIN SNAPSHOT

The CURRENT EXPLAIN SNAPSHOT special register holds a CHAR(8) value which controls the behavior of the Explain snapshot facility. This facility generates compressed information including access plan information, operator costs, and bind-time statistics (for more information see the *Administration Guide* ).

Only the following statements consider the value of this register: DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES or VALUES INTO.

The possible values are YES, NO, and EXPLAIN.<sup>18</sup>

- YES** Enables the snapshot facility and takes a snapshot of the internal representation of a dynamic SQL statement as the statement is compiled.
- EXPLAIN** Enables the facility like YES, however, the dynamic statements are not executed.
- NO** Disables the Explain snapshot facility.

The initial value is NO.

Its value can be changed by the SET CURRENT EXPLAIN SNAPSHOT statement (see “SET CURRENT EXPLAIN SNAPSHOT” on page 720).

The CURRENT EXPLAIN SNAPSHOT and CURRENT EXPLAIN MODE special register values interact when the Explain facility is invoked (see Table 108 on page 957 for details). The CURRENT EXPLAIN SNAPSHOT special register also interacts with the EXPLSNAP bind option (see Table 110 on page 959 for details).

#### Example

Set the host variable EXPL\_SNAP (char(8)) to the value currently in the CURRENT EXPLAIN SNAPSHOT special register.

---

<sup>18</sup> For static SQL, the EXPLSNAP bind option provides the same control. In the case of the PREP and BIND commands, the EXPLSNAP option values are: YES, NO and ALL.

## Special Registers

```
VALUES CURRENT EXPLAIN SNAPSHOT
INTO :EXPL_SNAP
```

### CURRENT NODE

The CURRENT NODE special register specifies an INTEGER value that identifies the coordinator node number (the partition to which an application connects).

CURRENT NODE returns 0 if the database instance is not defined to support partitioning (no db2nodes.cfg file<sup>19</sup>).

The CURRENT NODE can be changed by the CONNECT statement, but only under certain conditions (see “CONNECT (Type 1)” on page 432).

#### Example

Set the host variable APPL\_NODE (integer) to the number of the partition to which the application is connected.

```
VALUES CURRENT NODE
INTO :APPL_NODE
```

### CURRENT PATH

The CURRENT PATH special register specifies a VARCHAR(254) value that identifies the *SQL path* to be used to resolve function references and data type references that are used in dynamically prepared SQL statements.<sup>20</sup> CURRENT PATH is also used to resolve stored procedure references in CALL statements. The initial value is the default value specified below. For static SQL, the FUNCPATH bind option provides a SQL path that is used for function and data type resolution (see the *Command Reference* for more information on the FUNCPATH bind option).

The CURRENT PATH special register contains a list of one or more schema-names, where the schema-names are enclosed in double quotes and separated by commas (any quotes within the string are repeated as they are in any delimited identifier).

For example, a SQL path specifying that the database manager is to first look in the FERMAT, then XGRAPHIC, then SYSIBM schemas is returned in the CURRENT PATH special register as:

```
"FERMAT", "XGRAPHIC", "SYSIBM"
```

The default value is "SYSIBM","SYSFUN",X where X is the value of the USER special register delimited by double quotes.

Its value can be changed by the SET CURRENT FUNCTION PATH statement (see “SET PATH” on page 731). The schema SYSIBM does not need to be specified. If it is

---

<sup>19</sup> For partitioned databases, the db2nodes.cfg file exists and contains partition (or node) definitions. For details refer to the *Administration Guide*.

<sup>20</sup> CURRENT FUNCTION PATH is a synonym for CURRENT PATH.

not included in the SQL path, it is implicitly assumed as the first schema. SYSIBM does not take any of the 254 characters if it is implicitly assumed.

The use of the SQL path for function resolution is described in “Functions” on page 110. A data type that is not qualified with a schema name will be implicitly qualified with the schema name that is earliest in the SQL path and contains a data type with the same unqualified name specified. There are exceptions to this rule as described in the following statements: CREATE DISTINCT TYPE, CREATE FUNCTION, COMMENT ON and DROP.

### Example

Using the SYSCAT.VIEWS catalog view, find all views that were created with the exact same setting as the current value of the CURRENT PATH special register.

```
SELECT VIEWNAME, VIEWSCHEMA FROM SYSCAT.VIEWS
WHERE FUNC_PATH = CURRENT PATH
```

## CURRENT QUERY OPTIMIZATION

The CURRENT QUERY OPTIMIZATION special register specifies an INTEGER value that controls the class of query optimization performed by the database manager when binding dynamic SQL statements. The QUERYOPT bind option controls the class of query optimization for static SQL statements (see the *Command Reference* for additional information on the QUERYOPT bind option). The possible values range from 0 to 9. For example, if the query optimization class is set to the minimal class of optimization (0), then the value in the special register is 0. The default value is determined by the `dft_queryopt` database configuration parameter.

Its value can be changed by the SET CURRENT QUERY OPTIMIZATION statement (see “SET CURRENT QUERY OPTIMIZATION” on page 724).

### Example

Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```
SELECT PKGNAME, PKGSCHEMA FROM SYSCAT.PACKAGES
WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```

## CURRENT REFRESH AGE

The CURRENT REFRESH AGE special register specifies a timestamp duration value with a data type of DECIMAL(20,6). This duration is the maximum duration since a REFRESH TABLE statement has been processed on a REFRESH DEFERRED summary table such that the summary table can be used to optimize the processing of a query. If CURRENT REFRESH AGE has a value of 9999999999999999 (ANY), and QUERY OPTIMIZATION class is 5 or more, REFRESH DEFERRED summary tables are considered to optimize the processing of a dynamic SQL query. A summary table with the REFRESH IMMEDIATE attribute and not in check pending state is assumed to have a refresh age of zero.

## Special Registers

Its value can be changed by the SET CURRENT REFRESH AGE statement (see “SET CURRENT REFRESH AGE” on page 727). Summary tables defined with REFRESH DEFERRED are never considered by static embedded SQL queries.

The initial value of CURRENT REFRESH AGE is zero.

### CURRENT SCHEMA

The CURRENT SCHEMA special register specifies a CHAR(8) value that identifies the schema name used to qualify unqualified database object references where applicable in dynamically prepared SQL statements.<sup>21</sup>

The initial value of CURRENT SCHEMA is the authorization ID of the current session user.

Its value can be changed by the SET SCHEMA statement (see “SET SCHEMA” on page 733).

The QUALIFIER bind option controls the schema name used to qualify unqualified database object references where applicable for static SQL statements (see *Command Reference* for more information).

#### Example

Set the schema for object qualification to 'D123'.

```
SET CURRENT SCHEMA = 'D123'
```

### CURRENT SERVER

The CURRENT SERVER special register specifies a VARCHAR(18) value that identifies the current application server. The actual name of the application server (not an alias) is contained in the register.

The CURRENT SERVER can be changed by the CONNECT statement, but only under certain conditions (see “CONNECT (Type 1)” on page 432).

#### Example

Set the host variable APPL\_SERVE (varchar(18)) to the name of the application server to which the application is connected.

```
VALUES CURRENT SERVER  
INTO :APPL_SERVE
```

### CURRENT TIME

The CURRENT TIME special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with

---

<sup>21</sup> For compatibility with DB2 for OS/390, the special register CURRENT SQLID is treated as a synonym for CURRENT SCHEMA.

CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

### Example

Using the CL\_SCHED table, select all the classes (CLASS\_CODE) that start (STARTING) later today. Today's classes have a value of 3 in the DAY column.

```
SELECT CLASS_CODE FROM CL_SCHED
WHERE STARTING > CURRENT TIME AND DAY = 3
```

## CURRENT TIMESTAMP

The CURRENT TIMESTAMP special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading.

### Example

Insert a row into the IN\_TRAY table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables SRC (char(8)), SUB (char(64)), and TXT (varchar(200)).

```
INSERT INTO IN_TRAY
VALUES (CURRENT TIMESTAMP, :SRC, :SUB, :TXT)
```

## CURRENT TIMEZONE

The CURRENT TIMEZONE special register specifies the difference between UTC<sup>22</sup> and local time at the application server. The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds). The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC. The time is calculated from the operating system time at the moment the SQL statement is executed.<sup>23</sup>

The CURRENT TIMEZONE special register can be used wherever an expression of the DECIMAL(6,0) data type is used, for example, in time and timestamp arithmetic.

### Example

Insert a record into the IN\_TRAY table, using a UTC timestamp for the RECEIVED column.

---

<sup>22</sup> Coordinated Universal Time, formerly known as GMT.

<sup>23</sup> The CURRENT TIMEZONE value is determined from C runtime functions. See the *Quick Beginnings* for any installation requirements regarding time zone.

## Column Names

```
INSERT INTO IN_TRAY VALUES (  
    CURRENT_TIMESTAMP - CURRENT_TIMEZONE,  
    :source,  
    :subject,  
    :notetext )
```

## USER

The USER special register specifies the run-time authorization ID passed to the database manager when an application starts on a database. The data type of the register is CHAR(8).

### Example

Select all notes from the IN\_TRAY table that the user placed there himself.

```
SELECT * FROM IN_TRAY  
WHERE SOURCE = 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.
- 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 under Chapter 5, “Queries” on page 315.) 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-reference* in a FROM clause.

## Qualified Column Names

A qualifier for a column name may be a table, view, alias, or correlation name.

Whether a column name may be qualified depends on its context:



## Column Names

- Depending on the form of the COMMENT ON statement, a single column name may need to be qualified. Multiple column names must be unqualified.
- Where the column name specifies values of the column, it may be qualified at the user's option.
- In all other contexts, a column name must not be qualified.

Where a qualifier is optional, it can serve two purposes. They are described under “Column Name Qualifiers to Avoid Ambiguity” on page 101 and “Column Name Qualifiers in Correlated References” on page 102.

## 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.

```
FROM X.MYTABLE Z
```

With Z defined as a correlation name for X.MYTABLE, only Z can be used to qualify a reference to a column of that instance of X.MYTABLE in that SELECT statement.

A correlation name is associated with a table, view, alias, 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, or in different clauses of the same statement.

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, view, or alias. In the case of a nested table expression or table function, a correlation name is required to identify the result table. In the example, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

If a correlation name is specified for a table, view, or alias name, any qualified reference to a column of that instance of the table, view, or alias must use the correlation name, rather than the table, view, or alias name. For example, the reference to EMPLOYEE.PROJECT in the following example is incorrect, because a correlation name has been specified for EMPLOYEE:

Example

```
FROM: EMPLOYEE E
WHERE EMPLOYEE.PROJECT='ABC'      * incorrect*
```

The qualified reference to PROJECT should instead use the correlation name, "E", as shown below:

```
FROM EMPLOYEE E
WHERE E.PROJECT='ABC'
```

Names specified in a FROM clause are either *exposed* or *non-exposed*. A table, view, or alias name is said to be exposed in the FROM clause if a correlation name is not

## Column Names

specified. A correlation name is always an exposed name. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE but not for DEPARTMENT, so DEPARTMENT is an exposed name, and EMPLOYEE is not:

```
FROM EMPLOYEE E, DEPARTMENT
```

A table, view, or alias name that is exposed in a FROM clause may be the same as any other table name or view name exposed in that FROM clause or any correlation name in the FROM clause. This may result in ambiguous column name references which returns an error (SQLSTATE 42702).

The first two FROM clauses shown below are correct, because each one contains no more than one reference to EMPLOYEE that is exposed:

1. Given the FROM clause:

```
FROM EMPLOYEE E1, EMPLOYEE
```

a qualified reference such as EMPLOYEE.PROJECT denotes a column of the second instance of EMPLOYEE in the FROM clause. A qualified reference to the first instance of EMPLOYEE must use the correlation name "E1" (E1.PROJECT).

2. Given the FROM clause:

```
FROM EMPLOYEE, EMPLOYEE E2
```

a qualified reference such as EMPLOYEE.PROJECT denotes a column of the first instance of EMPLOYEE in the FROM clause. A qualified reference to the second instance of EMPLOYEE must use the correlation name "E2" (E2.PROJECT).

3. Given the FROM clause:

```
FROM EMPLOYEE, EMPLOYEE
```

the two exposed table names included in this clause (EMPLOYEE and EMPLOYEE) are the same. This is allowed, but references to specific column names would be ambiguous (SQLSTATE 42702).

4. Given the following statement:

```
SELECT *
FROM EMPLOYEE E1, EMPLOYEE E2          * incorrect *
WHERE EMPLOYEE.PROJECT = 'ABC'
```

the qualified reference EMPLOYEE.PROJECT is incorrect, because both instances of EMPLOYEE in the FROM clause have correlation names. Instead, references to PROJECT must be qualified with either correlation name (E1.PROJECT or E2.PROJECT).

5. Given the FROM clause:

```
FROM EMPLOYEE, X.EMPLOYEE
```

a reference to a column in the second instance of EMPLOYEE must use X.EMPLOYEE (X.EMPLOYEE.PROJECT). If X is the CURRENT SCHEMA special register value in dynamic SQL or the QUALIFIER precompile/bind option in static SQL, then the columns cannot be referenced since any such reference would be ambiguous.

## Column Names

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, these listed column names become the *exposed* names of the columns that must be used for references to the columns throughout the query. If a column name list is specified, then the column names of the underlying table become *non-exposed*.

Given the FROM clause:

```
FROM DEPARTMENT D (NUM,NAME,MGR,ANUM,LOC)
```

a qualified reference such as D.NUM denotes the first column of the DEPARTMENT table that is defined in the table as DEPTNO. A reference to D.DEPTNO using this FROM clause is incorrect since the column name DEPTNO is a non-exposed column name.

### Column Name Qualifiers to Avoid Ambiguity

In the context of a function, a GROUP BY clause, 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 expressions and table functions that might contain the column are called the *object tables* of the context. Two or more object tables might contain columns with the same name; one reason for qualifying a column name is to designate the table from which the column comes.

A nested table expression or table function will consider *table-references* that precede it in the FROM clause as object tables. The *table-references* that follow are not considered as object tables.

### Table Designators

A qualifier that designates a specific object table is called a *table designator*. The clause that identifies the object tables also establishes the table designators for them. For example, the object tables of an expression in a SELECT clause are named in the FROM clause that follows it:

```
SELECT CORZ.COLA, OWNY.MYTABLE.COLA
FROM OWNX.MYTABLE CORZ, OWNY.MYTABLE
```

Table designators in the FROM clause are established as follows:

- A name that follows a table, view, nested table expression or table function is both a correlation name and a table designator. Thus, CORZ is a table designator. CORZ is used to qualify the first column name in the select list.
- An exposed table or view name is a table designator. Thus, OWNY.MYTABLE is a table designator. OWNY.MYTABLE is used to qualify the second column name in the select list.

Each table designator should be unique within a particular FROM clause to avoid the possibility of ambiguous references to columns.

## Column Names

### Avoiding Undefined or Ambiguous References

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:

- 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 designated 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.
- The column name is qualified by a table designator, but the table designated is not unique in the FROM clause and both occurrences of the designated table include the column. The reference is ambiguous.
- The column name is in a nested table expression which is not preceded by the TABLE keyword or in a table function or nested table expression that is the right operand of a right outer join or a full outer join and the column name does not refer to a column of a *table-reference* within the nested table expression's fullselect. The reference is undefined.

Avoid ambiguous references by qualifying a column name with a uniquely defined table designator. If the column is contained in several object tables with different names, the table names can be used as designators. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name may be used. However, the qualifier used and the table used must be the same after fully qualifying the table name or view name and the table designator.

1. If the authorization ID of the statement is CORPDATA:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
FROM EMPLOYEE
```

is a valid statement.

2. If the authorization ID of the statement is REGION:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
FROM EMPLOYEE * incorrect *
```

is invalid, because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE.

### Column Name Qualifiers in Correlated References

A *fullselect* is a form of a query that may be used as a component of various SQL statements. See Chapter 5, "Queries" on page 315 for more information on fullselects. A fullselect used within a search condition of any statement is called a *subquery*. A fullselect used to retrieve a single value as an expression within a statement is called a

## Column Names

*scalar fullselect* or *scalar subquery*. A fullselect used in the FROM clause of a query is called a *nested table expression*. Subqueries in search conditions, scalar subqueries and nested table expressions are referred to as subqueries through the remainder of this topic.

A subquery may include subqueries of its own, and these may, in turn, include subqueries. Thus an SQL statement may 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 contains one or more table designators. A subquery can reference not only the columns of the tables identified at its own level in the hierarchy, but also the columns of the tables identified previously in the hierarchy, back 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*.

For compatibility with existing standards for SQL, both qualified and unqualified column names are allowed as correlated references. However, it is good practice to qualify all column references used in subqueries; otherwise, identical column names may lead to unintended results. For example, if a table in a hierarchy is altered to contain the same column name as the correlated reference and the statement is prepared again, the reference will apply to the altered table.

When a column name in a subquery is qualified, each level of the hierarchy is searched, starting at the same subquery as the qualified column name appears and continuing to the higher levels of the hierarchy until a table designator that matches the qualifier is found. Once found, it is verified that the table contains the given column. If the table is found at a higher level than the level containing column name, then it is a correlated reference to the level where the table designator was found. A nested table expression must be preceded with the optional TABLE keyword in order to search the hierarchy above the fullselect of the nested table expression.

When the column name in a subquery is not qualified, the tables referenced at each level of the hierarchy are searched, starting at the same subquery where the column name appears and continuing to higher levels of the hierarchy, until a match for the column name is found. If the column is found in a table at a higher level than the level containing column name, then it is a correlated reference to the level where the table containing the column was found. If the column name is found in more than one table at a particular level, the reference is ambiguous and considered an error.

In either case, T, used in the following example, refers to the table designator that contains column C. A column name, T.C (where T represents either an implicit or an explicit qualifier), is a correlated reference if, and only if, these conditions are met:

- T.C is used in an expression of a subquery.
- T does not designate a table used in the from clause of the subquery.
- T designates a table used at a higher level of the hierarchy that contains the subquery.

## Column Names

Since the same table or view can be identified at many levels, unique correlation names are recommended as table designators. If T is used to designate a table at more than one level (T is the table name itself or is a duplicate correlation name), T.C refers to the level where T is used that most directly contains the subquery that includes T.C. If a correlation to a higher level is needed, a unique correlation name must be used.

The correlated reference T.C identifies a value of C in a row or group of T to which two search conditions are being applied: condition 1 in the subquery, and condition 2 at some higher level. If condition 2 is used in a WHERE clause, the subquery is evaluated for each row to which condition 2 is applied. If condition 2 is used in a HAVING clause, the subquery is evaluated for each group to which condition 2 is applied. (For another discussion of the evaluation of subqueries, see the descriptions of the WHERE and HAVING clauses in Chapter 5, "Queries" on page 315.)

For example, in the following statement, the correlated reference X.WORKDEPT (in the last line) refers to the value of WORKDEPT in table EMPLOYEE at the level of the first FROM clause. (That clause establishes X as a correlation name for EMPLOYEE.) The statement lists employees who make less than the average salary for their department.

```
SELECT EMPNO, LASTNAME, WORKDEPT
FROM EMPLOYEE X
WHERE SALARY < (SELECT AVG(SALARY)
                FROM EMPLOYEE
                WHERE WORKDEPT = X.WORKDEPT)
```

The next example uses THIS as a correlation name. The statement deletes rows for departments that have no employees.

```
DELETE FROM DEPARTMENT THIS
WHERE NOT EXISTS(SELECT *
                 FROM EMPLOYEE
                 WHERE WORKDEPT = THIS.DEPTNO)
```

---

### References to Host Variables

A *host variable* is either:

- A variable in a host language such as a C variable, a C++ variable, a COBOL data item, a FORTRAN variable, or a REXX variable

or:

- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions

that is referenced in an SQL statement. Host variables are either directly defined by statements in the host language or are indirectly defined using SQL extensions.

A host variable in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

All host variables used in an SQL statement must be declared in an SQL DECLARE section in all host languages except REXX (see the *Embedded SQL Programming Guide* for more information on declaring host variables for SQL statements in application programs). No variables may be declared outside an SQL DECLARE section with names identical to variables declared inside an SQL DECLARE section. An SQL DECLARE section begins with BEGIN DECLARE SECTION and ends with END DECLARE SECTION.

The meta-variable *host-variable*, as used in the syntax diagrams, shows a reference to a host variable. A host-variable in the VALUES INTO clause or the INTO clause of a FETCH or a SELECT INTO statement, identifies a host variable to which a value from a column of a row or an expression is assigned. In all other contexts a host-variable specifies a value to be passed to the database manager from the application program.

### Host Variables in Dynamic SQL

In dynamic SQL statements, parameter markers are used instead of host variables. A parameter marker is a question mark (?) representing 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 example shows a static SQL statement using host variables:

```
INSERT INTO DEPARTMENT
VALUES (:hv_deptno, :hv_deptname, :hv_mgrno, :hv_admrdept)
```

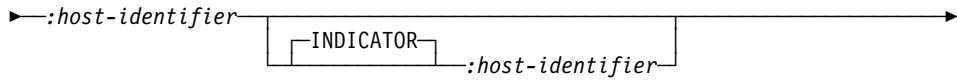
This example shows a dynamic SQL statement using parameter markers:

```
INSERT INTO DEPARTMENT VALUES (?, ?, ?, ?)
```

For more information on parameter markers, see “Parameter Markers” in “PREPARE” on page 673.

The meta-variable *host-variable* in syntax diagrams can generally be expanded to:

## References to Host Variables



Each *host-identifier* must be declared in the source program. The variable designated by the second host-identifier must have a data type of small integer.

The first host-identifier designates the *main variable*. Depending on the operation, it either provides a value to the database manager or is provided a value from the database manager. An input host variable provides a value in the runtime application code page. An output host variable is provided a value that, if necessary, is converted to the runtime application code page when the data is copied to the output application variable. A given host variable can serve as both an input and an output variable in the same program.

The second host-identifier designates its *indicator variable*. The purposes of the indicator variable are to:

- Specify the null value. A negative value of the indicator variable specifies the null value. A value of -2 indicates a numeric conversion or arithmetic expression error occurred in deriving the result.
- Record the original length of a truncated string (if the source of the value is not a large object type)
- Record the seconds portion of a time if the time is truncated on assignment to a host variable.

For example, if `:HV1:HV2` is used to specify an insert or update value, and if HV2 is negative, the value specified is the null value. If HV2 is not negative the value specified is the value of HV1.

Similarly, if `:HV1:HV2` is specified in a VALUES INTO clause or in a FETCH or SELECT INTO statement, and if the value returned is null, HV1 is not changed and HV2 is set to a negative value.<sup>24</sup> If the value returned is not null, that value is assigned to HV1 and HV2 is set to zero (unless the assignment to HV1 requires string truncation of a non-LOB string; in which case HV2 is set to the original length of the string). If an assignment requires truncation of the seconds part of a time, HV2 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 reference `:HV1` is always the value of HV1, and null values cannot be assigned to the variable. Thus, this form should not be used in an INTO clause unless the corresponding column cannot contain null values. If

---

<sup>24</sup> If the database is configured with DFT\_SQLMATHWARN yes (or was during binding of a static SQL statement), then HV2 could be -2. If HV2 is -2, then a value for HV1 could not be returned because of an error converting to the numeric type of HV1 or an error evaluating an arithmetic expression that is used to determine the value for HV1. When accessing a database with a client version earlier than DB2 Universal Database Version 5, HV2 will be -1 for arithmetic exceptions.



## References to Host Variables

this form is used and the column contains nulls, the database manager will generate an error at run time.

An SQL statement that references host variables must be within the scope of the declaration of those host variables. For host variables referenced in the SELECT statement of a cursor, that rule applies to the OPEN statement rather than to the DECLARE CURSOR statement.

### Example

Using the PROJECT table, set the host variable PNAME (varchar(26)) to the project name (PROJNAME), the host variable STAFF (dec(5,2)) to the mean staffing level (PRSTAFF), and the host variable MAJPROJ (char(6)) to the major project (MAJPROJ) for project (PROJNO) 'IF1000'. Columns PRSTAFF and MAJPROJ may contain null values, so provide indicator variables STAFF\_IND (smallint) and MAJPROJ\_IND (smallint).

```
SELECT PROJNAME, PRSTAFF, MAJPROJ
   INTO :PNAME, :STAFF :STAFF_IND, :MAJPROJ :MAJPROJ_IND
   FROM PROJECT
   WHERE PROJNO = 'IF1000'
```

**MBCS Considerations:** Whether multi-byte characters can be used in a host variable name depends on the host language.

## References to BLOB, CLOB, and DBCLOB Host Variables

Regular BLOB, CLOB, and DBCLOB variables, LOB locator variables (see “References to Locator Variables” on page 108), and LOB file reference variables (see “References to BLOB, CLOB, and DBCLOB File Reference Variables” on page 108) 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, a locator variable, or a file reference variable. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

It is sometimes possible to define a large enough variable to hold an entire large object value. If this is true and if there is no performance benefit to be gained by deferred transfer of data from the server, a locator is not needed. However, since host language or space restrictions will often dictate against storing an entire large object in temporary storage at one time and/or because of performance benefit, a large object may be referenced via a locator and portions of that object may be selected into or updated from host variables that contain only a portion of the large object at one time.

As with all other host variables, a large object locator variable may have an associated indicator variable. Indicator variables for large object locator host 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 point to a null value.

## References to Host Variables

### References to Locator Variables

A *locator variable* is a host variable that contains the locator representing a LOB value on the application server. (See “Manipulating Large Objects (LOBs) with Locators” on page 55 for information on how locators can be used to manipulate LOB values.)

A locator variable in an SQL statement must identify a locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement.

The term locator variable, as used in the syntax diagrams, shows a reference to a locator variable. The meta-variable *locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

When the indicator variable associated with a locator is null, the value of the referenced LOB is null.

If a locator-variable that does not currently represent any value is referenced, an error is raised (SQLSTATE 0F001).

At transaction commit, or any transaction termination, all locators acquired by that transaction are released.

### References to BLOB, CLOB, and DBCLOB File Reference Variables

BLOB, CLOB, and DBCLOB file reference variables are used for direct file input and output for LOBs, and can be defined in all host languages. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

A file reference variable represents (rather than contains) the file, just as a LOB locator represents, rather than contains, the LOB bytes. Database queries, updates and inserts may use file reference variables to store or to retrieve single column values.

A file reference variable has the following properties:

Data Type	BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared.
Direction	This must be specified by the application program at run time (as part of the File Options value -see below). The direction is one of: <ul style="list-style-type: none"><li>• Input (used as a source of data on an EXECUTE statement, an OPEN statement, an UPDATE statement, an INSERT statement, or a DELETE statement).</li><li>• Output (used as the target of data on a FETCH statement or a SELECT INTO statement).</li></ul>

## References to Host Variables

File name	<p>This must be specified by the application program at run time. It is one of:</p> <ul style="list-style-type: none"><li>• The complete path name of the file (which is advised).</li><li>• A relative file name. If a relative file name is provided, it is appended to the current path of the client process.</li></ul> <p>Within an application, a file should only be referenced in one file reference variable.</p>
File Name Length	<p>This must be specified by the application program at run time. It is the length of the file name (in bytes).</p>
File Options	<p>An application must assign one of a number of options to a file reference variable before it makes use of that variable. Options are set by an INTEGER value in a field in the file reference variable structure. One of the following values must be specified for each file reference variable:</p> <ul style="list-style-type: none"><li>• Input (from client to server) <b>SQL_FILE_READ</b><sup>25</sup> This is a regular file that can be opened, read and closed.</li><li>• Output (from server to client) <b>SQL_FILE_CREATE</b><sup>26</sup> Create a new file. If the file already exists, it is an error. <b>SQL_FILE_OVERWRITE (Overwrite)</b><sup>27</sup> If an existing file with the specified name exists, it is overwritten; otherwise a new file is created. <b>SQL_FILE_APPEND</b><sup>28</sup> If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created.</li></ul> <p><b>Data Length</b> This is unused on input. On output, the implementation sets the data length to the length of the new data written to the file. The length is in bytes.</p>

As with all other host variables, a file reference variable may have an associated indicator variable.

### Example of an Output File Reference Variable (in C)

Given a declare section is coded as:

---

<sup>25</sup> SQL-FILE-READ in COBOL, sql\_file\_read in FORTRAN, READ in REXX.

<sup>26</sup> SQL-FILE-CREATE in COBOL, sql\_file\_create in FORTRAN, CREATE in REXX.

<sup>27</sup> SQL-FILE-OVERWRITE in COBOL, sql\_file\_overwrite in FORTRAN, OVERWRITE in REXX.

<sup>28</sup> SQL-FILE-APPEND in COBOL, sql\_file\_append in FORTRAN, APPEND in REXX.

## Functions

```
EXEC SQL BEGIN DECLARE SECTION
SQL TYPE IS CLOB_FILE hv_text_file;
char hv_patent_title[64];
EXEC SQL END DECLARE SECTION
```

Following preprocessing this would be:

```
EXEC SQL BEGIN DECLARE SECTION
/* SQL TYPE IS CLOB_FILE hv_text_file; */
struct {
    unsigned long name_length; // File Name Length
    unsigned long data_length; // Data Length
    unsigned long file_options; // File Options
    char name[255]; // File Name
} hv_text_file;
char hv_patent_title[64];
EXEC SQL END DECLARE SECTION
```

Then, the following code can be used to select from a CLOB column in the database into a new file referenced by :hv\_text\_file.

```
strcpy(hv_text_file.name, "/u/gainer/papers/sigmod.94");
hv_text_file.name_length = strlen("/u/gainer/papers/sigmod.94");
hv_text_file.file_options = SQL_FILE_CREATE;

EXEC SQL SELECT content INTO :hv_text_file from papers
WHERE TITLE = 'The Relational Theory behind Juggling';
```

### Example of an Input File Reference Variable (in C)

Given the same declare section as above, the following code can be used to insert the data from a regular file referenced by :hv\_text\_file into a CLOB column.

```
strcpy(hv_text_file.name, "/u/gainer/patents/chips.13");
hv_text_file.name_length = strlen("/u/gainer/patents/chips.13");
hv_text_file.file_options = SQL_FILE_READ;
strcpy(:hv_patent_title, "A Method for Pipelining Chip Consumption");

EXEC SQL INSERT INTO patents( title, text )
VALUES(:hv_patent_title, :hv_text_file);
```

---

## Functions

A *database function* is a relationship between a set of input data values and a set of result values. For example, the `TIMESTAMP` function can be passed input data values of type `DATE` and `TIME` and the result is a `TIMESTAMP`. Functions can either be built-in or user-defined.

- *Built-in functions* are provided with the database manager providing a single result value and are identified as part of the `SYSIBM` schema. Examples of such functions include column functions such as `AVG`, operator functions such as `+`, casting functions such as `DECIMAL`, and others such as `SUBSTR`.

## Functions

- *User-defined functions* are functions that are registered to a database in SYSCAT.FUNCTIONS (using the CREATE FUNCTION statement). User-defined functions are never part of the SYSIBM schema. One such set of functions is provided with the database manager in a schema called SYSFUN.

With user-defined functions, DB2 allows users and application developers to extend the function of the database system by adding function definitions provided by users or third party vendors to be applied in the database engine itself. This allows higher performance than retrieving rows from the database and applying those functions on the retrieved data to further qualify or to perform data reduction. Extending database functions also lets the database exploit the same functions in the engine that an application uses, provides more synergy between application and database, and contributes to higher productivity for application developers because it is more object-oriented.

A complete list of functions in the SYSIBM and SYSFUN schemas is documented in Table 13 on page 156.

A user-defined function can be *external* or *sourced*. An *external function* is defined to the database with a reference to an object code library and a function within that library that will be executed when the function is invoked. External functions can not be column functions. A *sourced function* is defined to the database with a reference to another built-in or user-defined function that is already known to the database. Sourced functions can be scalar functions or column functions. They are very useful for supporting the use of existing functions with user-defined types.

Each user-defined function is also categorized as a *scalar*, *column* or *table* function. A *scalar function* is one which returns a single-valued answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can be either external or sourced. A *column function* is one which conceptually is passed a set of like values (a column) and returns a single-valued answer from this set. These are also sometimes called *aggregating functions* in DB2. An example of a column function is the built-in function AVG(). An external column UDF cannot be defined to DB2, but a column UDF which is sourced upon one of the built-in column functions can be defined. This is useful for distinct types. For example if there is a distinct type SHOESIZE defined with base type INTEGER, a UDF AVG(SHOESIZE) which is sourced on the built-in function AVG(INTEGER) could be defined, and it would be a column function. A *table function* is a function which returns a table to the SQL statement which references it, and it may only be referenced in the FROM clause of a SELECT. Such a function can be used to apply SQL language processing power to data which is not DB2 data, or to convert such data into a DB2 table. It could, for example, take a file and convert it to a table, sample data from the world-wide web and tabularize it, or access a Lotus Notes database and return information about mail messages, such as the date, sender, and the text of the message. This information can be joined with other tables in the database. A table function can only be an external function (a table function cannot be a sourced function).

A function is identified by its schema, a function name, the number of parameters and the data types of its parameters. This is called a *function signature* which must be unique within the database. There can be more than one function with the same name

## Functions

in a schema provided that the number of parameters or the data types of the parameters are different. A function name for which there are multiple function instances is called an *overloaded* function. A function name can be overloaded within a schema, in which case there is more than one function by that name in the schema (which of necessity have different parameter types). A function name can also be overloaded in a SQL path, in which case there is more than one function by that name in the path, and these functions do not necessarily have different parameter types.

A function can be invoked by referring in an allowable context to the qualified name (schema and function name) followed by the list of arguments enclosed in parentheses. A function can also be invoked without the schema name resulting in a choice of possible functions in different schemas with the same or acceptable parameters. In this case, the *SQL path* is used to assist in *function resolution*. The function path is a list of schemas that are searched to identify a function with the same name, number of parameters and acceptable data types. For static SQL statements, SQL path is specified using the FUNCSPATH bind option (see *Command Reference* for details). For dynamic SQL statements, SQL path is the value of the CURRENT FUNCTION PATH special register (see “CURRENT PATH” on page 94).

### Function Resolution

Given a function invocation, the database manager must decide which of the possible functions with the same name is the “best” fit. This includes resolving functions from the built-in and user-defined functions.

An *argument* is a value passed to a function upon invocation. When a function is invoked in SQL, it is passed a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a function. When a function is defined to the database, either internally (the built-in functions) or by a user (user-defined functions), its parameters (zero or more) are specified, the order of their definitions defining their positions and thus their semantics. Therefore, every parameter is a particular positional input of a function. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

The database manager uses the name of the function given in the invocation, the number and data types of the arguments, all the functions with the same name in the SQL path, and the data types of their corresponding parameters as the basis for deciding whether or not to select a function. The following are the possible outcomes of the decision process:

1. A particular function is deemed to be the best fit. For example, given the functions named RISK in the schema TEST with signatures defined as:

```
TEST.RISK(INTEGER)
TEST.RISK(DOUBLE)
```

a SQL path including the TEST schema and the following function reference (where DB is a DOUBLE column):

```
SELECT ... RISK(DB) ...
```

then, the second RISK will be chosen.

The following function reference (where SI is a SMALLINT column):

```
SELECT ... RISK(SI) ...
```

would choose the first RISK, since SMALLINT can be promoted to INTEGER and is a better match than DOUBLE which is further down the precedence list (as shown in Table 3 on page 66).

2. No function is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

```
SELECT ... RISK(C) ...
```

the argument is inconsistent with the parameter of both RISK functions.

3. A particular function is selected based on the SQL path and the number and data types of the arguments passed on invocation. For example, given functions named RANDOM with signatures defined as:

```
TEST.RANDOM(INTEGER)  
PROD.RANDOM(INTEGER)
```

and a SQL path of:

```
"TEST", "PROD"
```

Then the following function reference:

```
SELECT ... RANDOM(432) ...
```

will choose TEST.RANDOM since both RANDOM functions are equally good matches (exact matches in this particular case) and both schemas are in the path, but TEST precedes PROD in the SQL path.

### Method of Choosing the Best Fit

A comparison of the data types of the arguments with the defined data types of the parameters of the functions under consideration forms the basis for the decision of which function in a group of like-named functions is the "best fit". Note that the data type of the result of the function or the type of function (column, scalar, or table) under consideration **does not** enter into this determination.

Function resolution is done using the steps that follow.

1. First, find all functions from the catalog (SYSCAT.FUNCTIONS) and built-in functions such that all of the following are true:
  - a. For invocations where the schema name was specified (i.e. qualified references), then the schema name and the function name match the invocation name.
  - b. For invocations where the schema name was not specified (i.e. unqualified references), then the function name matches the invocation name and has a schema name that matches one of the schemas in the SQL path.
  - c. The number of defined parameters matches the invocation.

## Functions

- d. Each invocation argument matches the function's corresponding defined parameter in data type, or is "promotable" to it (see "Promotion of Data Types" on page 66).
2. Next, consider each argument of the function invocation, from left to right. For each argument, eliminate all functions that are not the *best match* for that argument. The *best match* for a given argument is the first data type appearing in the precedence list corresponding to the argument data type in Table 3 on page 66 for which there exists a function with a parameter of that data type. Lengths, precisions, scales and the "FOR BIT DATA" attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.
3. If more than one candidate function remains after Step 2, then it has to be the case (the way the algorithm works) that all the remaining candidate functions have identical signatures but are in different schemas. Choose the function whose schema is earliest in the user's SQL path.
4. If there are no candidate functions remaining after step 2, an error is returned (SQLSTATE 42884).

### Function Path Considerations for Built-in Functions

Built-in functions reside in a special schema called SYSIBM. Additional functions are available in the SYSFUN schema which are not considered built-in functions since they are developed as user-defined functions and have no special processing considerations. Users can not define additional functions in SYSIBM or SYSFUN schemas (or in any schema whose name begins with the letters "SYS").

As already stated, the built-in functions participate in the function resolution process exactly as do the user-defined functions. One difference between built-in and user-defined functions, from a function resolution perspective, is that the built-in function must always be considered by function resolution. Therefore, omission of SYSIBM from the path results in an assumption for function and data type resolution that SYSIBM is the first schema on the path.

For example, if a user's SQL path is defined as:

```
"SHAREFUN", "SYSIBM", "SYSFUN"
```

and there is a LENGTH function defined in schema SHAREFUN with the same number and types of arguments as SYSIBM.LENGTH, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SHAREFUN.LENGTH. However, if the user's SQL path is defined as:

```
"SHAREFUN", "SYSFUN"
```

and the same SHAREFUN.LENGTH function exists, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SYSIBM.LENGTH since SYSIBM is implicitly first in the path because it was not specified. It is possible to minimize potential problems in this area by:

- never using the names of built-in functions for user-defined functions, or



- qualifying any references to these functions, if for some reason it is deemed necessary to create a user-defined function with the same name as a built-in function.

### Example of Function Resolution

The following is an example of successful function resolution.

There are six FOO functions, in two different schemas, registered as (note that not all required keywords appear):

```
CREATE FUNCTION AUGUSTUS.FOO (CHAR(5), INT, DOUBLE) SPECIFIC FOO_1 ...
CREATE FUNCTION AUGUSTUS.FOO (INT, INT, DOUBLE) SPECIFIC FOO_2 ...
CREATE FUNCTION AUGUSTUS.FOO (INT, INT, DOUBLE, INT) SPECIFIC FOO_3 ...
CREATE FUNCTION JULIUS.FOO (INT, DOUBLE, DOUBLE) SPECIFIC FOO_4 ...
CREATE FUNCTION JULIUS.FOO (INT, INT, DOUBLE) SPECIFIC FOO_5 ...
CREATE FUNCTION JULIUS.FOO (SMALLINT, INT, DOUBLE) SPECIFIC FOO_6 ...
CREATE FUNCTION NERO.FOO (INT, INT, DEC(7,2)) SPECIFIC FOO_7 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and D is a DECIMAL column):

```
SELECT ... FOO(I1, I2, D) ...
```

Assume that the application making this reference has a SQL path established as:

```
"JULIUS", "AUGUSTUS", "CAESAR"
```

Following through the algorithm...

FOO\_7 is eliminated as a candidate, because the schema "NERO" is not included in the SQL path.

FOO\_3 is eliminated as a candidate, because it has the wrong number of parameters. FOO\_1 and FOO\_6 are eliminated because in both cases the first argument cannot be promoted to the data type of the first parameter.

Because there is more than one candidate remaining, the arguments are then considered in order.

For the first argument, all remaining functions — FOO\_2, FOO\_4 and FOO\_5 are an exact match with the argument type. No functions can be eliminated from consideration, therefore the next argument must be examined.

For this second argument, FOO\_2 and FOO\_5 are exact matches while FOO\_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation between FOO\_2 and FOO\_5.

On the third and last argument, neither FOO\_2 nor FOO\_5 match the argument type exactly, but both are equally good.

There are two functions remaining, FOO\_2 and FOO\_5, with identical parameter signatures. The final tie-breaker is to see which function's schema comes first in the SQL path, and on this basis FOO\_5 is finally chosen.

### Function Invocation

Once the function is selected, there are still possible reasons why the use of the function may not be permitted. Each function is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the function

## Functions

is invoked, an error will occur. For example, given functions named STEP defined, this time, with different data types as the result:

```
STEP(SMALLINT) returns CHAR(5)
STEP(DOUBLE)   returns INTEGER
```

and the following function reference (where S is a SMALLINT column):

```
SELECT ... 3 + STEP(S) ...
```

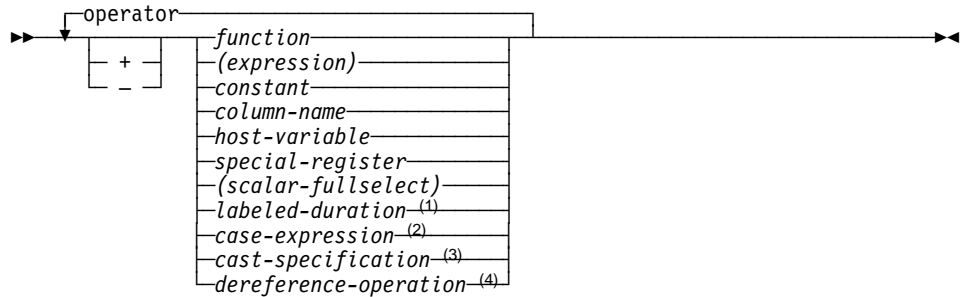
then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

A couple of other examples where this can happen are as follows, both of which will result in an error on the statement:

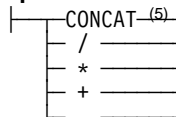
1. The function was referenced in a FROM clause, but the function selected by the function resolution step was a scalar or column function.
2. The reverse case, where the context calls for a scalar or column function, and function resolution selects a table function.

In cases where the arguments of the function invocation were not an exact match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns (see “Assignments and Comparisons” on page 70). This includes the case where precision, scale, or length differs between the argument and the parameter.

Expressions



operator:



Notes:

- 1 See “Labeled Durations” on page 123 for more information.
- 2 See “CASE Expressions” on page 129 for more information.
- 3 See “CAST Specifications” on page 131 for more information.
- 4 See “Dereference Operations” on page 134 for more information.
- 5 || may be used as a synonym for CONCAT.

An *expression* specifies a value.

A *scalar fullselect* as supported in an expression is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If the select list element is an expression that is simply a column name or a dereference operation, the result column name is based on the name of the column. See “fullselect” on page 350 for more information.

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

The concatenation operator (CONCAT) links two string operands to form a *string expression*.

The operands of concatenation must be compatible strings. Note that a binary string cannot be concatenated with a character string, including character strings defined as

## Expressions

FOR BIT DATA (SQLSTATE 42884). For more information on compatibility, refer to the compatibility matrix on page Table 5 on page 71.

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. Note that no check is made for improperly formed mixed data when doing concatenation.

The length of the result is the sum of the lengths of the operands.

The data type and length attribute of the result is determined from that of the operands as shown in the following table:

Operands	Combined Length Attributes	Result
CHAR(A) CHAR(B)	<255	CHAR(A+B)
CHAR(A) CHAR(B)	>254	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	>4000	LONG VARCHAR
CHAR(A) LONG VARCHAR	-	LONG VARCHAR
VARCHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
VARCHAR(A) VARCHAR(B)	>4000	LONG VARCHAR
VARCHAR(A) LONG VARCHAR	-	LONG VARCHAR
LONG VARCHAR LONG VARCHAR	-	LONG VARCHAR
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) VARCHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) LONG VARCHAR	-	CLOB(MIN(A+32K, 2G))
CLOB(A) CLOB(B)	-	CLOB(MIN(A+B, 2G))
GRAPHIC(A) GRAPHIC(B)	<128	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>127	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC
GRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
VARGRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
VARGRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC
VARGRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
LONG VARGRAPHIC LONG VARGRAPHIC	-	LONG VARGRAPHIC

Operands	Combined Length Attributes	Result
DBCLOB(A) GRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) VARGRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) LONG VARGRAPHIC	-	DBCLOB(MIN(A+16K, 1G))
DBCLOB(A) DBCLOB(B)	-	DBCLOB(MIN(A+B, 1G))
BLOB(A) BLOB(B)	-	BLOB(MIN(A+B, 2G))

Note that, for compatibility with previous versions, there is no automatic escalation of results involving LONG data types to LOB data types. For example, concatenation of a CHAR(200) value and a completely full LONG VARCHAR value would result in an error rather than in a promotion to a CLOB data type.

The code page of the result is considered a derived code page and is determined by the code page of its operands as explained in “Rules for String Conversions” on page 85.

One operand may be a parameter marker. If a parameter marker is used, then the data type and length attributes of that operand are considered to be the same as those for the non-parameter marker operand. The order of operations must be considered to determine these attributes in cases with nested concatenation.

*Example 1:* If FIRSTNAME is Pierre and LASTNAME is Fermat, then the following :

```
FIRSTNAME CONCAT ' ' CONCAT LASTNAME
```

returns the value Pierre Fermat

*Example 2:* Given:

COLA defined as VARCHAR(5) with value 'AA'

:host\_var defined as a character host variable with length 5 and value 'BB'

COLC defined as CHAR(5) with value 'CC'

COLD defined as CHAR(5) with value 'DDDDD'

The value of: COLA **CONCAT** :host\_var **CONCAT** COLC **CONCAT** COLD is:

```
'AABB CC DDDDD'
```

The data type is VARCHAR, the length attribute is 17 and the result code page is the database code page.

*Example 3:* Given:

COLA is defined as CHAR(10)

## Expressions

COLB is defined as VARCHAR(5)

The parameter marker in the expression:

```
COLA CONCAT COLB CONCAT ?
```

is considered VARCHAR(15) since COLA CONCAT COLB is evaluated first giving a result which is the first operand of the second CONCAT operation.

### User-defined Types

A user-defined type cannot be used with the concatenation operator even if its source data type is character. To concatenate, create a function with the CONCAT operator as its source. For example, if there were distinct types TITLE and TITLE\_DESCRIPTION, both of which had VARCHAR(25) data types, then the following user-defined function, ATTACH, could be used to concatenate them.

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternately, the concatenation operator could be overloaded using a user-defined function to add the new data types.

```
CREATE FUNCTION CONCAT (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 value derived from the application of the operators to the values of the operands.

If any operand can be null, or the database is configured with DFT\_SQLMATHWARN set to yes, the result can be null.

If any operand has the null value, the result of the expression is the null value.

Arithmetic operators must not be applied to character strings. For example, USER+2 is invalid.

The prefix operator + (unary plus) does not change its operand. The prefix operator - (unary minus) reverses the sign of a nonzero operand; and if the data type of A is small integer, then 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. These operators can also be treated as functions. Thus, the expression "+(a,b) is equivalent to the expression a+b. "operator" function.

### Arithmetic Errors

If an arithmetic error such as zero divide or a numeric overflow occurs during the processing of an expression, an error is returned and the SQL statement processing the expression fails with an error (SQLSTATE 22003 or 22012).

A database can be configured (using `DFT_SQLMATHWARN` set to yes) so that arithmetic errors return a null value for the expression, issue a warning (SQLSTATE 01519 or 01564), and proceed with processing of the SQL statement. When arithmetic errors are treated as nulls, there are implications on the results of SQL statements. The following are some examples of these implications.

- An arithmetic error that occurs in the expression that is the argument of a column function causes the row to be ignored in the determining the result of the column function. If the arithmetic error was an overflow, this may significantly impact the result values.
- An arithmetic error that occurs in the expression of a predicate in a WHERE clause can cause rows to not be included in the result.
- An arithmetic error that occurs in the expression of a predicate in a check constraint results in the update or insert proceeding since the constraint is not false.

If these types of impacts are not acceptable, additional steps should be taken to handle the arithmetic error to produce acceptable results. Some examples are:

- add a case expression to check for zero divide and set the desired value for such a situation
- add additional predicates to handle nulls (like a check constraint on not nullable columns could become:

```
check (c1*c2 is not null and c1*c2>5000)
```

to cause the constraint to be violated on an overflow).

### 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* unless either (or both) operand is a big integer, in which case the result is a big integer. Any remainder of division is lost. The result of an integer arithmetic operation (including unary minus) must be within the range of the result type.

### Integer and Decimal Operands

If one operand is an integer and the other is a decimal, the operation is performed in decimal using a temporary copy of the integer which has been converted to a decimal number with precision  $p$  and scale 0.  $p$  is 19 for a big integer, 11 for a large integer and 5 for a small integer.

### 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 are dependent on the operation and the precision and scale of the operands. If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands. The copy of the shorter operand is extended with trailing zeros so that its fractional part has the same number of digits as the longer operand.

## Expressions

The result of a decimal operation must not have a precision greater than 31. The result of decimal addition, subtraction, and multiplication is derived from a temporary result which may have a precision greater than 31. If the precision of the temporary result is not greater than 31, the final result is the same as the temporary result.

### Decimal Arithmetic in SQL

The following formulas define the precision and scale of the result of decimal operations in SQL. 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.

#### Addition and Subtraction

The precision is  $\min(31, \max(p-s, p'-s') + \max(s, s') + 1)$ . The scale of the result of addition and subtraction is  $\max(s, s')$ .

#### Multiplication

The precision of the result of multiplication is  $\min(31, p+p')$  and the scale is  $\min(31, s+s')$ .

#### Division

The precision of the result of division is 31. The scale is  $31-p+s-s'$ . The scale must not be negative.

### Floating-Point Operands

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point, the operands having first been converted to double-precision floating-point numbers, if necessary. 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 which 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 which has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

### User-defined Types as Operands

A user-defined type cannot be used with arithmetic operators even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, then the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE (INCOME, EXPENSES)  
RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a user-defined function to subtract the new data types.



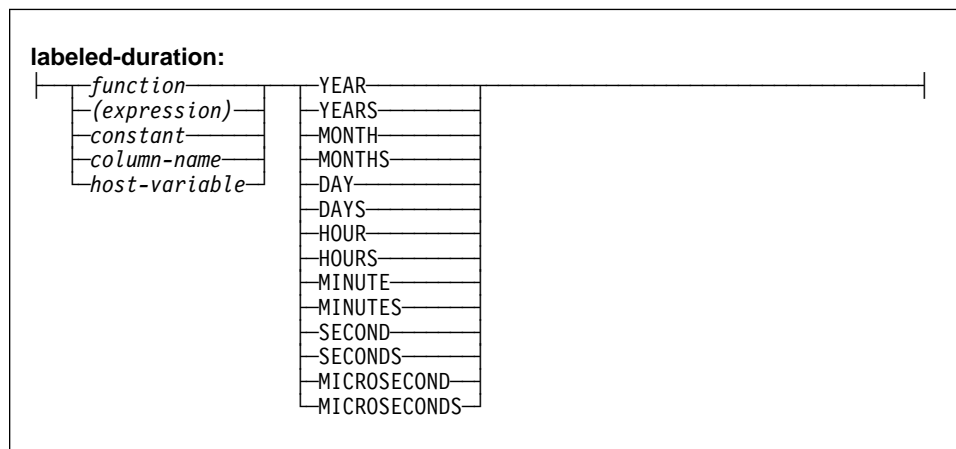
```
CREATE FUNCTION "-" (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

## Datetime Operations and Durations

Datetime values can be incremented, decremented, and subtracted. These operations may involve decimal numbers called *durations*. Following is a definition of durations and a specification of the rules for datetime arithmetic.

A duration is a number representing an interval of time. There are four types of durations:

### Labeled Durations



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: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS<sup>29</sup>. The number specified is converted as if it were assigned to a DECIMAL(15,0) number. A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME, or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

### Date Duration

A *date duration* represents a number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format

<sup>29</sup> Note that the singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

<sup>30</sup> The period in the format indicates a DECIMAL data type.

## Expressions

*yyyymmdd.*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days.<sup>30</sup> The result of subtracting one date value from another, as in the expression `HIREDATE - BRTHDATE`, 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.<sup>30</sup> 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 *yyyymmddhhmmss.zzzzzz*, where *yyyy*, *mm*, *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.

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.

### 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  $\text{result} = \text{DATE1} - \text{DATE2}$ .

If  $\text{DAY}(\text{DATE2}) \leq \text{DAY}(\text{DATE1})$   
 then  $\text{DAY}(\text{RESULT}) = \text{DAY}(\text{DATE1}) - \text{DAY}(\text{DATE2})$ .

If  $\text{DAY}(\text{DATE2}) > \text{DAY}(\text{DATE1})$   
 then  $\text{DAY}(\text{RESULT}) = N + \text{DAY}(\text{DATE1}) - \text{DAY}(\text{DATE2})$   
 where  $N =$  the last day of  $\text{MONTH}(\text{DATE2})$ .  
 $\text{MONTH}(\text{DATE2})$  is then incremented by 1.

If  $\text{MONTH}(\text{DATE2}) \leq \text{MONTH}(\text{DATE1})$   
 then  $\text{MONTH}(\text{RESULT}) = \text{MONTH}(\text{DATE1}) - \text{MONTH}(\text{DATE2})$ .

If  $\text{MONTH}(\text{DATE2}) > \text{MONTH}(\text{DATE1})$   
 then  $\text{MONTH}(\text{RESULT}) = 12 + \text{MONTH}(\text{DATE1}) - \text{MONTH}(\text{DATE2})$ .  
 $\text{YEAR}(\text{DATE2})$  is then incremented by 1.

$\text{YEAR}(\text{RESULT}) = \text{YEAR}(\text{DATE1}) - \text{YEAR}(\text{DATE2})$ .

For example, the result of  $\text{DATE}('3/15/2000') - '12/31/1999'$  is 00000215. (or, a duration of 0 years, 2 months, and 15 days).

**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. In this case, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

## Expressions

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 a warning indicator in the SQLCA is set 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, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

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:

$DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS.$

When adding durations to dates, adding one month to a given date gives the same date one month later unless that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

**Note:** 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.

### 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  $result = TIME1 - TIME2$ .

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).
        MINUTE(TIME2) is then incremented by 1.
If MINUTE(TIME2) <= MINUTE(TIME1)
    then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).
If MINUTE(TIME1) > MINUTE(TIME1)
    then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
        HOUR(TIME2) is then 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).

**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. 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 will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

Time durations, whether positive or negative, also can 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. `TIME1 + X`, where “X” is a `DECIMAL(6,0)` number, is equivalent to the expression:

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
```

**Note:** Although the time `'24:00:00'` is accepted as a valid time, it is never returned as the result of time addition or subtraction, even if the duration operand is zero (e.g. `time('24:00:00')±0 seconds = '00:00:00'`).

### Timestamp Arithmetic

Timestamps can be subtracted, incremented, or decremented.

**Subtracting Timestamps:** The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps. The data type of the result is `DECIMAL(20,6)`.

If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation `result = TS1 - TS2`:

## Expressions

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. Microseconds overflow into seconds.

## Precedence of Operations

Expressions within parentheses are evaluated first.<sup>31</sup> When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication and division are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

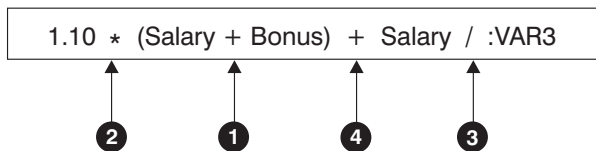
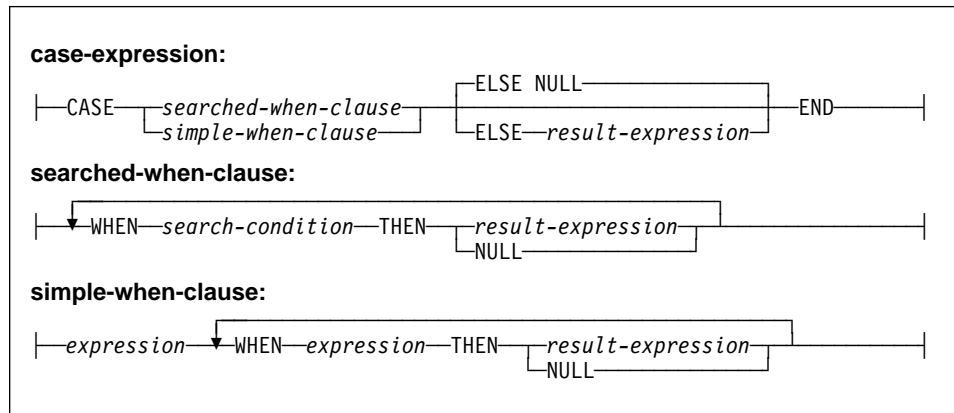


Figure 10.

<sup>31</sup> Note that parentheses are also used in subselect statements, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.

## CASE Expressions



CASE expressions allow 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 then the result is the value of the *result-expression* or NULL. If no case evaluates to true and the ELSE keyword is not present then the result is NULL. Note that 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.

If the CASE expression is in a VALUES clause, an IN predicate, a GROUP BY clause, or an ORDER BY clause, the *search-condition* in a searched-when-clause cannot be a quantified predicate, IN predicate using a fullselect, or an EXISTS predicate (SQLSTATE 42625).

When using the *simple-when-clause*, the value of the *expression* prior to the first *WHEN* keyword is tested for equality with the value of the *expression* following the *WHEN* keyword. The data type of the *expression* prior to the first *WHEN* keyword must therefore be comparable to the data types of each *expression* following the *WHEN* keyword(s). The *expression* prior to the first *WHEN* keyword in a *simple-when-clause* cannot include a function that is variant or has an external action (SQLSTATE 42845).

A *result-expression* is an *expression* following the THEN or ELSE keywords. There must be at least one *result-expression* in the CASE expression (NULL cannot be specified for every case) (SQLSTATE 42625). All *result-expressions* must have compatible data types (SQLSTATE 42804), where the attributes of the result are determined based on the “Rules for Result Data Types” on page 82.

**Examples:**

- If the first character of a department number is a division in the organization, then a CASE expression can be used to list the full name of the division to which each employee belongs:

## Expressions

```
SELECT EMPNO, LASTNAME,  
       CASE SUBSTR(WORKDEPT,1,1)  
       WHEN 'A' THEN 'Administration'  
       WHEN 'B' THEN 'Human Resources'  
       WHEN 'C' THEN 'Accounting'  
       WHEN 'D' THEN 'Design'  
       WHEN 'E' THEN 'Operations'  
       END  
FROM EMPLOYEE;
```

- The number of years of education are used in the EMPLOYEE table to give the education level. A CASE expression can be used to group these and to show the level of education.

```
SELECT EMPNO, FIRSTNAME, MIDINIT, LASTNAME,  
       CASE  
       WHEN EDLEVEL < 15 THEN 'SECONDARY'  
       WHEN EDLEVEL < 19 THEN 'COLLEGE'  
       ELSE 'POST GRADUATE'  
       END  
FROM EMPLOYEE
```

- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% 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 NULL  
       ELSE COMM/SALARY  
       END) > 0.25;
```

- The following CASE expressions are the same:

```
SELECT LASTNAME,  
       CASE  
       WHEN LASTNAME = 'Haas' THEN 'President'  
       ...  
SELECT LASTNAME,  
       CASE LASTNAME  
       WHEN 'Haas' THEN 'President'  
       ...
```

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 9 shows the equivalent expressions using CASE or these functions.

Table 9 (Page 1 of 2). Equivalent CASE Expressions

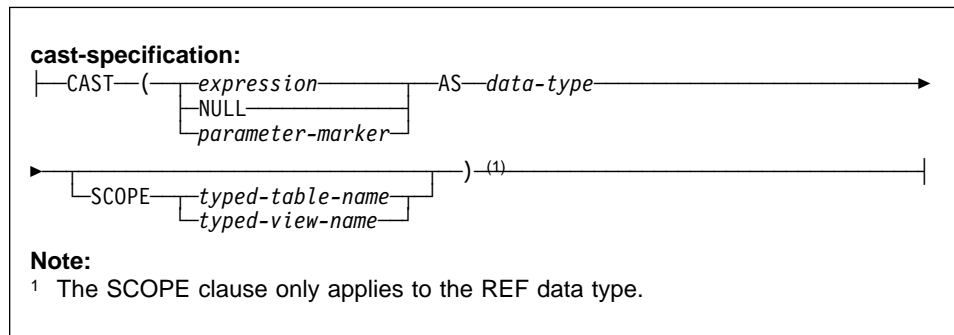
Expression	Equivalent Expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)



Table 9 (Page 2 of 2). Equivalent CASE Expressions

Expression	Equivalent Expression
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)

## CAST Specifications



The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data type*.

### *expression*

If the cast operand is an expression (other than parameter marker or NULL), the result is the argument value converted to the specified target *data type*.

The supported casts are shown in Table 4 on page 69 where the first column represents the data type of the cast operand (source data type) and the data types across the top represent the target data type of the CAST specification. If the cast is not supported an error will occur (SQLSTATE 42846).

When casting character strings (other than CLOBs) to a character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. When casting graphic character strings (other than DBCLOBs) to a graphic character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. For BLOB, CLOB and DBCLOB cast operands, the warning is issued if any characters are truncated.

### **NULL**

If the cast operand is the keyword NULL, the result is a null value that has the specified *data type*.

### *parameter-marker*

A parameter marker (specified as a question mark character) is normally considered an expression, but is documented separately in this case because it has a special meaning. If the cast operand is a *parameter-marker*, the specified *data type* is considered a promise that the replacement will be assignable to the specified data type (using store assignment for strings). Such a parameter marker is consid-

## Expressions

ered a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of function resolution, DESCRIBE of a select list or for column assignment.

### *data type*

The name of an existing data type. If the type name is not qualified, the SQL path is used to do data type resolution. A data type that has an associated attributes like length or precision and scale should include these attributes when specifying *data type* (CHAR defaults to a length of 1 and DECIMAL defaults to a precision of 5 and scale of 0 if not specified). Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an *expression*, see “Casting Between Data Types” on page 67 for the target data types that are supported based on the data type of the cast operand (source data type).
- For a cast operand that is the keyword NULL, any existing data type can be used.
- For a cast operand that is a parameter marker, the target data type can be any existing data type. If the data type is a user-defined type, the application using the parameter marker will use the source data type of the user-defined type.

### SCOPE

When the data type is a reference type, a scope may be defined that identifies the target table or target view of the reference.

#### *typed-table-name*

The name of a typed table. The table must already exist (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM).

#### *typed-view-name*

The name of a typed view. The view must exist or have the same name as the view being created that includes the cast as part of the view definition (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM).

When numeric data is cast to character the result data type is a fixed-length character string (see “CHAR” on page 195). When character data is cast to numeric, the result data type depends on the type of number specified. For example, if cast to integer, it would become a large integer (see “INTEGER” on page 242).

### Examples:

- An application is only interested in the integer portion of the SALARY (defined as decimal(9,2)) from the EMPLOYEE table. The following query, including the employee number and the integer value of SALARY, could be prepared.  

```
SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE
```
- Assume the existence of a distinct type called T\_AGE that is defined on SMALLINT and used to create column AGE in PERSONNEL table. Also assume the existence

of a distinct type called R\_YEAR that is defined on INTEGER and used to create column RETIRE\_YEAR in PERSONNEL table. The following update statement could be prepared.

```
UPDATE PERSONNEL SET RETIRE_YEAR =?  
WHERE AGE = CAST( ? AS T_AGE)
```

The first parameter is an untyped parameter marker that would have a data type of R\_YEAR, although the application will use an integer for this parameter marker. This does not require the explicit CAST specification because it is an assignment.

The second parameter marker is a typed parameter marker that is cast as a distinct type T\_AGE. This satisfies the requirement that the comparison must be performed with compatible data types. The application will use the source data type (which is SMALLINT) for processing this parameter marker.

Successful processing of this statement assumes that the function path includes the schema name of the schema (or schemas) where the two distinct types are defined.

## Expressions

### Dereference Operations

#### dereference-operation:

`|—scoped-ref-expression— → —attribute-name—|`

The dereference operation returns the named column value from the target table or a subtable, or the target view or a subview, of the scoped reference expression from the row with the matching *object identifier column* (OID column). See “CREATE TABLE” on page 522 for further information on the OID column. The result can be null, regardless of whether the column corresponding to the *attribute-name* can be null. If there is no row in the target table with a matching OID, the result is null. The dereference operation takes precedence over all other operators.

#### *scoped-ref-expression*

An expression that is a reference type that has a scope (SQLSTATE 428DT). If the expression is a host variable, parameter marker or other unscoped reference type value, a CAST specification with a SCOPE clause is required to give the reference a scope.

#### *attribute-name*

Specifies the name of an attribute of the target type (column in the target table or view) of the *scoped-ref-expression* (SQLSTATE 42704). The *attribute-name* must be an attribute of the target type of the reference. The data type of the attribute determines the data type of the result of the dereference operation. When the dereference operation is used in a select list and is not included as part of an expression, the attribute name becomes the result column name.

The authorization ID of the statement that uses a dereference operation must have SELECT privilege on the target table of the *scoped-ref-expression* (SQLSTATE 42501).

#### Examples:

- Assume the existence of an EMPLOYEE table that contains a column called DEPTREF which is a reference type scoped to a typed table based on a type that includes the attribute DEPTNAME. The values of DEPTREF in the table EMPLOYEE should correspond to the OID column values in the target table of DEPTREF column.

```
SELECT EMPNO, DEPTREF->DEPTNAME FROM EMPLOYEE
```

---

### Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given row or group.

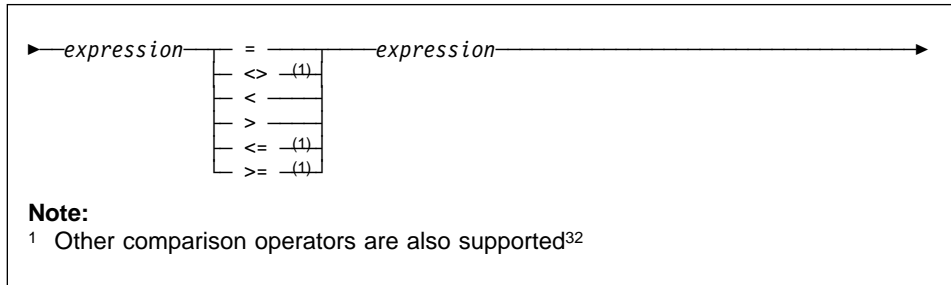
The following rules apply to all types of predicates:

- All values specified in a predicate must be compatible.
- An expression used in a Basic, Quantified, IN, or BETWEEN predicate must not result in a character string with a length attribute greater than 4000, a graphic string with a length attribute greater than 2000, or a LOB string of any size.
- The value of a host variable may be null (that is, the variable may have a negative indicator variable).
- The code page conversion of operands of predicates involving two or more operands, with the exception of LIKE, are done according to "Rules for String Conversions" on page 85
- Use of a DATALINK value is limited to the NULL predicate.

A fullselect is a form of the SELECT statement which is described under Chapter 5, "Queries" on page 315. A fullselect used in a predicate is also called a *subquery*.

## Basic Predicate

## Basic Predicate



A *basic predicate* compares two values.

If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

For values *x* and *y*:

<b>Predicate</b>	<b>Is True If and Only If...</b>
<i>x</i> = <i>y</i>	<i>x</i> is equal to <i>y</i>
<i>x</i> <> <i>y</i>	<i>x</i> is not equal to <i>y</i>
<i>x</i> < <i>y</i>	<i>x</i> is less than <i>y</i>
<i>x</i> > <i>y</i>	<i>x</i> is greater than <i>y</i>
<i>x</i> >= <i>y</i>	<i>x</i> is greater than or equal to <i>y</i>
<i>x</i> <= <i>y</i>	<i>x</i> is less than or equal to <i>y</i>

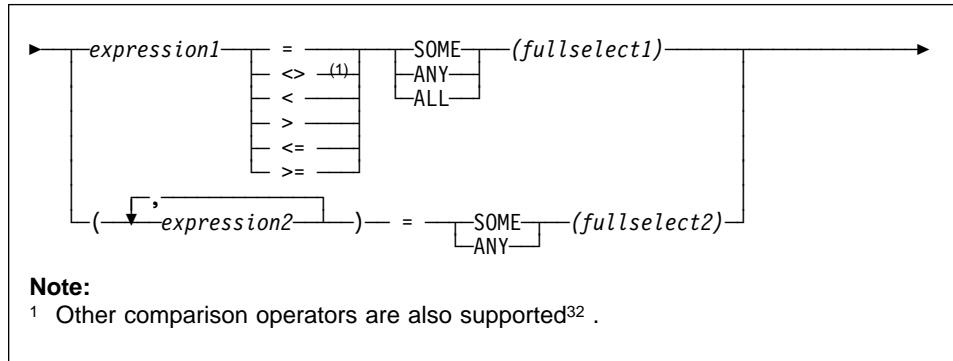
Examples:

```
EMPNO='528671'  
SALARY < 20000  
PRSTAFF <> :VAR1  
SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
```

<sup>32</sup> 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 that uses these operators, and are not recommended for use when writing new SQL statements.

Quantified Predicate



A *quantified predicate* compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the predicate operator (SQLSTATE 428C4). The fullselect may return any number of rows.

When ALL is specified:

- The result of the predicate is true if the fullselect returns no values or if the specified relationship is true for every value returned by the fullselect.
- The result is false if the specified relationship is false for at least one value returned by the fullselect.
- The result is unknown if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of the null value.

When SOME or ANY is specified:

- The result of the predicate is true if the specified relationship is true for each value of at least one row returned by the fullselect.
- The result is false if the fullselect returns no rows or if the specified relationship is false for at least one value of every row returned by the fullselect.
- The result is unknown if the specified relationship is not true for any of the rows and at least one comparison is unknown because of a null value.

Examples: Use the following tables when referring to the following examples.

## Quantified Predicate

<b>TBLAB:</b>	<table border="1"><thead><tr><th>COLA</th><th>COLB</th></tr></thead><tbody><tr><td>1</td><td>12</td></tr><tr><td>2</td><td>12</td></tr><tr><td>3</td><td>13</td></tr><tr><td>4</td><td>14</td></tr><tr><td>-</td><td>-</td></tr></tbody></table>	COLA	COLB	1	12	2	12	3	13	4	14	-	-	<b>TBLXY:</b>	<table border="1"><thead><tr><th>COLX</th><th>COLY</th></tr></thead><tbody><tr><td>2</td><td>22</td></tr><tr><td>3</td><td>23</td></tr></tbody></table>	COLX	COLY	2	22	3	23
COLA	COLB																				
1	12																				
2	12																				
3	13																				
4	14																				
-	-																				
COLX	COLY																				
2	22																				
3	23																				

Figure 11.

### Example 1

```
SELECT COLA FROM TBLAB  
WHERE COLA = ANY(SELECT COLX FROM TBLXY)
```

Results in 2,3. The subselect returns (2,3). COLA in rows 2 and 3 equals at least one of these values.

### Example 2

```
SELECT COLA FROM TBLAB  
WHERE COLA > ANY(SELECT COLX FROM TBLXY)
```

Results in 3,4. The subselect returns (2,3). COLA in rows 3 and 4 is greater than at least one of these values.

### Example 3

```
SELECT COLA FROM TBLAB  
WHERE COLA > ALL(SELECT COLX FROM TBLXY)
```

Results in 4. The subselect returns (2,3). COLA in row 4 is the only one that is greater than both these values.

### Example 4

```
SELECT COLA FROM TBLAB  
WHERE COLA > ALL(SELECT COLX FROM TBLXY  
WHERE COLX<0)
```

Results in 1,2,3,4, null. The subselect returns no values. Thus, the predicate is true for all rows in TBLAB.

### Example 5

```
SELECT * FROM TBLAB  
WHERE (COLA, COLB+10) = SOME (SELECT COLX, COLY FROM TBLXY)
```

The subselect returns all entries from TBLXY. The predicate is true for the subselect, hence the result is as follows:



## Quantified Predicate

COLA	COLB
2	12
3	13

*Example 6*

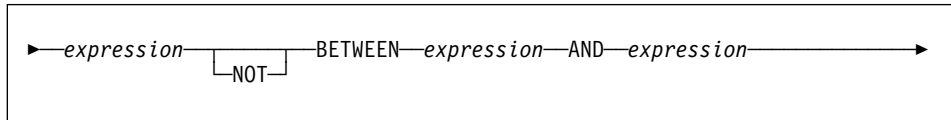
```
SELECT * FROM TBLAB  
  WHERE (COLA, COLB) = ANY (SELECT COLX, COLY-10 FROM TBLXY)
```

The subselect returns COLX and COLY-10 from TBLXY. The predicate is true for the subselect, hence the result is as follows:

COLA	COLB
2	12
3	13

## BETWEEN Predicate

### BETWEEN Predicate



The BETWEEN predicate compares a value with a range of values.

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
value1 >= value2 AND value1 <= value3
```

The BETWEEN predicate:

```
value1 NOT BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3); that is,  
value1 < value2 OR value1 > value3.
```

The values for the expressions in the BETWEEN predicate can have different code pages. The operands are converted as if the above equivalent search conditions were specified.

The first operand (expression) cannot include a function that is variant or has an external action (SQLSTATE 426804).

Given a mixture of datetime values and string representations of datetime values, all values are converted to the data type of the datetime operand.

Examples:

*Example 1*

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000
```

Results in all salaries between \$20,000.00 and \$40,000.00.

*Example 2*

```
SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

Assuming :HV1 is 5000, results in all salaries below \$25,000.00 and above \$40,000.00.

*Example 3*

Given the following:

## BETWEEN Predicate

<i>Table 10.</i>		
Expressions	Type	Code Page
HV_1	host variable	437
HV_2	host variable	437
Col_1	column	850

When evaluating the predicate:

```
:HV_1 BETWEEN :HV_2 AND COL_1
```

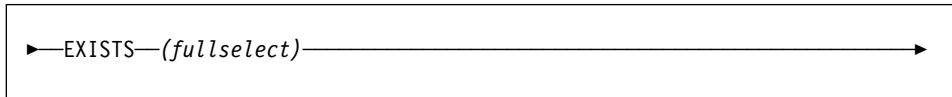
It will be interpreted as:

```
:HV_1 >= :HV_2  
AND :HV_1 <= COL_1
```

The first occurrence of :HV\_1 will remain in the application code page since it is being compared to :HV\_2 which will also remain in the application code page. The second occurrence of :HV\_1 will be converted to the database code page since it is being compared to a column value.

## EXISTS Predicate

### EXISTS Predicate



The EXISTS predicate tests for the existence of certain rows.

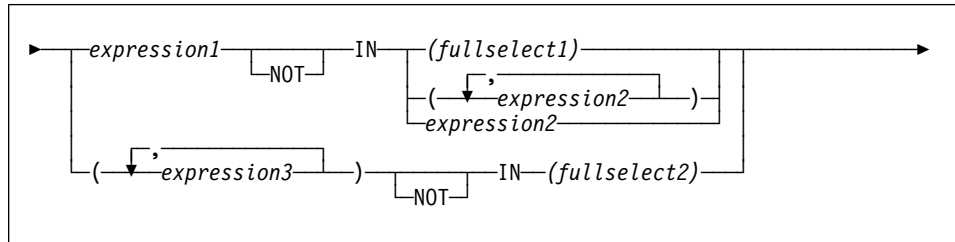
The fullselect may specify any number of columns, and

- The result is true only if the number of rows specified by the fullselect is not zero.
- The result is false only if the number of rows specified is zero
- The result cannot be unknown.

Example:

```
EXISTS (SELECT * FROM TEMPL WHERE SALARY < 10000)
```

## IN Predicate



The IN predicate compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the IN keyword (SQLSTATE 428C4). The fullselect may return any number of rows.

- An IN predicate of the form:
 

```
expression IN expression
```

 is equivalent to a basic predicate of the form:
 

```
expression = expression
```
- An IN predicate of the form:
 

```
expression IN (fullselect)
```

 is equivalent to a quantified predicate of the form:
 

```
expression = ANY (fullselect)
```
- An IN predicate of the form:
 

```
expression NOT IN (fullselect)
```

 is equivalent to a quantified predicate of the form:
 

```
expression <> ALL (fullselect)
```
- An IN predicate of the form:
 

```
expression IN (expressiona, expressionb, ..., expressionk)
```

 is equivalent to:
 

```
expression = ANY (fullselect)
```

 where fullselect in the values-clause form is:
 

```
VALUES (expressiona), (expressionb), ..., (expressionk)
```
- An IN predicate of the form:
 

```
(expressiona, expressionb, ..., expressionk) IN (fullselect)
```

 is equivalent to a quantified predicate of the form:
 

```
(expressiona, expressionb, ..., expressionk) = ANY (fullselect)
```

## IN Predicate

The values for *expression1* and *expression2* or the column of *fullselect1* in the IN predicate must be compatible. Each *expression3* value and its corresponding column of *fullselect2* in the IN predicate must be compatible. The “Rules for Result Data Types” on page 82 can be used to determine the attributes of the result used in the comparison.

The values for the expressions in the IN predicate (including corresponding columns of a fullselect) can have different code pages. If a conversion is necessary then the code page is determined by applying “Rules for String Conversions” on page 85 to the IN list first and then to the predicate using the derived code page for the IN list as the second operand.

Examples:

*Example 1:* The following evaluates to true if the value in the row under evaluation in the DEPTNO column contains D01, B01, or C01:

```
DEPTNO IN ('D01', 'B01', 'C01')
```

*Example 2:* The following evaluates to true only if the EMPNO (employee number) on the left side matches the EMPNO of an employee in department E11:

```
EMPNO IN (SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

*Example 3:* Given the following information, this example evaluates to true if the specific value in the row of the COL\_1 column matches any of the values in the list:

Table 11. IN Predicate example		
Expressions	Type	Code Page
COL_1	column	850
HV_2	host variable	437
HV_3	host variable	437
CON_1	constant	850

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

The two host variables will be converted to code page 850 based on the “Rules for String Conversions” on page 85.

*Example 4:* The following evaluates to true if the specified year in EMENDATE (the date an employee activity on a project ended) matches any of the values specified in the list (the current year or the two previous years):

```
YEAR(EMENDATE) IN (YEAR(CURRENT DATE),  
                   YEAR(CURRENT DATE - 1 YEAR),  
                   YEAR(CURRENT DATE - 2 YEARS))
```

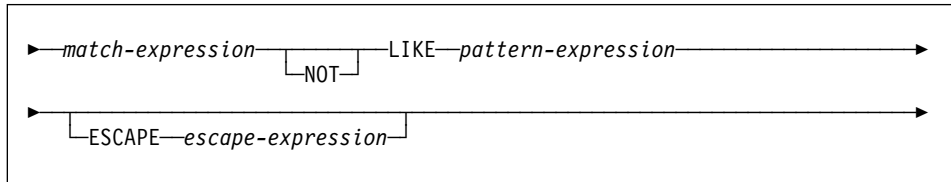
*Example 5:* The following evaluates to true if both ID and DEPT on the left side match MANAGER and DEPTNUMB respectively for any row of the ORG table.

## IN Predicate

```
(ID, DEPT) IN (SELECT MANAGER, DEPTNUMB FROM ORG)
```

## LIKE Predicate

### LIKE Predicate



The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.

If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The values for *match-expression*, *pattern-expression*, and *escape-expression* are compatible string expressions. There are slight differences in the types of string expressions supported for each of the arguments. The valid types of expressions are listed under the description of each argument.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

#### *match-expression*

An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable (including a locator variable or a file reference variable)
- a scalar function
- a large object locator
- a column name
- an expression concatenating any of the above

#### *pattern-expression*

An expression that specifies the string that is to be matched.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable
- a scalar function whose operands are any of the above
- an expression concatenating any of the above

with the restrictions that:



## LIKE Predicate

- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC or DBCLOB. In addition it cannot be a BLOB file reference variable.
- The actual length of *pattern-expression* cannot be more than 4000 bytes.

A **simple description** of the use of the LIKE pattern is that 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 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 use of the LIKE pattern follows. Note that this description ignores the use of the *escape-expression*; its use 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, it follows that 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  $p$ ).

## LIKE Predicate

When the *escape-expression* is specified, the *pattern-expression* must not contain the escape character identified by the *escape-expression* except when immediately followed by the escape character, the underscore character or the percent sign character (SQLSTATE 22025).

If the *match-expression* is a character string in an MBCS database then it can contain **mixed data**. In this case, the pattern can include both SBCS and MBCS characters. 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 (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

### *escape-expression*

This optional argument is an expression that specifies a character to be used to modify the special meaning of the underscore (`_`) and percent (`%`) characters in the *pattern-expression*. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable
- a scalar function whose operands are any of the above
- an expression concatenating any of the above

with the restrictions that:

- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- The result of the expression must be one SBCS or DBCS character or a binary string containing exactly 1 byte (SQLSTATE 22019).

When escape characters are present in the pattern string, an underscore, percent sign, or escape character can represent a literal occurrence of itself. This is true if the character in question is preceded by an odd number of successive escape characters. It is not true otherwise.

In a pattern, a sequence of successive escape characters is treated as follows:

Let S be such a sequence, and suppose that S is not part of a larger sequence of successive escape characters. Suppose also that S contains a total of n characters. Then the rules governing S depend on the value of n:

- If n is odd, S must be followed by an underscore or percent sign (SQLSTATE 22025). S and the character that follows it represent (n-1)/2 literal occurrences of the escape character followed by a literal occurrence of the underscore or percent sign.

## LIKE Predicate

- If *n* is even, *S* represents *n*/2 literal occurrences of the escape character. Unlike the case where *n* is odd, *S* could end the pattern. If it does not end the pattern, it can be followed by any character (except, of course, an escape character, which would violate the assumption that *S* is not part of a larger sequence of successive escape characters). If *S* is followed by an underscore or percent sign, that character has its special meaning.

Following is an illustration of the effect of successive occurrences of the escape character (which, in this case, is the back slash (\) ).

Pattern string	Actual Pattern
\%	A percent sign
\\%	A back slash followed by zero or more arbitrary characters
\\\%	A back slash followed by a percent sign

The code page used in the comparison is based on the code page of the *match-expression* value.

- The *match-expression* value is never converted.
- If the code page of *pattern-expression* is different from the code page of *match-expression*, the value of *pattern-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA (in which case there is no conversion).
- If the code page of *escape-expression* is different from the code page of *match-expression*, the value of *escape-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA (in which case there is no conversion).

### Examples

- Search for the string 'SYSTEMS' appearing anywhere within the PROJNAME column in the PROJECT table.

```
SELECT PROJNAME FROM PROJECT
WHERE PROJECT.PROJNAME LIKE '%SYSTEMS%'
```

- Search for a string with a first character of 'J' that is exactly two characters long in the FIRSTNAME column of the EMPLOYEE table.

```
SELECT FIRSTNAME FROM EMPLOYEE
WHERE EMPLOYEE.FIRSTNAME LIKE 'J_'
```

- In the CORP\_SERVERS table, search for a string in the LA\_SERVERS column that matches the value in the CURRENT SERVER special register.

```
SELECT LA_SERVERS FROM CORP_SERVERS
WHERE CORP_SERVERS.LA_SERVERS LIKE CURRENT SERVER
```

- Retrieve all strings that begin with the sequence of characters '%\_\' in column A of the table T.

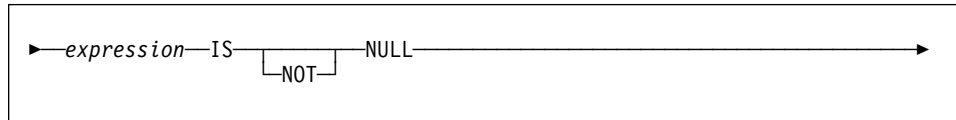
## LIKE Predicate

```
SELECT A FROM T WHERE T.A LIKE  
'\%\_\\%' ESCAPE '\'
```

- Use the BLOB scalar function, to obtain a one byte escape character which is compatible with the match and pattern data types (both BLOBs).

```
SELECT COLBLOB FROM TABLET  
WHERE COLBLOB LIKE :pattern_var ESCAPE BLOB(X'0E')
```

### 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.

Examples:

**PHONENO IS NULL**

**SALARY IS NOT NULL**

## TYPE Predicate

### TYPE Predicate

→ *expression* IS OF DYNAMIC TYPE ( , *typename* ) →  
↑ ↓  
ONLY

A *TYPE predicate* compares the type of an expression with one or more user-defined structured types.

The dynamic type of an expression involving the dereferencing of a reference type is the actual type of the referenced row from the target typed table or view. This may differ from the target type of an expression involving the reference which is called the static type of the expression.

If the value of *expression* is null, the result of the predicate is unknown. The result of the predicate is true if the dynamic type of the *expression* is a subtype of one of the structured types specified by *typename*, otherwise the result is false. If *ONLY* precedes any *typename* the proper subtypes of that type are not considered.

If *typename* is not qualified, it is resolved using the SQL path. Each *typename* must identify a user-defined type that is in the type hierarchy of the static type of *expression* (SQLSTATE 428DU).

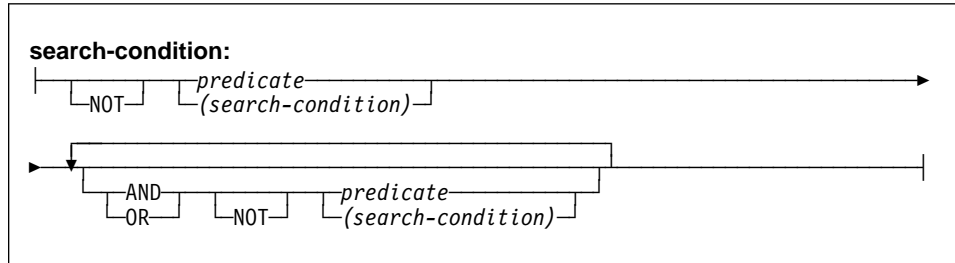
The Deref function should be used whenever the TYPE predicate has an expression involving a reference type value. The static type for this form of *expression* is the target type of the reference. See "DEREF" on page 217 for more information about the Deref function.

Example:

A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO\_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. The following is a type predicate that evaluates to true when a row corresponding to WHO\_RESPONSIBLE is a manager :

```
DEREF (WHO_RESPONSIBLE) IS OF DYNAMIC TYPE (MGR)
```

Search Conditions



A *search condition* specifies a condition that is “true,” “false,” or “unknown” about a given row.

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 Table 12, in which P and Q are any predicates:

Table 12. Truth Tables 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.

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.

## Search Conditions

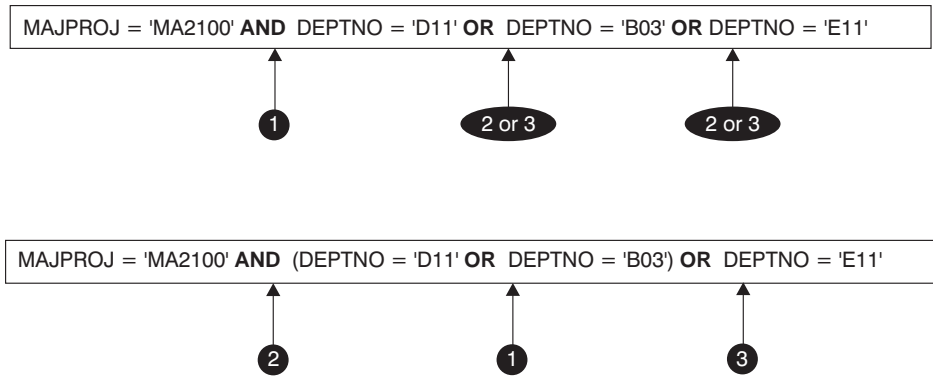


Figure 12. Search Conditions Evaluation Order



---

## Chapter 4. Functions

A function is an operation that is denoted by a function name followed by a pair of parentheses enclosing the specification of arguments (there may be no arguments).

Functions are classified as *column functions*, *scalar functions* or *table functions*. The argument of a column function is a collection of like values. It returns a single value (possibly null), and can be specified in an SQL statement where an *expression* can be used. Additional restrictions apply to the use of column functions as specified in "Column Functions" on page 170. The argument(s) of a scalar function are individual scalar values, which can be of different types and have different meanings. It returns a single value (possibly null), and can be specified in an SQL statement wherever an *expression* can be used. The argument(s) of a table function are individual scalar values, which can be of different types and have different meanings. It returns a table to the SQL statement, and can be specified only within the FROM clause of a SELECT. Additional restrictions apply to the use of table functions as specified in "from-clause" on page 321.

Table 13 on page 156 shows the functions that are supported. The "Function Name" combined with the "Schema" give the fully qualified name of the function. "Description" briefly describes what the function does. "Input Parameters" gives the data type that is expected for each argument during function invocation. Many of the functions include variations of the input parameters allowing either different data types or different numbers of arguments to be used. The combination of schema, function name and input parameters make up a function signature. Each function signature may return a value of a different type which is shown in the "Returns" columns. There are some distinctions that should be understood about the input parameter types. In some cases the type is specified as a specific built-in data type and in other cases it will use a general variable like *any-numeric-type*. When a specific data type is listed, this means that an exact match will only occur with the specified data type. When a general variable is used, each of the data types associated with that variable will result in an exact match. This distinction impacts function selection as described in "Function Resolution" on page 112.

There may be additional functions available because user-defined functions may be created in different schemas using one of these function signatures as a source (see "CREATE FUNCTION" for details) or users may create external functions using their own programs.

Table 13 (Page 1 of 13). Supported Functions

Function name	Schema	Description
	Input Parameters	
ABS or ABSVAL	SYSFUN	Returns the absolute value of the argument.
	SMALLINT	SMALLINT
	INTEGER	INTEGER
	BIGINT	BIGINT
	DOUBLE	DOUBLE
ACOS	SYSFUN	Returns the arccosine of the argument as an angle expressed in radians.
	DOUBLE	DOUBLE
ASCII	SYSFUN	Returns the ASCII code value of the leftmost character of the argument as an integer.
	CHAR	INTEGER
	VARCHAR	INTEGER
	CLOB(1M)	INTEGER
ASIN	SYSFUN	Returns the arcsine of the argument as an angle, expressed in radians.
	DOUBLE	DOUBLE
ATAN	SYSFUN	Returns the arctangent of the argument as an angle, expressed in radians.
	DOUBLE	DOUBLE
ATAN2	SYSFUN	Returns the arctangent of <i>x</i> and <i>y</i> coordinates, specified by the first and second arguments respectively, as an angle, expressed in radians.
	DOUBLE, DOUBLE	DOUBLE
AVG	SYSIBM	Returns the average of a set of numbers (column function).
	<i>numeric-type</i> <sup>4</sup>	<i>numeric-type</i> <sup>1</sup>
BIGINT	SYSIBM	Returns a 64 bit integer representation of a number or character string in the form of an integer constant.
	<i>numeric-type</i>	BIGINT
	VARCHAR	BIGINT
BLOB	SYSIBM	Casts from source type to BLOB, with optional length.
	<i>string-type</i>	BLOB
	<i>string-type</i> , INTEGER	BLOB
CEIL or CEILING	SYSFUN	Returns the smallest integer greater than or equal to the argument.
	SMALLINT	SMALLINT
	INTEGER	INTEGER
	BIGINT	BIGINT
	DOUBLE	DOUBLE

Table 13 (Page 2 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
CHAR	SYSIBM	Returns a string representation of the source type.	
	<i>character-type</i>		CHAR
	<i>character-type</i> , INTEGER		CHAR( <i>integer</i> )
	<i>datetime-type</i>		CHAR
	<i>datetime-type</i> , keyword <sup>2</sup>		CHAR
	SMALLINT		CHAR(6)
	INTEGER		CHAR(11)
	BIGINT		CHAR(20)
	DECIMAL		CHAR(2+ <i>precision</i> )
DECIMAL, VARCHAR		CHAR(2+ <i>precision</i> )	
CHAR	SYSFUN	Returns a character string representation of a floating-point number.	
	DOUBLE		CHAR(24)
CHR	SYSFUN	Returns the character that has the ASCII code value specified by the argument. The value of the argument should be between 0 and 255; otherwise, the return value is null.	
	INTEGER		CHAR(1)
CLOB	SYSIBM	Casts from source type to CLOB, with optional length.	
	<i>character-type</i>		CLOB
	<i>character-type</i> , INTEGER		CLOB
COALESCE <sup>3</sup>	SYSIBM	Returns the first non-null argument in the set of arguments.	
	<i>any-type</i> , <i>any-union-compatible-type</i> , ...		<i>any-type</i>
CONCAT or	SYSIBM	Returns the concatenation of 2 string arguments.	
	<i>string-type</i> , <i>compatible-string-type</i>		<i>max string-type</i>
COS	SYSFUN	Returns the cosine of the argument, where the argument is an angle expressed in radians.	
	DOUBLE		DOUBLE
COT	SYSFUN	Returns the cotangent of the argument, where the argument is an angle expressed in radians.	
	DOUBLE		DOUBLE
COUNT	SYSIBM	Returns the count of the number of rows in a set of rows or values (column function).	
	<i>any-builtin-type</i> <sup>4</sup>		INTEGER
COUNT_BIG	SYSIBM	Returns the number of rows or values in a set of rows or values (column function). Result can be greater than the maximum value of integer.	
	<i>any-builtin-type</i> <sup>4</sup>		DECIMAL(31,0)
DATE	SYSIBM	Returns a date from a single input value.	
	DATE		DATE
	TIMESTAMP		DATE
	DOUBLE		DATE
	VARCHAR		DATE

Table 13 (Page 3 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
DAY	SYSIBM	Returns the day part of a value.	
	VARCHAR		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
DAYNAME	SYSFUN	Returns a mixed case character string containing the name of the day (e.g. Friday) for the day portion of the argument based on what the locale was when db2start was issued.	
	VARCHAR(26)		VARCHAR(100)
	DATE		VARCHAR(100)
	TIMESTAMP		VARCHAR(100)
DAYOFWEEK	SYSFUN	Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.	
	VARCHAR(26)		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
DAYOFYEAR	SYSFUN	Returns the day of the year in the argument as an integer value in the range 1-366.	
	VARCHAR(26)		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
DAYS	SYSIBM	Returns an integer representation of a date.	
	VARCHAR		INTEGER
	TIMESTAMP		INTEGER
	DATE		INTEGER
DBCLOB	SYSIBM	Casts from source type to DBCLOB, with optional length.	
	<i>graphic-type</i>		DBCLOB
	<i>graphic-type</i> , INTEGER		DBCLOB
DECIMAL or DEC	SYSIBM	Returns decimal representation of a number, with optional precision and scale.	
	<i>numeric-type</i>		DECIMAL
	<i>numeric-type</i> , INTEGER		DECIMAL
	<i>numeric-type</i> INTEGER, INTEGER		DECIMAL
DECIMAL or DEC	SYSIBM	Returns decimal representation of a character string, with optional precision, scale, and decimal-character.	
	VARCHAR		DECIMAL
	VARCHAR, INTEGER		DECIMAL
	VARCHAR, INTEGER, INTEGER		DECIMAL
VARCHAR, INTEGER, INTEGER, VARCHAR		DECIMAL	

Table 13 (Page 4 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
DEGREES	SYSFUN	Returns the number of degrees converted from the argument in expressed in radians.	
	DOUBLE		DOUBLE
DEREF	SYSIBM	Returns an instance of the target type of the reference type argument.	
	REF( <i>any-structured-type</i> ) with defined scope		<i>any-structured-type (same as input target type)</i>
DIFFERENCE	SYSFUN	Returns the difference between the sounds of the words in the two argument strings as determined using the SOUNDEX function. A value of 4 means the strings sound the same.	
	VARCHAR, VARCHAR		INTEGER
DIGITS	SYSIBM	Returns the character string representation of a number.	
	DECIMAL		CHAR
DLCOMMENT	SYSIBM	Returns the comment attribute of a datalink value.	
	DATALINK		VARCHAR(254)
DLLINKTYPE	SYSIBM	Returns the link type attribute of a datalink value.	
	DATALINK		VARCHAR(4)
DLURLCOMPLETE	SYSIBM	Returns the complete URL (including access token) of a datalink value.	
	DATALINK		VARCHAR
DLURLPATH	SYSIBM	Returns the path and file name (including access token) of a datalink value.	
	DATALINK		VARCHAR
DLURLPATHONLY	SYSIBM	Returns the path and file name (without any access token) of a datalink value.	
	DATALINK		VARCHAR
DLURLSCHEME	SYSIBM	Returns the scheme from the URL attribute of a datalink value.	
	DATALINK		VARCHAR
DLURLSERVER	SYSIBM	Returns the server from the URL attribute of a datalink value.	
	DATALINK		VARCHAR
DLVALUE	SYSIBM	Builds a datalink value from a data-location argument, link type argument and optional comment-string argument.	
	VARCHAR		DATALINK
	VARCHAR, VARCHAR		DATALINK
	VARCHAR, VARCHAR, VARCHAR		DATALINK
DOUBLE or DOUBLE_PRECISION	SYSIBM	Returns the floating-point representation of a number.	
	<i>numeric-type</i>		DOUBLE
DOUBLE	SYSFUN	Returns the floating-point number corresponding to the character string representation of a number. Leading and trailing blanks in <i>argument</i> are ignored.	
	VARCHAR		DOUBLE
EVENT_MON_STATE	SYSIBM	Returns the operational state of particular event monitor.	
	VARCHAR		INTEGER
EXP	SYSFUN	Returns the exponential function of the argument.	
	DOUBLE		DOUBLE

Table 13 (Page 5 of 13). Supported Functions

Function name	Schema	Description
	Input Parameters	
FLOAT	SYSIBM	Same as DOUBLE.
FLOOR	SYSFUN	Returns the largest integer less than or equal to the argument.
	SMALLINT	SMALLINT
	INTEGER	INTEGER
	BIGINT	BIGINT
	DOUBLE	DOUBLE
GENERATE_UNIQUE	SYSIBM	Returns a bit data character string that is unique compared to any other execution of the same function.
	<i>no argument</i>	CHAR(13) FOR BIT DATA
GRAPHIC	SYSIBM	Cast from source type to GRAPHIC, with optional length.
	<i>graphic-type</i>	GRAPHIC
	<i>graphic-type</i> , INTEGER	GRAPHIC
GROUPING	SYSIBM	Used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set (column function). The value returned is: 1 The value of the argument in the returned row is a null value and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set. 0 otherwise.
	<i>any-type</i>	SMALLINT
HEX	SYSIBM	Returns the hexadecimal representation of a value.
	<i>any-builtin-type</i>	VARCHAR
HOUR	SYSIBM	Returns the hour part of a value.
	VARCHAR	INTEGER
	TIME	INTEGER
	TIMESTAMP	INTEGER
	DECIMAL	INTEGER
INSERT	SYSFUN	Returns a string where <i>argument3</i> bytes have been deleted from <i>argument1</i> beginning at <i>argument2</i> and where <i>argument4</i> has been inserted into <i>argument1</i> beginning at <i>argument2</i> .
	VARCHAR, INTEGER, INTEGER, VARCHAR	VARCHAR
	CLOB(1M), INTEGER, INTEGER, CLOB(1M)	CLOB(1M)
	BLOB(1M), INTEGER, INTEGER, BLOB(1M)	BLOB(1M)
INTEGER or INT	SYSIBM	Returns the integer representation of a number.
	<i>numeric-type</i>	INTEGER
	VARCHAR	INTEGER
JULIAN_DAY	SYSFUN	Returns an integer value representing the number of days from January 1, 4712 B.C. (the start of the Julian date calendar) to the date value specified in the <i>argument</i> .
	VARCHAR(26)	INTEGER
	DATE	INTEGER
	TIMESTAMP	INTEGER

Table 13 (Page 6 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
LCASE	SYSFUN	Returns a string in which all the characters have been converted to lower case characters. LCASE will only handle characters in the invariant set. Therefore, LCASE(UCASE(string)) will not necessarily return the same result as LCASE(string).	
	VARCHAR		VARCHAR
	CLOB(1M)		CLOB(1M)
LEFT	SYSFUN	Returns a string consisting of the leftmost <i>argument2</i> bytes in <i>argument1</i> .	
	VARCHAR, INTEGER		VARCHAR
	CLOB(1M), INTEGER		CLOB(1M)
	BLOB(1M), INTEGER		BLOB(1M)
LENGTH	SYSIBM	Returns the length of the operand in bytes (except for double byte string types which return the length in characters).	
	<i>any-builtin-type</i>		INTEGER
LN	SUSFUN	Returns the natural logarithm of the argument (same as LOG).	
	DOUBLE		DOUBLE
LOCATE	SYSFUN	Returns the starting position of the first occurrence of <i>argument1</i> within <i>argument2</i> . If the optional third argument is specified, it indicates the character position in <i>argument2</i> at which the search is to begin. If <i>argument1</i> is not found within <i>argument2</i> , the value 0 is returned.	
	VARCHAR, VARCHAR		INTEGER
	VARCHAR, VARCHAR, INTEGER		INTEGER
	CLOB(1M), CLOB(1M)		INTEGER
	CLOB(1M), CLOB(1M), INTEGER		INTEGER
	BLOB(1M), BLOB(1M)		INTEGER
	BLOB(1M), BLOB(1M), INTEGER		INTEGER
LOG	SYSFUN	Returns the natural logarithm of the argument (same as LN).	
	DOUBLE		DOUBLE
LOG10	Returns the base 10 logarithm of the argument.		
	DOUBLE		DOUBLE
LONG_VARCHAR	SYSIBM	Returns a long string.	
	<i>character-type</i>		LONG VARCHAR
LONG_VARGRAPHIC	SYSIBM	Casts from source type to LONG_VARGRAPHIC.	
	<i>graphic-type</i>		LONG VARGRAPHIC
LTRIM	SYSFUN	Returns the characters of the argument with leading blanks removed.	
	VARCHAR		VARCHAR
	CLOB(1M)		CLOB(1M)
MAX	SYSIBM	Returns the maximum value in a set of values (column function).	
	<i>any-builtin-type</i> <sup>5</sup>		<i>same as input type</i>

Table 13 (Page 7 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
MICROSECOND	SYSIBM	Returns the microsecond (time-unit) part of a value.	
	VARCHAR		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
MIDNIGHT_SECONDS	SYSFUN	Returns an integer value in the range 0 to 86400 representing the number of seconds between midnight and time value specified in the <i>argument</i> .	
	VARCHAR(26)		INTEGER
	TIME		INTEGER
	TIMESTAMP		INTEGER
MIN	SYSIBM	Returns the minimum value in a set of values (column function).	
	<i>any-builtin-type</i> <sup>5</sup>		<i>same as input type</i>
MINUTE	SYSIBM	Returns the minute part of a value.	
	VARCHAR		INTEGER
	TIME		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
MOD	SYSFUN	Returns the remainder ( modulus) of <i>argument1</i> divided by <i>argument2</i> . The result is negative only if <i>argument1</i> is negative.	
	SMALLINT, SMALLINT		SMALLINT
	INTEGER, INTEGER		INTEGER
	BIGINT, BIGINT		BIGINT
MONTH	SYSIBM	Returns the month part of a value.	
	VARCHAR		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
MONTHNAME	SYSFUN	Returns a mixed case character string containing the name of month (e.g. January) for the month portion of the argument that is a date or timestamp, based on what the locale was when the database was started.	
	VARCHAR(26)		VARCHAR(100)
	DATE		VARCHAR(100)
	TIMESTAMP		VARCHAR(100)
NODENUMBER <sup>3</sup>	SYSIBM	Returns the node number of the row. The argument is a column name within a table.	
	<i>any-type</i>		INTEGER
NULLIF <sup>3</sup>	SYSIBM	Returns NULL if the arguments are equal, else returns the first argument.	
	<i>any-type</i> <sup>5</sup> , <i>any-comparable-type</i> <sup>5</sup>		<i>any-type</i>
PARTITION <sup>3</sup>	SYSIBM	Returns the partitioning map index (0 to 4095) of the row. The argument is a column name within a table.	
	<i>any-type</i>		INTEGER



Table 13 (Page 8 of 13). Supported Functions

Function name	Schema	Description	Returns
	Input Parameters		
POSSTR	SYSIBM	Returns the position at which one string is contained in another.	INTEGER
	<i>string-type, compatible-string-type</i>		
POWER	SYSFUN	Returns the value of <i>argument1</i> to the power of <i>argument2</i> .	INTEGER
	INTEGER, INTEGER		
	BIGINT, BIGINT		
	DOUBLE, INTEGER		
	DOUBLE, DOUBLE		
QUARTER	SYSFUN	Returns an integer value in the range 1 to 4 representing the quarter of the year for the date specified in the argument.	INTEGER
	VARCHAR(26)		
	DATE		
	TIMESTAMP		
RADIANS	SYSFUN	Returns the number of radians converted from argument which is expressed in degrees.	DOUBLE
	DOUBLE		
RAISE_ERROR	SYSIBM	Raises an error in the SQLCA. The sqlstate returned is indicated by <i>argument1</i> . The second argument contains any text to be returned.	<i>any-type</i> <sup>6</sup>
	VARCHAR, VARCHAR		
RAND	SYSFUN	Returns a random floating point value between 0 and 1 using the argument as the optional seed value.	DOUBLE
	<i>no argument required</i>		
	INTEGER		
REAL	SYSIBM	Returns the single-precision floating-point representation of a number.	REAL
	<i>numeric-type</i>		
REPEAT	SYSFUN	Returns a character string composed of <i>argument1</i> repeated <i>argument2</i> times.	VARCHAR
	VARCHAR, INTEGER		
	CLOB(1M), INTEGER		
	BLOB(1M), INTEGER		
REPLACE	SYSFUN	Replaces all occurrences of <i>argument2</i> in <i>argument1</i> with <i>argument3</i> .	VARCHAR
	VARCHAR, VARCHAR, VARCHAR		
	CLOB(1M), CLOB(1M), CLOB(1M)		
	BLOB(1M), BLOB(1M), BLOB(1M)		
RIGHT	SYSFUN	Returns a string consisting of the rightmost <i>argument2</i> bytes in <i>argument1</i> .	VARCHAR
	VARCHAR, INTEGER		
	CLOB(1M), INTEGER		
	BLOB(1M), INTEGER		

Table 13 (Page 9 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
ROUND	SYSFUN	Returns the first argument rounded to <i>argument2</i> places <b>right</b> of the decimal point. If <i>argument2</i> is negative, <i>argument1</i> is rounded to the absolute value of <i>argument2</i> places to the <b>left</b> of the decimal point.	
	INTEGER, INTEGER		INTEGER
	BIGINT, INTEGER		BIGINT
	DOUBLE, INTEGER		DOUBLE
RTRIM	SYSFUN	Returns the characters of the argument with trailing blanks removed.	
	VARCHAR		VARCHAR
	CLOB(1M)		CLOB(1M)
SECOND	SYSIBM	Returns the second (time-unit) part of a value.	
	VARCHAR		INTEGER
	TIME		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
SIGN	SYSFUN	Returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.	
	SMALLINT		SMALLINT
	INTEGER		INTEGER
	BIGINT		BIGINT
	DOUBLE		DOUBLE
SIN	SYSFUN	Returns the sine of the argument, where the argument is an angle expressed in radians.	
	DOUBLE		DOUBLE
SMALLINT	SYSIBM	Returns the small integer representation of a number.	
	<i>numeric-type</i>		SMALLINT
	VARCHAR		SMALLINT
SOUNDEX	SYSFUN	Returns a 4 character code representing the sound of the words in the argument. The result can be used to compare with the sound of other strings. See also DIFFERENCE.	
	VARCHAR		CHAR(4)
SPACE	SYSFUN	Returns a character string consisting of <i>argument1</i> blanks.	
	INTEGER		VARCHAR
SQRT	SYSFUN	Returns the square root of the argument.	
	DOUBLE		DOUBLE
STDDEV	SYSIBM	Returns the standard deviation of a set of numbers (column function).	
	DOUBLE		DOUBLE

Table 13 (Page 10 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
SUBSTR	SYSIBM	Returns a substring of a string <i>argument1</i> starting at <i>argument2</i> for <i>argument3</i> characters. If <i>argument3</i> is not specified, the remainder of the string is assumed.	
	<i>string-type</i> , INTEGER		<i>string-type</i>
	<i>string-type</i> , INTEGER, INTEGER		<i>string-type</i>
SUM	SYSIBM	Returns the sum of a set of numbers (column function).	
	<i>numeric-type</i> <sup>4</sup>		<i>max-numeric-type</i> <sup>1</sup>
TABLE_NAME	SYSIBM	Returns an unqualified name of a table or view based on the object name given in <i>argument1</i> and the optional schema name given in <i>argument2</i> . It is used to resolve aliases.	
	VARCHAR		VARCHAR(18)
	VARCHAR, VARCHAR		VARCHAR(18)
TABLE_SCHEMA	SYSIBM	Returns the schema name portion of the two part table or view name given by the object name in <i>argument1</i> and the optional schema name in <i>argument2</i> . It is used to resolve aliases.	
	VARCHAR		CHAR(8)
	VARCHAR, VARCHAR		CHAR(8)
TAN	SYSFUN	Returns the tangent of the argument, where the argument is an angle expressed in radians.	
	DOUBLE		DOUBLE
TIME	SYSIBM	Returns a time from a value.	
	TIME		TIME
	TIMESTAMP		TIME
	VARCHAR		TIME
TIMESTAMP	SYSIBM	Returns a timestamp from a value or a pair of values.	
	TIMESTAMP		TIMESTAMP
	VARCHAR		TIMESTAMP
	VARCHAR, VARCHAR		TIMESTAMP
	VARCHAR, TIME		TIMESTAMP
	DATE, VARCHAR		TIMESTAMP
	DATE, TIME		TIMESTAMP
TIMESTAMP_ISO	SYSFUN	Returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of CURRENT DATE for the date elements and zero for the fractional time element.	
	DATE		TIMESTAMP
	TIME		TIMESTAMP
	TIMESTAMP		TIMESTAMP
	VARCHAR(26)		TIMESTAMP

Table 13 (Page 11 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
TIMESTAMPDIFF	SYSFUN	Returns an estimated number of intervals of type <i>argument1</i> based on the difference between two timestamps. The second argument is the result of subtracting two timestamp types and converting the result to CHAR. Valid values of <i>interval(argument1)</i> are:  1 Fractions of a second 2 Seconds 4 Minutes 8 Hours 16 Days 32 Weeks 64 Months 128 Quarters 256 Years	
	INTEGER, CHAR(22)		INTEGER
TRANSLATE	SYSIBM	Returns a string in which one or more characters may have been translated into other characters.	
	CHAR		CHAR
	VARCHAR		VARCHAR
	CHAR, VARCHAR, VARCHAR		CHAR
	VARCHAR, VARCHAR, VARCHAR		VARCHAR
	CHAR, VARCHAR, VARCHAR, VARCHAR		CHAR
	VARCHAR, VARCHAR, VARCHAR, VARCHAR		VARCHAR
	GRAPHIC, VARGRAPHIC, VARGRAPHIC		GRAPHIC
	VARGRAPHIC, VARGRAPHIC, VARGRAPHIC		VARGRAPHIC
	GRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC		GRAPHIC
VARGRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC		VARGRAPHIC	
TRUNC or TRUNCATE	SYSFUN	Returns <i>argument1</i> truncated to <i>argument2</i> places <b>right</b> of the decimal point. If <i>argument2</i> is negative, <i>argument1</i> is truncated to the absolute value of <i>argument2</i> places to the <b>left</b> of the decimal point.	
	INTEGER, INTEGER		INTEGER
	BIGINT, INTEGER		BIGINT
	DOUBLE, INTEGER		DOUBLE
TYPE_ID <sup>3</sup>	SYSIBM	Returns the internal data type identifier of the dynamic data type of the argument. Note that the result of this function is not portable across databases.	
	<i>any-structured-type</i>		INTEGER
TYPE_NAME <sup>3</sup>	SYSIBM	Returns the unqualified name of the dynamic data type of the argument.	
	<i>any-structured-type</i>		VARCHAR(18)
TYPE_SCHEMA <sup>3</sup>	SYSIBM	Returns the schema name of the dynamic type of the argument.	
	<i>any-structured-type</i>		CHAR(8)
UCASE	SYSFUN	Returns a string in which all the characters have been converted to upper case characters.	
	VARCHAR		VARCHAR
VALUE <sup>3</sup>	SYSIBM	Same as COALESCE.	

Table 13 (Page 12 of 13). Supported Functions

Function name	Schema	Description	
	Input Parameters		Returns
VARCHAR	SYSIBM	Returns a VARCHAR representation of the first argument. If a second argument is present, it specifies the length of the result.	
	<i>character-type</i>		VARCHAR
	<i>character-type</i> , INTEGER		VARCHAR
	<i>datetime-type</i>		VARCHAR
VARGRAPHIC	SYSIBM	Returns a VARGRAPHIC representation of the first argument. If a second argument is present, it specifies the length of the result.	
	<i>graphic-type</i>		VARGRAPHIC
	<i>graphic-type</i> , INTEGER		VARGRAPHIC
	VARCHAR		VARGRAPHIC
VARIANCE or VAR	SYSIBM	Returns the variance of a set of numbers (column function).	
	DOUBLE		DOUBLE
WEEK	SYSFUN	Returns the week of the year in of the argument as an integer value in the range of 1-54.	
	VARCHAR(26)		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
YEAR	SYSIBM	Returns the year part of a value.	
	VARCHAR		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
“+”	SYSIBM	Adds two numeric operands.	
	<i>numeric-type, numeric-type</i>		<i>max numeric-type</i>
“+”	SYSIBM	Unary plus operator.	
	<i>numeric-type</i>		<i>numeric-type</i>
“+”	SYSIBM	Datetime plus operator.	
	DATE, DECIMAL(8,0)		DATE
	TIME, DECIMAL(6,0)		TIME
	TIMESTAMP, DECIMAL(20,6)		TIMESTAMP
	DECIMAL(8,0), DATE		DATE
	DECIMAL(6,0), TIME		TIME
	DECIMAL(20,6), TIMESTAMP		TIMESTAMP
	<i>datetime-type, DOUBLE, labeled-duration-code</i>		<i>datetime-type</i>
“-”	SYSIBM	Subtracts two numeric operands.	
	<i>numeric-type, numeric-type</i>		<i>max numeric-type</i>
“-”	SYSIBM	Unary minus operator.	
	<i>numeric-type</i>		<i>numeric-type</i> <sup>1</sup>

Table 13 (Page 13 of 13). Supported Functions

Function name	Schema	Description
	Input Parameters	
“-”	SYSIBM	Datetime minus operator.
	DATE, DATE	DECIMAL(8,0)
	TIME, TIME	DECIMAL(6,0)
	TIMESTAMP, TIMESTAMP	DECIMAL(20,6)
	DATE, VARCHAR	DECIMAL(8,0)
	TIME, VARCHAR	DECIMAL(6,0)
	TIMESTAMP, VARCHAR	DECIMAL(20,6)
	VARCHAR, DATE	DECIMAL(8,0)
	VARCHAR, TIME	DECIMAL(6,0)
	VARCHAR, TIMESTAMP	DECIMAL(20,6)
	DATE, DECIMAL(8,0)	DATE
	TIME, DECIMAL(6,0)	TIME
	TIMESTAMP, DECIMAL(20,6)	TIMESTAMP
	<i>datetime-type, DOUBLE, labeled-duration-code</i>	<i>datetime-type</i>
“*”	SYSIBM	Multiplies two numeric operands.
	<i>numeric-type, numeric-type</i>	<i>max numeric-type</i>
“/”	SYSIBM	Divides two numeric operands.
	<i>numeric-type, numeric-type</i>	<i>max numeric-type</i>
“  ”	SYSIBM	Same as CONCAT.

**Notes**

- References to string data types that are not qualified by a length should be assumed to support the maximum length for the data type (e.g. VARCHAR means VARCHAR(4000)).
- References to a DECIMAL data type without precision and scale should be assumed to allow any supported precision and scale.

### **Key to Table**

**any-builtin-type** Any data type that is not a distinct type.

**any-type** Any type defined to the database.

**any-structured-type** Any user-defined structured type defined to the database.

**any-comparable-type** Any type that is comparable with other argument types as defined in “Assignments and Comparisons” on page 70.

**any-union-compatible-type** Any type that is compatible with other argument types as defined in “Rules for Result Data Types” on page 82.

**character-type** Any of the character string types: CHAR, VARCHAR, LONG VARCHAR, CLOB.

**compatible-string-type** A string type that comes from the same grouping as the other argument (e.g. if one argument is a *character-type* the other must also be a *character-type*).

**datetime-type** Any of the datetime types: DATE, TIME, TIMESTAMP.

**graphic-type** Any of the double byte character string types: GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB.

**labeled-duration-code** As a type this is a SMALLINT. If the function is invoked using the infix form of the plus or minus operator, labeled-durations as defined in “Labeled Durations” on page 123 can be used. For a source function that does not use the plus or minus operator character as the name, the following values must be used for the labeled-duration-code argument when invoking the function.

- 1 YEAR or YEARS
- 2 MONTH or MONTHS
- 3 DAY or DAYS
- 4 HOUR or HOURS
- 5 MINUTE or MINUTES
- 6 SECOND or SECONDS
- 7 MICROSECOND or MICROSECONDS

**LOB-type** Any of the large object types: BLOB, CLOB, DBCLOB.

**max-numeric-type** The maximum numeric type of the arguments where maximum is defined as the rightmost *numeric-type*.

**max-string-type** The maximum string type of the arguments where maximum is defined as the rightmost *character-type* or *graphic-type*. If arguments are BLOB, the *max-string-type* is BLOB.

**numeric-type** Any of the numeric types: SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE.

**string-type** Any type from *character type*, *graphic-type* or BLOB.

### **Table Footnotes**

- 1 When the input parameter is SMALLINT, the result type is INTEGER. When the input parameter is REAL, the result type is DOUBLE.
- 2 Keywords allowed are ISO, USA, EUR, JIS, and LOCAL. This function signature is not supported as a sourced function.
- 3 This function cannot be used as a source function.
- 4 The keyword ALL or DISTINCT may be used before the first parameter. If DISTINCT is specified, use of long string types or a DATALINK type is not supported.
- 5 Use of long string types or a DATALINK type is not supported.
- 6 The type returned by RAISE\_ERROR depends upon the context of its use. RAISE\_ERROR, if not cast to a particular type, will return a type appropriate to its invocation within a CASE expression.

---

## Column Functions

The argument of a column function is a set of values derived from an expression. The expression may include columns but cannot include a *scalar-fullselect* or another column function (SQLSTATE 42607). The scope of the set is a group or an intermediate result table as explained in Chapter 5, “Queries” on page 315.

If a GROUP BY clause is specified in a query and the intermediate result from the FROM, WHERE, GROUP BY and HAVING clauses is the empty set; then the column functions are not applied, the result of the query is the empty set, the SQLCODE is set to +100 and the SQLSTATE is set to '02000'.

If a GROUP BY clause is not specified in a query and the intermediate result is of the FROM, WHERE, and HAVING clauses is the empty set, then the column functions are applied to the empty set.

For example, the result of the following SELECT statement is the number of distinct values of JOBCODE for employees in department D01:

```
SELECT COUNT(DISTINCT JOBCODE)  
FROM CORPDATA.EMPLOYEE  
WHERE WORKDEPT = 'D01'
```

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.

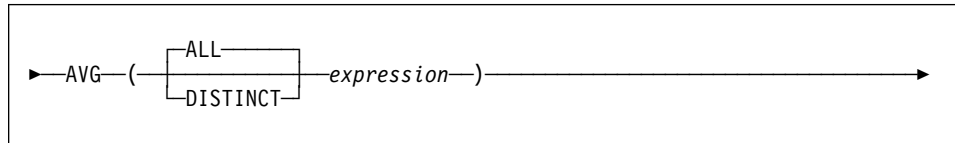
Expressions can be used in column functions, for example:

```
SELECT MAX(BONUS + 1000)  
INTO :TOP_SALESREP_BONUS  
FROM EMPLOYEE  
WHERE COMM > 5000
```

The column functions that follow are in the SYSIBM schema and may be qualified with the schema name (for example, SYSIBM.COUNT(\*)).



## AVG



The schema is SYSIBM.

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

The argument values must be numbers and their sum must be within the range of the data type of the result. The result can be null.

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.
- The result is double-precision floating point if the argument values are single-precision floating point.

If the data type of the argument values is decimal with precision  $p$  and scale  $s$ , the precision of the result is 31 and the scale is  $31-p+s$ .

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 eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the average value of the set.

If the type of the result is integer, the fractional part of the average is lost.

Examples:

- Using the PROJECT table, set the host variable AVERAGE (decimal(5,2)) to the average staffing level (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(PRSTAFF)
  INTO :AVERAGE
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

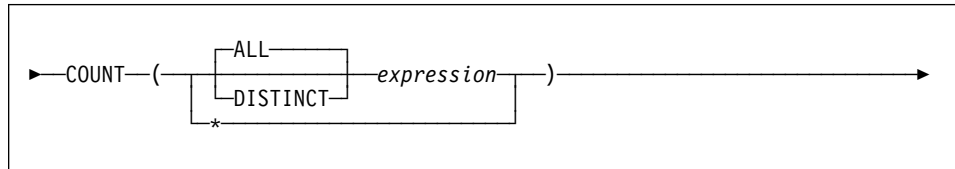
Results in AVERAGE being set to 4.25 (that is 17/4) when using the sample table.

- Using the PROJECT table, set the host variable ANY\_CALC (decimal(5,2)) to the average of each unique staffing level value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(DISTINCT PRSTAFF)
  INTO :ANY_CALC
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

Results in ANY\_CALC being set to 4.66 (that is 14/3) when using the sample table.

## COUNT



The schema is SYSIBM.

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

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

The result of the function 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. A 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 and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT(*expression*) or COUNT(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples:

- Using the EMPLOYEE table, set the host variable FEMALE (int) to the number of rows where the value of the SEX column is 'F'.

```
SELECT COUNT(*)
INTO :FEMALE
FROM EMPLOYEE
WHERE SEX = 'F'
```

Results in FEMALE being set to 13 when using the sample table.

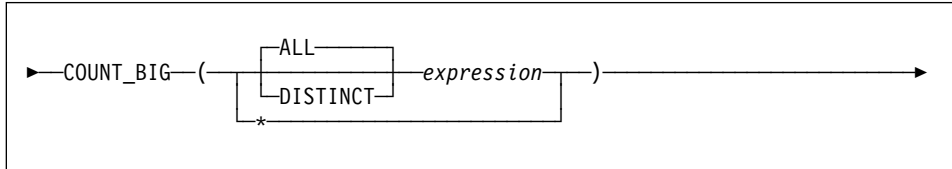
- Using the EMPLOYEE table, set the host variable FEMALE\_IN\_DEPT (int) to the number of departments (WORKDEPT) that have at least one female as a member.

```
SELECT COUNT(DISTINCT WORKDEPT)
INTO :FEMALE_IN_DEPT
FROM EMPLOYEE
WHERE SEX = 'F'
```

Results in FEMALE\_IN\_DEPT being set to 5 when using the sample table. (There is at least one female in departments A00, C01, D11, D21, and E11.)

## COUNT\_BIG

## COUNT\_BIG



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 integer.

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

The result of the function is a decimal 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 non-null values in the set.

The argument of COUNT\_BIG(*expression*) or COUNT\_BIG(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples:

- Refer to COUNT examples and substitute COUNT\_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- Some applications may require the use of COUNT but need to support values larger than the largest integer. This can be achieved by use of sourced user-defined functions and setting the SQL path. The following series of statements shows how to create a sourced function to support COUNT(\*) based on COUNT\_BIG and returning a decimal value with a precision of 15. The SQL path is set such that the sourced function based on COUNT\_BIG is used in subsequent statements such as the query shown.

```
CREATE FUNCTION RICK.COUNT() RETURNS DECIMAL(15,0)
SOURCE SYSIBM.COUNT_BIG();
SET CURRENT FUNCTION PATH RICK, SYSTEM PATH;
SELECT COUNT(*) FROM EMPLOYEE;
```

## COUNT\_BIG

Note how the sourced function is defined with no parameters to support COUNT(\*). This only works if you name the function COUNT and do not qualify the function with the schema name when it is used. To get the same effect as COUNT(\*) with a name other than COUNT, invoke the function with no parameters. Thus, if RICK.COUNT had been defined as RICK.MYCOUNT instead, the query would have to be written as follows:

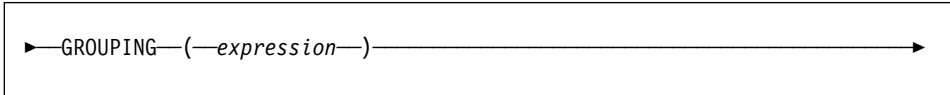
```
SELECT MYCOUNT() FROM EMPLOYEE;
```

If the count is taken on a specific column, the sourced function must specify the type of the column. The following statements created a sourced function that will take any CHAR column as a argument and use COUNT\_BIG to perform the counting.

```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE  
  SOURCE SYSIBM.COUNT_BIG(CHAR());  
SELECT COUNT(DISTINCT WORKDEPT) FROM EMPLOYEE;
```

## GROUPING

## GROUPING



The schema is SYSIBM.

Used in conjunction with grouping-sets and super-groups (see “group-by-clause” on page 329 for details), the GROUPING function returns a value which indicates whether or not a row returned in a GROUP BY answer set is a row generated by a grouping set that excludes the column represented by *expression*.

The argument can be of any type, but must be an item of a GROUP BY clause.

The result of the function is a small integer. It is set to one of the following values:

- 1 The value of *expression* in the returned row is a null value, and the row was generated by the super-group. This generated row can be used to provide sub-total values for the GROUP BY expression.
- 0 The value is other than the above.

Example:

The following query:

```
SELECT SALES_DATE,
       SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD,
       GROUPING(SALES_DATE) AS DATE_GROUP,
       GROUPING(SALES_PERSON) AS SALES_GROUP
FROM SALES
GROUP BY CUBE (SALES_DATE, SALES_PERSON)
ORDER BY SALES_DATE, SALES_PERSON
```

results in:

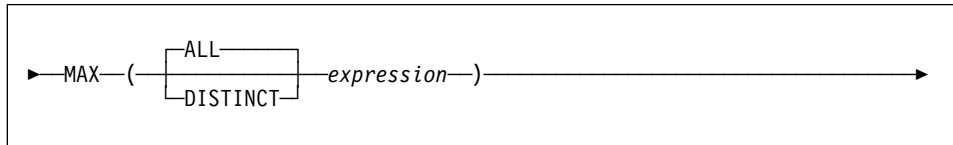
SALES_DATE	SALES_PERSON	UNITS_SOLD	DATE_GROUP	SALES_GROUP
12/31/1995	GOUNOT	1	0	0
12/31/1995	LEE	6	0	0
12/31/1995	LUCCHESSI	1	0	0
12/31/1995	-	8	0	1
03/29/1996	GOUNOT	11	0	0
03/29/1996	LEE	12	0	0
03/29/1996	LUCCHESSI	4	0	0
03/29/1996	-	27	0	1
03/30/1996	GOUNOT	21	0	0
03/30/1996	LEE	21	0	0
03/30/1996	LUCCHESSI	4	0	0
03/30/1996	-	46	0	1

## GROUPING

03/31/1996	GOUNOT	3	0	0
03/31/1996	LEE	27	0	0
03/31/1996	LUCCHESSI	1	0	0
03/31/1996	-	31	0	1
04/01/1996	GOUNOT	14	0	0
04/01/1996	LEE	25	0	0
04/01/1996	LUCCHESSI	4	0	0
04/01/1996	-	43	0	1
-	GOUNOT	50	1	0
-	LEE	91	1	0
-	LUCCHESSI	14	1	0
-	-	155	1	1

An application can recognize a SALES\_DATE sub-total row by the fact that the value of DATE\_GROUP is 0 and the value of SALES\_GROUP is 1. A SALES\_PERSON sub-total row can be recognized by the fact that the value of DATE\_GROUP is 1 and the value of SALES\_GROUP is 0. A grand total row can be recognized by the value 1 for both DATE\_GROUP and SALES\_GROUP.

## 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 type other than a long string or DATALINK.

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

The data type, length and code page of the result are the same as the data type, length and code page of the argument values. The result is considered to be a derived value and 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 a null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Examples:

- Using the EMPLOYEE table, set the host variable MAX\_SALARY (decimal(7,2)) to the maximum monthly salary (SALARY/12) value.

```
SELECT MAX(SALARY) / 12
  INTO :MAX_SALARY
  FROM EMPLOYEE
```

Results in MAX\_SALARY being set to 4395.83 when using the sample table.

- Using the PROJECT table, set the host variable LAST\_PROJ(char(24)) to the project name (PROJNAME) that comes last in the collating sequence.

```
SELECT MAX(PROJNAME)
  INTO :LAST_PROJ
  FROM PROJECT
```

Results in LAST\_PROJ being set to 'WELD LINE PLANNING' when using the sample table.

- Similar to the previous example, set the host variable LAST\_PROJ (char(40)) to the project name that comes last in the collating sequence when a project name is

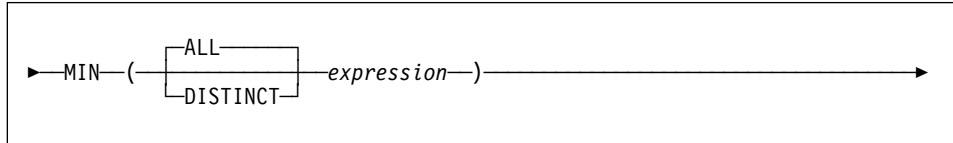


concatenated with the host variable PROJSUPP. PROJSUPP is '\_Support'; it has a char(8) data type.

```
SELECT MAX(PROJNAME CONCAT PROJSUPP)  
INTO :LAST_PROJ  
FROM PROJECT
```

Results in LAST\_PROJ being set to 'WELD LINE PLANNING\_SUPPORT' when using the sample table.

## 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 type other than a long string or DATALINK.

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

The data type, length, and code page of the result are the same as the data type, length, and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If this function is applied to an empty set, the result of the function is a null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Examples:

- Using the EMPLOYEE table, set the host variable COMM\_SPREAD (decimal(7,2)) to the difference between the maximum and minimum commission (COMM) for the members of department (WORKDEPT) 'D11'.

```
SELECT MAX(COMM) - MIN(COMM)
  INTO :COMM_SPREAD
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D11'
```

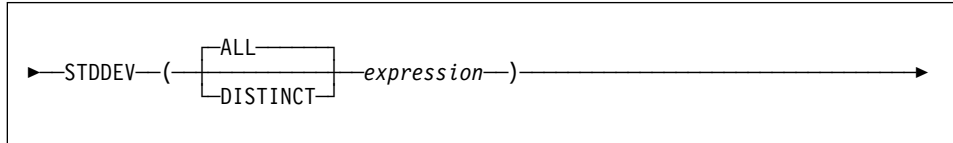
Results in COMM\_SPREAD being set to 1118 (that is, 2580 - 1462) when using the sample table.

- Using the PROJECT table, set the host variable (FIRST\_FINISHED (char(10))) to the estimated ending date (PRENDATE) of the first project scheduled to be completed.

```
SELECT MIN(PRENDATE)
  INTO :FIRST_FINISHED
  FROM PROJECT
```

Results in FIRST\_FINISHED being set to '1982-09-15' when using the sample table.

## STDDEV



The schema is SYSIBM.

The STDDEV function returns the standard deviation of a set of numbers.

The argument values must be numbers.

The data type of the result is double-precision floating point. 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 DISTINCT is specified, duplicate values are eliminated.

If the function is applied to an empty set, the result is a 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.

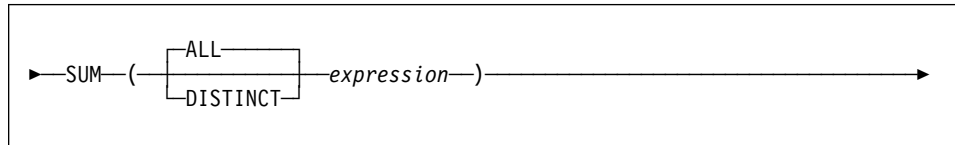
Example:

- Using the EMPLOYEE table, set the host variable DEV (double precision floating point) to the standard deviation of the salaries for those employees in department (WORKDEPT) 'A00'.

```
SELECT STDDEV(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

Results in DEV being set to approximately 9938.00 when using the sample table.

## SUM



The schema is SYSIBM.

The SUM function returns the sum of a set of numbers.

The argument values must be numbers (built-in types only) 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.
- The result is double-precision floating point if the argument values are single-precision floating point.

If the data type of the argument values is decimal, the precision of the result is 31 and the scale is the same as the scale 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 DISTINCT is specified, duplicate values are also eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the sum of the values in the set.

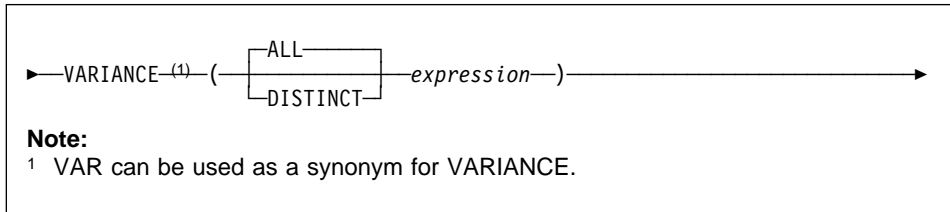
Example:

- Using the EMPLOYEE table, set the host variable JOB\_BONUS (decimal(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

```
SELECT SUM(BONUS)
INTO :JOB_BONUS
FROM EMPLOYEE
WHERE JOB = 'CLERK'
```

Results in JOB\_BONUS being set to 2800 when using the sample table.

## VARIANCE



The schema is SYSIBM.

The VARIANCE function returns the variance of a set of numbers.

The argument values must be numbers.

The data type of the result is double-precision floating point. 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 DISTINCT is specified, duplicate values are eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Example:

- Using the EMPLOYEE table, set the host variable VARNCE (double precision floating point) to the variance of the salaries for those employees in department (WORKDEPT) 'A00'.

```
SELECT VARIANCE(SALARY)
  INTO :VARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

Results in VARNCE being set to approximately 98763888.88 when using the sample table.

---

## Scalar Functions

A 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.

The restrictions on the use of column functions do not apply to scalar functions because a scalar function is applied to a single value rather than a set of values.

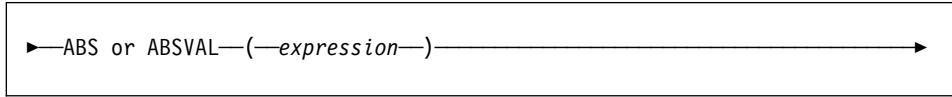
*Example:* The result of the following SELECT statement has as many rows as there are employees in department D01:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BRTHDATE)
FROM EMPLOYEE
WHERE WORKDEPT = 'D01'
```

The scalar functions that follow are in the SYSIBM schema and may be qualified with the schema name (for example, SYSIBM.CHAR(123)).

## ABS or ABSVAL

### ABS or ABSVAL



The schema is SYSFUN.

Returns the absolute value of the argument.

The argument can be of any built-in numeric data type. If it is of type DECIMAL or REAL, it is converted to a double-precision floating-point number for processing by the function.

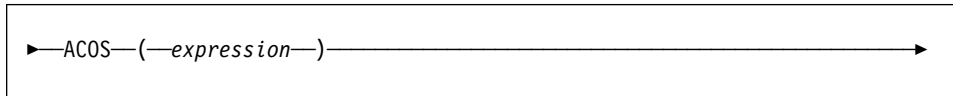
The result of the function is:

- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DOUBLE if the argument is DOUBLE, DECIMAL or REAL.

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



## ACOS



► ACOS(*expression*)

The schema is SYSFUN.

Returns the arccosine of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type. 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.

## ASCII

## ASCII

► ASCII(*expression*) ►

The schema is SYSFUN.

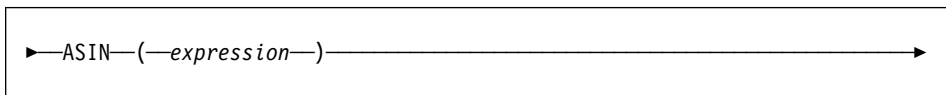
Returns the ASCII code value of the leftmost character of the argument as an integer.

The argument can be of any built-in character string type up to a maximum of 1048576 bytes (1M). LONG VARCHAR is converted to CLOB for processing by the function.

The result of the function is always INTEGER.

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

## ASIN



```
ASIN(expression)
```

The schema is SYSFUN.

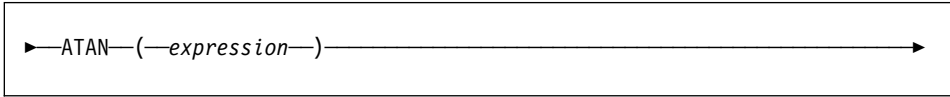
Returns the arcsine on the argument as an angle expressed in radians.

The argument can be of any built-in numeric type. 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.

## ATAN

## ATAN



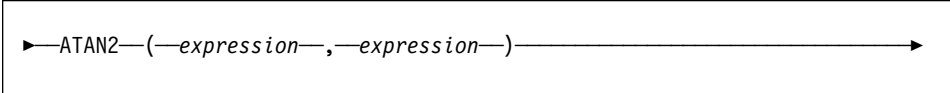
The schema is SYSFUN.

Returns the arctangent of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type. 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.

## ATAN2



▶—ATAN2—(—*expression*—,—*expression*—)————▶

The schema is SYSFUN.

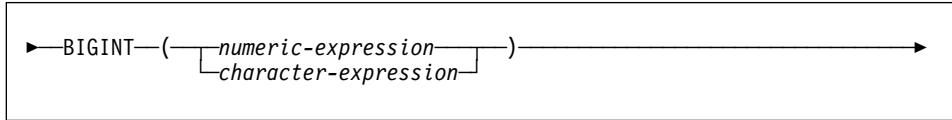
Returns the arctangent of x and y coordinates as an angle expressed in radians. The x and y coordinates are specified by the first and second arguments respectively.

The first and the second arguments can be of any built-in numeric data type. Both are 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 any argument is null, the result is the null value.

## BIGINT

### BIGINT



The schema is SYSIBM.

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

#### *numeric-expression*

An expression that returns a value of any built-in numeric data type.

If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

#### *character-expression*

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). The character string cannot be a long string.

If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a big integer column or variable.

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

Examples:

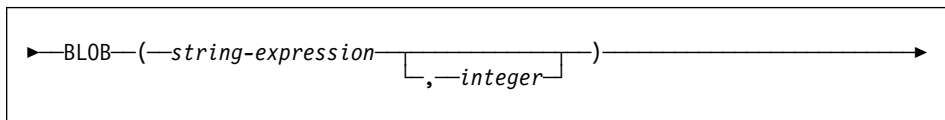
- From ORDERS\_HISTORY table, count the number of orders and return the result as a big integer value.

```
SELECT BIGINT (COUNT_BIG(*) )  
FROM ORDERS_HISTORY
```

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

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

## BLOB



The schema is SYSIBM.

The BLOB function returns a BLOB representation of a string of any type.

*string-expression*

A *string-expression* whose value can be a character string, graphic string, or a binary string.

*integer*

An integer value specifying the length attribute of the resulting BLOB data type. If *integer* is not specified, the length attribute of the result is the same as the length of the input, except where the input is graphic. In this case, the length attribute of the result is twice the length of the input.

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

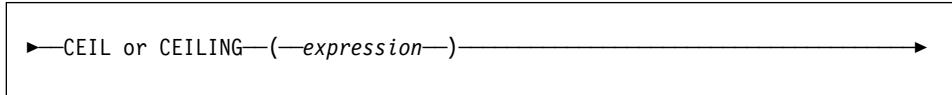
Examples

- Given a table with 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 TOPOGRAPIC_MAP LIKE BLOB('%Pellow Island%')
```

## CEIL or CEILING

### CEIL or CEILING



The schema is SYSFUN.

Returns the smallest integer value greater than or equal to the argument.

The argument can be of any built-in numeric type. If the argument is of type DECIMAL or REAL, it is converted to a double-precision floating-point number for processing by the function. If the argument is of type SMALLINT or INTEGER, the argument value is returned.

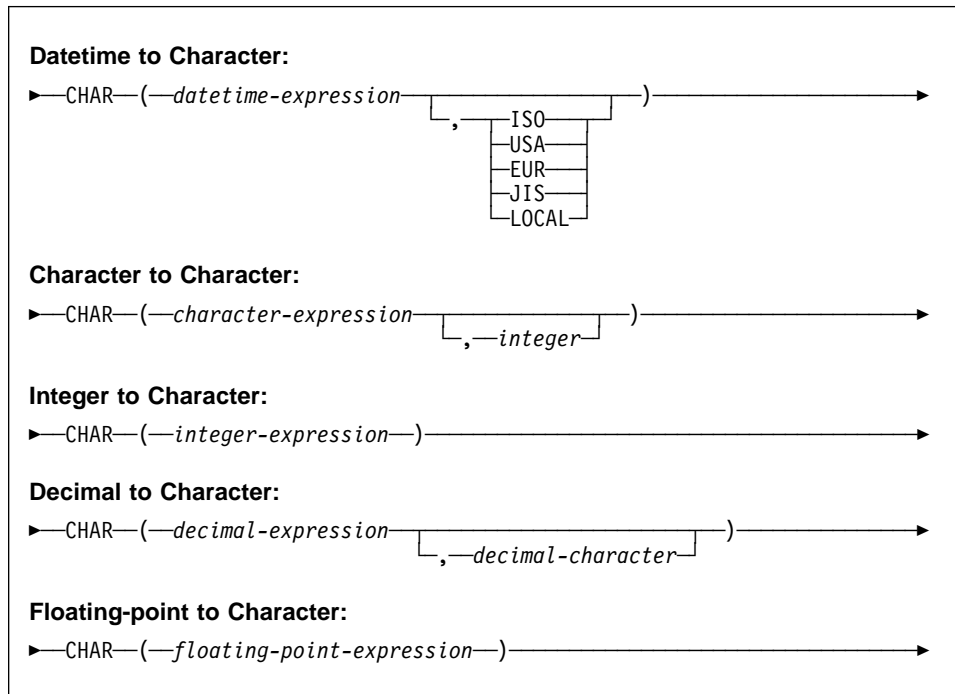
The result of the function is:

- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DOUBLE if the argument is DECIMAL, REAL or DOUBLE. Decimal values with more than 15 digits to the left of the decimal will not return the desired integer value due to loss of precision in the conversion to DOUBLE.

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



CHAR



The schema is SYSIBM. However, the schema for CHAR(*floating-point-expression*) is SYSFUN.

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

- 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 SMALLINT, INTEGER or BIGINT
- Decimal number if the first argument is a decimal number
- Double-precision floating-point number if the first argument is a DOUBLE or REAL.

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

**Datetime to Character**

*datetime-expression*

An expression that is one of the following three data types

**date** The result is the character string representation of the date in the format specified by the second argument. The length of the result is 10. An error occurs if the second

## CHAR

argument is specified and is not a valid value (SQLSTATE 42703 ).

**time** The result is the character string representation of the time in the format specified by the second argument. The length of the result is 8. An error occurs if the second argument is specified and is not a valid value (SQLSTATE 42703 ).

**timestamp** The second argument is not applicable and must not be specified. SQLSTATE 42815 The result is the character string representation of the timestamp. The length of the result is 26.

The code page of the string is the code page of the database at the application server.

### Character to Character

#### *character-expression*

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

#### *integer*

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

If the length of the character-expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the character-expression was not a long string (LONG VARCHAR or CLOB).

### Integer to Character

#### *integer-expression*

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

The result is the character string representation of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. It is left justified.

- If the first argument is a small integer:

The length of the result is 6. If the number of characters in the result is less than 6, then the result is padded on the right with blanks to length 6.

- If the first argument is a large integer:

The length of the result is 11. If the number of characters in the result is less than 11, then the result is padded on the right with blanks to length 11.

- If the first argument is a big integer:

The length of the result is 20. If the number of characters in the result is less than 20, then the result is padded on the right with blanks to length 20.

The code page of the string is the code page of the database at the application server.

### Decimal to Character

#### *decimal-expression*

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

#### *decimal-character*

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character cannot be a digit, plus ('+'), minus ('-') or blank. (SQLSTATE 42815). The default is the period ('.') character

The result is the fixed-length character-string representation of the argument. The result includes a decimal character and  $p$  digits, where  $p$  is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. The length of the result is  $2+p$ , where  $p$  is the precision of the *decimal-expression*. This means that a positive value will always include one trailing blank.

The code page of the string is the code page of the database at the application server.

### Floating-point to Character

#### *floating-point-expression*

An expression that returns a value that 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. 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 value 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, then the result is padded on the right with blanks to length 24.

The code page of the string is the code page of the database at the application server.

## CHAR

Examples:

- Assume the column PRSTDATE has an internal value equivalent to 1988-12-25.

```
CHAR(PRSTDATE, USA)
```

Results in the value '12/25/1988'.

- Assume the column STARTING has an internal value equivalent to 17.12.30, the host variable HOUR\_DUR (decimal(6,0)) is a time duration with a value of 050000. (that is, 5 hours).

```
CHAR(STARTING, USA)
```

Results in the value '5:12 PM'.

```
CHAR(STARTING + :HOUR_DUR, USA)
```

Results in the value '10:12 PM'.

- Assume the column RECEIVED (timestamp) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns.

```
CHAR(RECEIVED)
```

Results in the value '1988-12-25-17.12.30.000000'.

- Use the CHAR function to make the type fixed length character and reduce the length of the displayed results to 10 characters for the LASTNAME column (defined as VARCHAR(15)) of the EMPLOYEE table.

```
SELECT CHAR(LASTNAME,10) FROM EMPLOYEE
```

For rows having a LASTNAME with a length greater than 10 characters (excluding trailing blanks), a warning that the value is truncated is returned.

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

```
SELECT CHAR(EDLEVEL) FROM EMPLOYEE
```

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

- Assume that STAFF has a SALARY column defined as decimal with precision of 9 and scale of 2. The current value is 18357.50 and it is to be displayed with a comma as the decimal character (18357,50).

```
CHAR(SALARY, ',')
```

returns the value '00018357,50 '.

- Assume the same SALARY column subtracted from 20000.25 is to be displayed with the default decimal character.

```
CHAR(20000.25 - SALARY)
```

returns the value '-0001642.75'.

- Assume a host variable, SEASONS\_TICKETS, has an integer data type and a 10000 value.

## CHAR

**CHAR**(DECIMAL(:SEASONS\_TICKETS,7,2))

Results in the character value '10000.00 '.

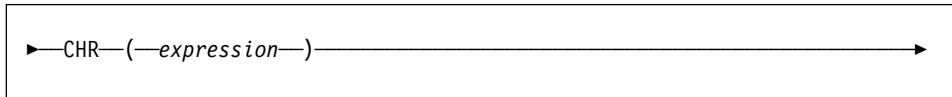
- Assume a host variable, DOUBLE\_NUM has a double data type and a value of -987.654321E-35.

**CHAR**(:DOUBLE\_NUM)

Results in the character value of '-9.87654321E-33 '. Since the result data type is CHAR(24), there are 9 trailing blanks in the result.

## CHR

## CHR



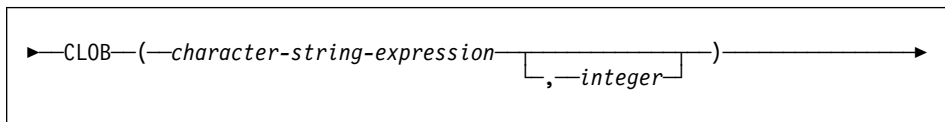
The schema is SYSFUN.

Returns the character that has the ASCII code value specified by the argument.

The argument can be either INTEGER or SMALLINT. The value of the argument should be between 0 and 255; otherwise, the return value is null.

The result of the function is CHAR(1). The result can be null; if the argument is null, the result is the null value.

## CLOB



The schema is SYSIBM.

The CLOB function returns a CLOB representation of a character string type.

*character-string-expression*

An *expression* that returns a value that is a character string.

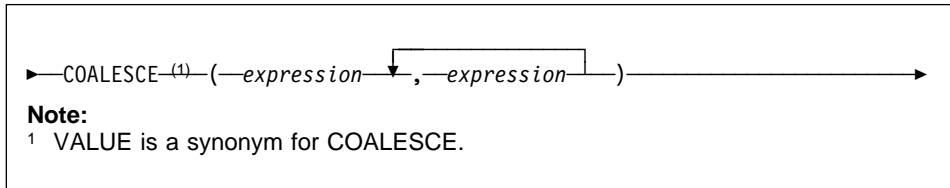
*integer*

An integer value specifying the length attribute of the resulting CLOB data type. The value must be between 0 and 2 147 483 647. If *integer* is not specified, the length of the result is the same as the length of the first argument.

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

## COALESCE

## COALESCE



The schema is SYSIBM.

COALESCE returns the first argument that is not null.

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 the arguments can be null, and the result is null only if all the arguments are null. The selected argument is converted, if necessary, to the attributes of the result.

The arguments must be compatible. See “Rules for Result Data Types” on page 82 for what data types are compatible and the attributes of the result. They can be of either a built-in or user-defined data type.<sup>33</sup>

Examples:

- When selecting all the values from all the rows in the DEPARTMENT table, if the department manager (MGRNO) is missing (that is, null), then return a value of 'ABSENT'.

```
SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'), ADMRDEPT
FROM DEPARTMENT
```

- When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is, null), then return a value of zero.

```
SELECT EMPNO, COALESCE(SALARY, 0)
FROM EMPLOYEE
```

<sup>33</sup> This function may not be used as a source function when creating a user-defined function. Since it accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.



## CONCAT

► `CONCAT`<sup>(1)</sup> (`expression1`, `expression2`)

**Note:**

<sup>1</sup> `||` may be used as a synonym for `CONCAT`.

The schema is SYSIBM.

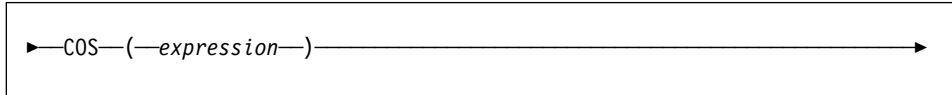
Returns the concatenation of two string arguments. The two arguments must be compatible types.

The result of the function is a string. Its length is sum of the lengths of the two arguments. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

See “With the Concatenation Operator” on page 117 for more information.

## COS

## COS



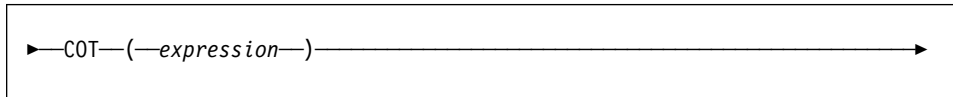
The schema is SYSFUN.

Returns the cosine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type. 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.

## COT



The schema is SYSFUN.

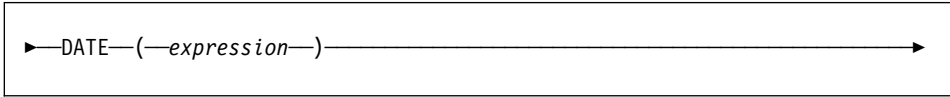
Returns the cotangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type. 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.

## DATE

## DATE



The schema is SYSIBM.

The DATE function returns a date from a value.

The argument must be a date, timestamp, a positive number less than or equal to 3652059, a valid character string representation of a date or timestamp, or a character string of length 7 that is neither a CLOB nor a LONG VARCHAR.

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 date, timestamp, or valid string representation of a date or timestamp:
  - The result is the date part of the value.
- If the argument is a number:
  - The result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.
- If the argument is a character string with a length of 7:
  - The result is the date represented by the character string.

Examples:

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

```
DATE(RECEIVED)
```

Results in an internal representation of '1988-12-25'.

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

```
DATE('1988-12-25')
```

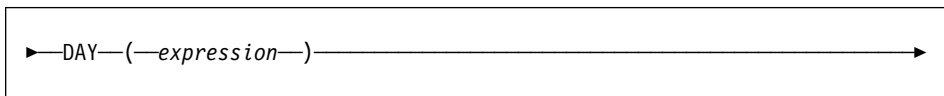
- This example results in an internal representation of '1988-12-25'.

```
DATE('25.12.1988')
```

- This example results in an internal representation of '0001-02-04'.

```
DATE(35)
```

## DAY



The schema is SYSIBM.

The DAY function returns the day part of a value.

The argument must be a date, timestamp, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The other rules depend on the data type of the argument:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp:
  - The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a 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.

Examples:

- Using the PROJECT table, set the host variable END\_DAY (smallint) to the day that the WELD LINE PLANNING project (PROJNAME) is scheduled to stop (PRENDATE).

```
SELECT DAY(PRENDATE)
  INTO :END_DAY
  FROM PROJECT
  WHERE PROJNAME = 'WELD LINE PLANNING'
```

Results in END\_DAY being set to 15 when using the sample table.

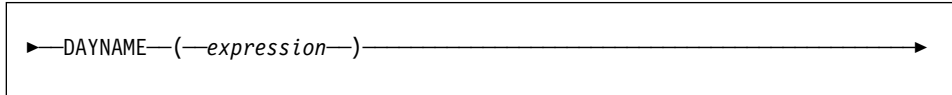
- Assume that the column DATE1 (date) has an internal value equivalent to 2000-03-15 and the column DATE2 (date) has an internal value equivalent to 1999-12-31.

```
DAY(DATE1 - DATE2)
```

Results in the value 15.

## DAYNAME

## DAYNAME



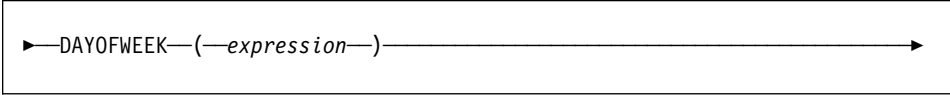
The schema is SYSFUN.

Returns a mixed case character string containing the name of the day (e.g. Friday) for the day portion of the argument based on the locale when the database was started.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

## DAYOFWEEK



► DAYOFWEEK(*expression*) →

The schema is SYSFUN.

Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## DAYOFYEAR

### DAYOFYEAR

► DAYOFYEAR(*expression*)

The schema is SYSFUN.

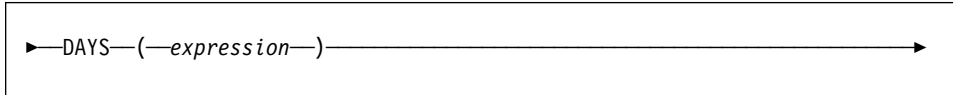
Returns the day of the year in the argument as an integer value in the range 1-366.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.



DAYS



The schema is SYSIBM.

The DAYS function returns an integer representation of a date.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The 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.

Examples:

- Using the PROJECT table, set the host variable EDUCATION\_DAYS (int) to the number of elapsed days (PRENDATE - PRSTDATE) estimated for the project (PROJNO) 'IF2000'.

```
SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
  INTO :EDUCATION_DAYS
  FROM PROJECT
  WHERE PROJNO = 'IF2000'
```

Results in EDUCATION\_DAYS being set to 396 when using the sample table.

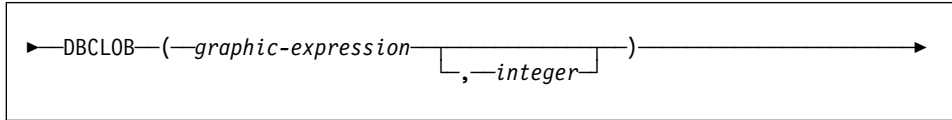
- Using the PROJECT table, set the host variable TOTAL\_DAYS (int) to the sum of elapsed days (PRENDATE - PRSTDATE) estimated for all projects in department (DEPTNO) 'E21'.

```
SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
  INTO :TOTAL_DAYS
  FROM PROJECT
  WHERE DEPTNO = 'E21'
```

Results in TOTAL\_DAYS being set to 1584 when using the sample table.

## DBCLOB

## DBCLOB



The schema is SYSIBM.

The DBCLOB function returns a DBCLOB representation of a graphic string type.

*graphic-expression*

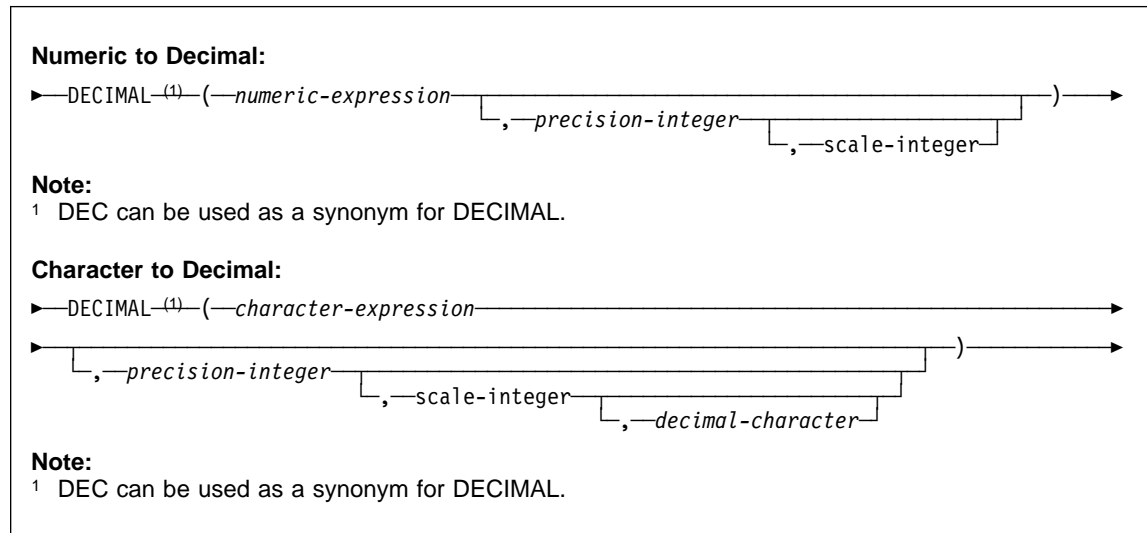
An *expression* that returns a value that is a graphic string.

*integer*

An integer value specifying the length attribute of the resulting DBCLOB data type. The value must be between 0 and 1 073 741 823. If *integer* is not specified, the length of the result is the same as the length of the first argument.

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

## DECIMAL



The schema is SYSIBM.

The DECIMAL function returns a decimal representation of

- A number
- A character string representation of a decimal number
- A character string representation of a integer number.

The result of the function is a decimal number with precision of  $p$  and scale of  $s$ , where  $p$  and  $s$  are the second and third arguments. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

#### Numeric to Decimal

##### *numeric-expression*

An expression that returns a value of any numeric data type.

##### *precision-integer*

An integer constant with a value in the range of 1 to 31.

The default for the *precision-integer* depends on the data type of the *numeric-expression*:

- 15 for floating-point and decimal
- 19 for big integer
- 11 for large integer
- 5 for small integer.

##### *scale-integer*

An integer constant in the range of 0 to the *precision-integer* value. The default is zero.

## DECIMAL

The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of  $p$  and a scale of  $s$ , where  $p$  and  $s$  are the second and third arguments. An error occurs if the number of significant decimal digits required to represent the whole part of the number is greater than  $p-s$ .

### Character to Decimal

#### *character-expression*

An *expression* that returns a value that is a character string with a length not greater than the maximum length of a character constant (4 000 bytes). It cannot have a CLOB or LONG VARCHAR data type. Leading and trailing blanks are eliminated from the string. The resulting substring must conform to the rules for forming an SQL integer or decimal constant (SQLSTATE 22018).

The *character-expression* is converted to the database code page if required to match the code page of the constant *decimal-character*.

#### *precision-integer*

An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default is 15.

#### *scale-integer*

An integer constant with a value in the range 0 to *precision-integer* that specifies the scale of the result. If not specified, the default is 0.

#### *decimal-character*

Specifies the single byte character constant that is used to delimit the decimal digits in *character-expression* from the whole part of the number. The character cannot be a digit plus ('+') , minus ('-') or blank and can appear at most once in *character-expression* (SQLSTATE 42815).

The result is a decimal number with precision  $p$  and scale  $s$  where  $p$  and  $s$  are the second and third arguments. Digits are truncated from the end if the number of digits right of the decimal character is greater than the scale  $s$ . An error occurs if the number of significant digits left of the decimal character (the whole part of the number) in *character-expression* is greater than  $p-s$  (SQLSTATE 22003). The default decimal character is not valid in the substring if the *decimal-character* argument is specified (SQLSTATE 22018).

### Examples:

- Use the DECIMAL function in order to force a DECIMAL data type (with a precision of 5 and a scale of 2) to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECIMAL(EDLEVEL,5,2)
FROM EMPLOYEE
```

## DECIMAL

- Assume the host variable PERIOD is of type INTEGER. Then, in order to use its value as a date duration it must be "cast" as decimal(8,0).

```
SELECT PRSTDATE + DECIMAL(:PERIOD,8)
FROM PROJECT
```

- Assume that updates to the SALARY column are input through a window as a character string using comma as a decimal character (for example, the user inputs 21400,50). Once validated by the application, it is assigned to the host variable newsalary which is defined as CHAR(10).

```
UPDATE STAFF
SET SALARY = DECIMAL(:newsalary, 9, 2, ',')
WHERE ID = :empid;
```

The value of newsalary becomes 21400.50.

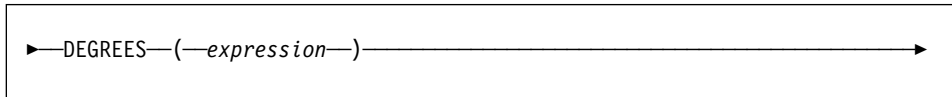
- Add the default decimal character (.) to a value.

```
DECIMAL('21400,50', 9, 2, '.')
```

This fails because a period (.) is specified as the decimal character but a comma (,) appears in the first argument as a delimiter.

## DEGREES

## DEGREES



The schema is SYSFUN.

Returns the number of degrees converted from the argument expressed in radians.

The argument can be of any built-in numeric type. It is converted to 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.

**DEREF**

► **DEREF**—(*expression*)—►

The schema is SYSIBM.

The Deref function returns an instance of the target type of the argument.

The argument can be any value with a reference data type that has a defined scope (SQLSTATE 428DT).

The static data type of the result is the target type of the argument. The dynamic data type of the result is a subtype of the target type of the argument. The result can be null. The result is the null value if *expression* is a null value or if *expression* is a reference that has no matching OID in the target table.

The result is an instance of the subtype of the target type of the reference. The result is determined by finding the row of the target table or target view of the reference that has an object identifier that matches the reference value. The type of this row determines the dynamic type of the result. Since the type of the result can be based on a row of a subtable or subview of the target table or target view, the authorization ID of the statement must have SELECT privilege on the target table and all of its subtables or the target view and all of its subviews (SQLSTATE 42501).

This function can only be used on the left side of the TYPE predicate or in the argument of the TYPE\_ID, TYPE\_NAME and TYPE\_SCHEMA functions.

Examples:

See Examples section in "TYPE\_NAME" on page 303.

## DIFFERENCE

## DIFFERENCE

→DIFFERENCE(—*expression*—,—*expression*—)→

The schema is SYSFUN.

Returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.

The arguments can be character strings that are either CHAR or VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example:

```
VALUES (DIFFERENCE('CONSTRAINT','CONSTANT'),SOUNDEX('CONSTRAINT'),
        SOUNDEX('CONSTANT')),
        (DIFFERENCE('CONSTRAINT','CONTRITE'),SOUNDEX('CONSTRAINT'),
        SOUNDEX('CONTRITE'))
```

This example returns the following.

```
1          2      3
-----
4 C523 C523
2 C523 C536
```

In the first row, the words have the same result from SOUNDEX while in the second row the words have only some similarity.



## DIGITS

The schema is SYSIBM.

The DIGITS function returns a character-string representation of a number.

The argument must be an expression that returns a value of type SMALLINT, INTEGER, BIGINT or DECIMAL.

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 character. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- $p$  if the argument is a decimal number with a precision of  $p$ .

Examples:

- Assume that a table called TABLEX contains an INTEGER column called INTCOL containing 10-digit numbers. List all distinct four digit combinations of the first four digits contained in column INTCOL.

```
SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
FROM TABLEX
```

- Assume that COLUMNX has the DECIMAL(6,2) data type, and that one of its values is -6.28. Then, for this value:

```
DIGITS(COLUMNX)
```

returns the value '000628'.

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.

## DLCOMMENT

### DLCOMMENT

► `DLCOMMENT`—(*data link expression*)—►

The schema is SYSIBM.

The `DLCOMMENT` function returns the comment value, if it exists, from a `DATALINK` value.

The argument must be an expression that results in a value with data type of `DATALINK`.

The result of the function is `VARCHAR(254)`. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

- Prepare a statement to select the date, the description and the comment from the link to the `ARTICLES` column from the `HOCKEY_GOALS` table. The rows to be selected are those for goals scored by either of the Richard brothers (Maurice or Henri).

```
stmtvar = "SELECT DATE_OF_GOAL, DESCRIPTION, DLCOMMENT(ARTICLES)
          FROM HOCKEY_GOALS
          WHERE BY_PLAYER = 'Maurice Richard' OR BY_PLAYER = 'Henri Richard' ";
EXEC SQL PREPARE HOCKEY_STMT FROM :stmtvar;
```

- Given a `DATALINK` value that was inserted into column `COLA` of a row in table `TBLA` using the scalar function:

```
DLVALUE('http://d1fs.a1maden.ibm.com/x/y/a.b','URL','A comment')
```

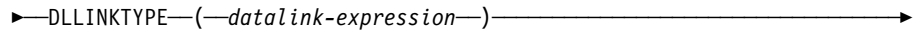
then the following function operating on that value:

```
DLCOMMENT(COLA)
```

will return the value:

```
A comment
```

**DLLINKTYPE**



The schema is SYSIBM.

The DLLINKTYPE function returns the linktype value from a DATALINK value.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(4). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
DLVALUE('http://d1fs.almaden.ibm.com/x/y/a.b','URL','a comment')
```

then the following function operating on that value:

```
DLLINKTYPE(COLA)
```

will return the value:

```
URL
```

## DLURLCOMPLETE

### DLURLCOMPLETE

►DLURLCOMPLETE—(*—datalink-expression—*)—————►

The schema is SYSIBM.

The DLURLCOMPLETE function returns the complete URL value from a DATALINK value with a link type of URL. The value is the same as what would be returned by the concatenation of DLURLSCHEME with '://', then DLURLSERVER, then '/' and then DLURLPATH. When appropriate, the value includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
```

then the following function operating on that value:

```
DLURLCOMPLETE(COLA)
```

will return the value:

```
HTTP://DLFS.ALMA DEN.IBM.COM/x/y/*****;a.b
```

(where \*\*\*\*\* represents the access token)

**DLURLPATH**

►DLURLPATH(—*datalink-expression*—)—————►

The schema is SYSIBM.

The DLURLPATH function returns the path and file name necessary to access a file within a given server from a DATALINK value with a linktype of URL. When appropriate, the value includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
DLVALUE('http://dfs.almaden.ibm.com/x/y/a.b','URL','a comment')
```

then the following function operating on that value:

```
DLURLPATH(COLA)
```

will return the value:

```
/x/y/*****;a.b
```

(where \*\*\*\*\* represents the access token)

## DLURLPATHONLY

### DLURLPATHONLY

►DLURLPATHONLY(—*datalink-expression*—)—————►

The schema is SYSIBM.

The DLURLPATHONLY function returns the path and file name necessary to access a file within a given server from a DATALINK value with a linktype of URL. The value returned NEVER includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
DLVALUE('http://d1fs.a1maden.ibm.com/x/y/a.b','URL','a comment')
```

then the following function operating on that value:

```
DLURLPATHONLY(COLA)
```

will return the value:

```
/x/y/a.b
```

## DLURLSCHEME

►DLURLSCHEME—(*data link-expression*)—►

The schema is SYSIBM.

The DLURLSCHEME function returns the scheme from a DATALINK value with a linktype of URL. The value will always be in upper case.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(20). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
DLVALUE('http://d1fs.almaden.ibm.com/x/y/a.b','URL','a comment')
```

then the following function operating on that value:

```
DLURLSCHEME(COLA)
```

will return the value:

```
HTTP
```

## DLURLSERVER

### DLURLSERVER

►DLURLSERVER(*—datalink-expression—*)►

The schema is SYSIBM.

The DLURLSERVER function returns the file server from a DATALINK value with a linktype of URL. The value will always be in upper case.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
```

then the following function operating on that value:

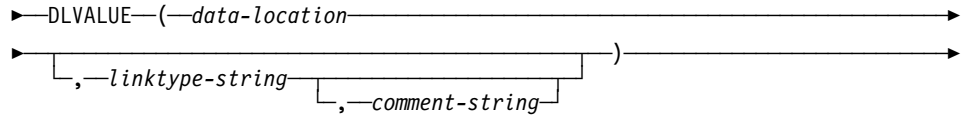
```
DLURLSERVER(COLA)
```

will return the value:

```
DLFS.ALMADEN.IBM.COM
```



## DLVALUE



The schema is SYSIBM.

The DLVALUE function returns a DATALINK value. When the function is on the right hand side of a SET clause in an UPDATE statement or is in a VALUES clause in an INSERT statement, it usually also creates a link to a file. However, if only a comment is specified (in which case the data-location is a zero-length string), the DATALINK value is created with empty linkage attributes so there is no file link.

*data-location*

If the link type is URL, then this is an expression that yields a varying length character string containing a complete URL value. If the expression is not an empty string and does not include the URL scheme and URL server, then the defaults of scheme "FILE" and the server name of the database server are included in the complete URL value.

*linktype-string*

An optional VARCHAR expression that specifies the link type of the DATALINK value. The only valid value is 'URL' (SQLSTATE 428D1).

*comment-string*

An optional VARCHAR(254) value that provides a comment or additional location information.

The result of the function is a DATALINK value. If any argument of the DLVALUE function can be null, the result can be null; If the *data-location* is null, the result is the null value.

When defining a DATALINK value using this function, consider the maximum length of the target of the value. For example, if a column is defined as DATALINK(200), then the maximum length of the *data-location* plus the *comment* is 200 bytes.

Example:

- Insert a row into the table. The URL values for the first two links are contained in the variables named url\_article and url\_snapshot. The variable named url\_snapshot\_comment contains a comment to accompany the snapshot link. There is, as yet, no link for the movie, only a comment in the variable named url\_movie\_comment.

## DLVALUE

```
EXEC SQL INSERT INTO HOCKEY_GOALS
      VALUES('Maurice Richard',
             'Montreal Canadien',
             '?',
             'Boston Bruins',
             '1952-04-24',
             'Winning goal in game 7 of Stanley Cup final',
             DLVALUE(:url_article),
             DLVALUE(:url_snapshot, 'URL', :url_snapshot_comment),
             DLVALUE(' ', 'URL', :url_movie_comment) );
```

## DOUBLE

**Numeric to Double:**

►DOUBLE<sup>(1)</sup>(*numeric-expression*)

**Note:**

<sup>1</sup> FLOAT or DOUBLE\_PRECISION can be used as a synonym for the DOUBLE function in the SYSIBM schema.

**Character String to Double:**

►DOUBLE(*string-expression*)

The schema is SYSIBM. However, the schema for DOUBLE(string-expression) is SYSFUN.

The DOUBLE function returns a floating-point number corresponding to a:

- number if the argument is a numeric expression
- character string representation of a number if the argument is a string expression.

**Numeric to Double***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 double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable.

**Character String to Double***string-expression*

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

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

The result is the same number that would occur if the string was considered a constant and assigned to a double-precision floating-point column or variable.

Example:

## DOUBLE

Using the EMPLOYEE table, 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 eliminate the possibility of out-of-range results, DOUBLE is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, DOUBLE(SALARY)/COMM
FROM EMPLOYEE
WHERE COMM > 0
```

## EVENT\_MON\_STATE

►—EVENT\_MON\_STATE—(—*string-expression*—)—————►

The schema is SYSIBM.

The EVENT\_MON\_STATE function returns the current state of an event monitor.

The argument is a string expression with a resulting type of CHAR or VARCHAR and a value that is the name of an event monitor. If the named event monitor does not exist in the SYSCAT.EVENTMONITORS catalog table, SQLSTATE 42704 will be returned.

The result is an integer with one of the following values:

- 0 The event monitor is inactive.
- 1 The event monitor is active.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

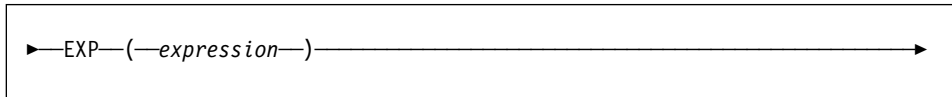
Example:

- The following example selects all of the defined event monitors, and indicates whether each is active or inactive:

```
SELECT EVMONNAME,
       CASE
         WHEN EVENT_MON_STATE(EVMONNAME) = 0 THEN 'Inactive'
         WHEN EVENT_MON_STATE(EVMONNAME) = 1 THEN 'Active'
       END
FROM SYSCAT.EVENTMONITORS
```

## EXP

## EXP



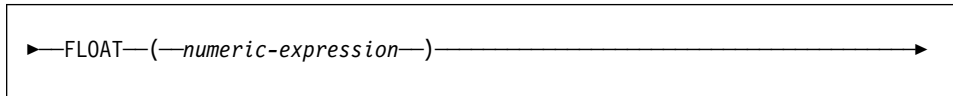
The schema is SYSFUN.

Returns the exponential function of the argument.

The argument can be of any built-in numeric data type. It is converted to 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.

## FLOAT



▶—FLOAT—(*numeric-expression*)—▶

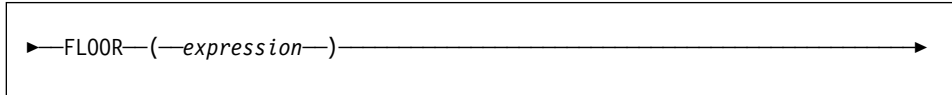
The schema is SYSIBM.

The FLOAT function returns a floating-point representation of a number.

FLOAT is a synonym for DOUBLE. See “DOUBLE” on page 229 for details.

## FLOOR

## FLOOR



The schema is SYSFUN.

Returns the largest integer value less than or equal to the argument.

The argument can be of any built-in numeric type. If the argument is of type DECIMAL or REAL, it is converted to a double-precision floating-point number for processing by the function. If the argument is of type SMALLINT, INTEGER or BIGINT the argument value is returned.

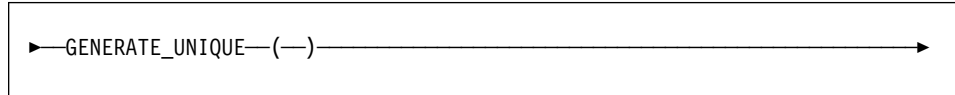
The result of the function is:

- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DOUBLE if the argument is DOUBLE, DECIMAL or REAL. Decimal values with more than 15 digits to the left of the decimal will not return the desired integer value due to loss of precision in the conversion to DOUBLE.

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



## GENERATE\_UNIQUE



The schema is SYSIBM.

The GENERATE\_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function.<sup>34</sup>

There are no arguments to this function (the empty parentheses must be specified).

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the partition number where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The value includes the partition number where the function executed so that a table partitioned across multiple partitions also has unique values in some sequence. The sequence is based on the time the function was executed.

This function differs from using the special register CURRENT\_TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement or an insert statement with a fullselect.

The timestamp value that is part of the result of this function can be determined using the TIMESTAMP scalar function with the result of GENERATE\_UNIQUE as an argument.

Examples:

- Create a table that includes a column that is unique for each row. Populate this column using the GENERATE\_UNIQUE function. Notice that the UNIQUE\_ID column has "FOR BIT DATA" specified to identify the column as a bit data character string.

<sup>34</sup> The system clock is used to generate the internal Universal Time, Coordinated (UTC) timestamp along with the partition number on which the function executes. Adjustments that move the actual system clock backward could result in duplicate values.

## GENERATE\_UNIQUE

```
CREATE TABLE EMP_UPDATE
(UNIQUE_ID CHAR(13) FOR BIT DATA,
                                EMPNO CHAR(6),
                                TEXT VARCHAR(1000))
```

```
INSERT INTO EMP_UPDATE
VALUES (GENERATE_UNIQUE(), '000020', 'Update entry...'),
(GENERATE_UNIQUE(), '000050', 'Update entry...')
```

This table will have a unique identifier for each row provided that the UNIQUE\_ID column is always set using GENERATE\_UNIQUE. This can be done by introducing a trigger on the table.

```
CREATE TRIGGER EMP_UPDATE_UNIQUE
NO CASCADE BEFORE INSERT ON EMP_UPDATE
REFERENCING NEW AS NEW_UPD
FOR EACH ROW MODE DB2SQL
SET NEW_UPD.UNIQUE_ID = GENERATE_UNIQUE()
```

With this trigger defined, the previous INSERT statement could be issued without the first column as follows.

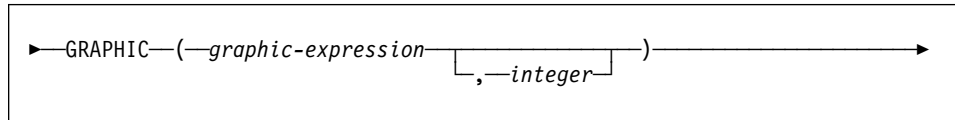
```
INSERT INTO EMP_UPDATE (EMPNO, TEXT)
VALUES ('000020', 'Update entry 1...'),
('000050', 'Update entry 2...')
```

The timestamp (in UTC) for when a row was added to EMP\_UPDATE can be returned using:

```
SELECT TIMESTAMP (UNIQUE_ID), EMPNO, TEXT FROM EMP_UPDATE
```

Therefore, there is no need to have a timestamp column in the table to record when a row is inserted.

## GRAPHIC



The schema is SYSIBM.

The GRAPHIC function returns a GRAPHIC representation of a graphic string type.

*graphic-expression*

An *expression* that returns a value that is a graphic string.

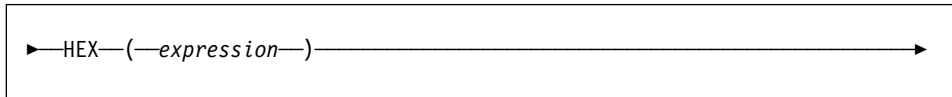
*integer*

An integer value specifying the length attribute of the resulting GRAPHIC data type. The value must be between 1 and 127. If *integer* is not specified, the length of the result is the same as the length of the first argument.

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

## HEX

## HEX



The schema is SYSIBM.

The HEX function returns a hexadecimal representation of a value as a character string.

The argument can be an expression that is a value of any built-in data type with a maximum length of 2000 bytes.

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 code page is the database code page.

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 or a numeric value the result is the hexadecimal representation of the internal form of the argument. The hexadecimal representation that is returned may be different depending on the application server where the function is executed. Cases where differences would be evident include:

- Character string arguments when the HEX function is performed on an ASCII client with an EBCDIC server or on an EBCDIC client with an ASCII server.
- Numeric arguments (in some cases) when the HEX function is performed where client and server systems have different byte orderings for numeric values.

The type and length of the result vary based on the type and length of character string arguments.

- Character string
  - Fixed length not greater than 127
    - Result is a character string of fixed length twice the defined length of the argument.
  - Fixed length greater than 127
    - Result is a character string of varying length twice the defined length of the argument.
  - Varying length
    - Result is a character string of varying length with maximum length twice the defined maximum length of the argument.
- Graphic string

- Fixed length not greater than 63
  - Result is a character string of fixed length four times the defined length of the argument.
- Fixed length greater than 63
  - Result is a character string of varying length four times the defined length of the argument.
- Varying length
  - Result is a character string of varying length with maximum length four times the defined maximum length of the argument.

Examples:

Assume the use of a DB2 for AIX application server for the following examples.

- Using the DEPARTMENT table set the host variable HEX\_MGRNO (char(12)) to the hexadecimal representation of the manager number (MGRNO) for the 'PLANNING' department (DEPTNAME).

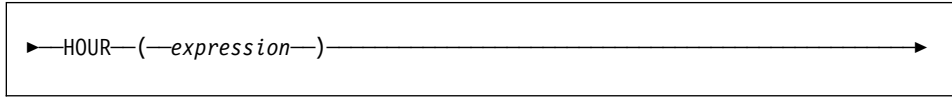
```
SELECT HEX(MGRNO)
INTO :HEX_MGRNO FROM DEPARTMENT WHERE DEPTNAME = 'PLANNING'
```

HEX\_MGRNO will be set to '303030303230' when using the sample table (character value is '000020').

- Suppose COL\_1 is a column with a data type of char(1) and a value of 'B'. The hexadecimal representation of the letter 'B' is X'42'. HEX(COL\_1) returns a two-character string '42'.
- Suppose COL\_3 is a column with a data type of decimal(6,2) and a value of 40.1. An eight-character string '0004010C' is the result of applying the HEX function to the internal representation of the decimal value, 40.1.

## HOUR

## HOUR



The schema is SYSIBM.

The HOUR function returns the hour part of a value.

The argument must be a time, timestamp, time duration, timestamp duration or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The other rules depend on the data type of the argument:

- If the argument is a time, timestamp or valid string representation of a time or timestamp:
  - The result is the hour part of the value, which is an integer between 0 and 24.
- If the argument is a time duration or timestamp duration:
  - The result is the hour part of the value, which is an integer between –99 and 99. A nonzero result has the same sign as the argument.

Example:

Using the CL\_SCHED sample table, select all the classes that start in the afternoon.

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

## INSERT

```

▶—INSERT—(—expression1—,—expression2—,——————▶
▶—expression3—,—expression4—)—————▶

```

The schema is SYSFUN.

Returns a string where expression3 bytes have been deleted from expression1 beginning at expression2 and where expression4 has been inserted into expression1 beginning at expression2. If the length of the result string exceeds the maximum for the return type, an error occurs (SQLSTATE 38552).

The first argument is a character string or a binary string with a maximum length of 1048576 bytes. The second and third arguments must be a numeric value with a data type of SMALLINT or INTEGER. If the first argument is a character string, then the fourth argument must also be a character string with a maximum length of 1048576 bytes. If the first argument is a binary string, then the fourth argument must be a binary string with a maximum length of 1048576 bytes. For the first and fourth arguments, CHAR is converted to VARCHAR and LONG VARCHAR to CLOB(1M), for second and third arguments SMALLINT is converted to INTEGER for processing by the function.

The result is based on the argument types as follows:

- VARCHAR(4000) if both the first and fourth arguments are VARCHAR or CHAR
- CLOB(1M) if either the first or fourth argument is CLOB or LONG VARCHAR
- BLOB(1M) if both first and fourth arguments are BLOB.

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

Example:

- Delete one character from the word 'DINING' and insert 'VID', both beginning at the third character.

```
VALUES CHAR(INSERT('DINING', 3, 1, 'VID'), 10)
```

This example returns the following:

```

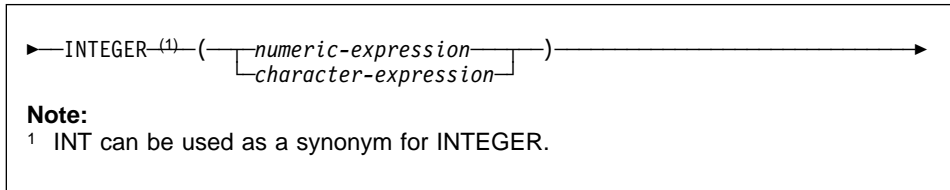
1
-----
DIVIDING

```

As mentioned, the output of the INSERT function is VARCHAR(4000). For the above example the function CHAR has been used to limit the output of INSERT to 10 bytes. The starting location of a particular string can be found using LOCATE. Refer to "LOCATE" on page 248 for more information.

## INTEGER

## INTEGER



The schema is SYSIBM.

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

### *numeric-expression*

An expression that returns a value of any built-in numeric data type.

If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

### *character-expression*

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). The character string cannot be a long string.

If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a large integer column or variable.

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

Examples:

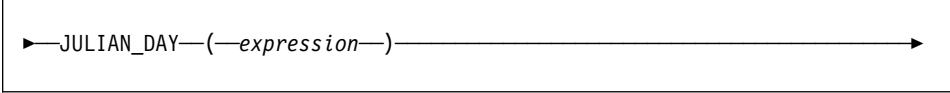
- Using the `EMPLOYEE` table, select a list containing salary (`SALARY`) divided by education level (`EDLEVEL`). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (`EMPNO`). The list should be in descending order of the calculated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

```
SELECT INTEGER (SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
FROM EMPLOYEE
ORDER BY 1 DESC
```

- Using the `EMPLOYEE` table, select the `EMPNO` column in integer form for further processing in the application.

```
SELECT INTEGER(EMPNO) FROM EMPLOYEE
```



**JULIAN\_DAY**

► `JULIAN_DAY` (`-expression-`) ►

The schema is SYSFUN.

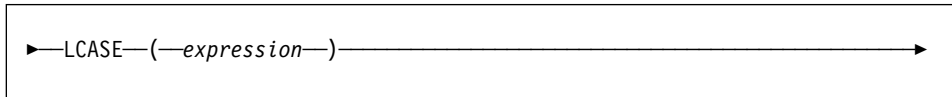
Returns an integer value representing the number of days from January 1,4712 B.C. (the start of Julian date calendar) to the date value specified in the argument.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## LCASE

## LCASE



The schema is SYSFUN.

Returns a string in which all the characters A-Z have been converted to the characters a-z (characters with diacritical marks are not converted). Note that LCASE(UCASE(string)) will therefore not necessarily return the same result as LCASE(string).

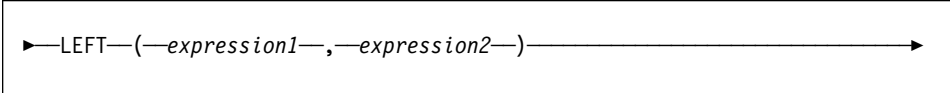
The argument can be of any built-in character string type up to a maximum of 1048576 bytes (1M).

The result of the function is:

- VARCHAR(4000) if the argument is VARCHAR or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR

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

## LEFT



```
▶LEFT(expression1, expression2)▶
```

The schema is SYSFUN.

Returns a string consisting of the leftmost expression2 bytes in expression1.

The first argument is a character string or binary string with maximum length of 1048576 bytes. The second argument must be of INTEGER or SMALLINT datatype.

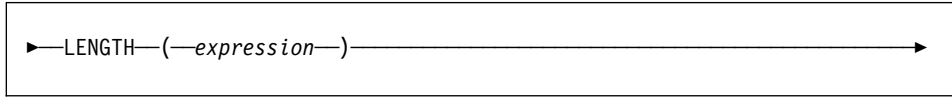
The result of the function is:

- VARCHAR(4000) if the argument is VARCHAR or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR
- BLOB(1M) if the argument is BLOB.

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

## LENGTH

## LENGTH



The schema is SYSIBM.

The LENGTH function returns the length of a value.

The argument can be 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 DBCS characters. The length of all other values is the number of bytes used to represent the value:

- 2 for small integer
- 4 for large integer
- $(p/2)+1$  for decimal numbers with precision  $p$
- The length of the string for binary strings
- The length of the string for character strings
- 4 for single-precision floating-point
- 8 for double-precision floating-point
- 4 for date
- 3 for time
- 10 for timestamp

Examples:

- Assume the host variable ADDRESS is a varying length character string with a value of '895 Don Mills Road'.

```
LENGTH(:ADDRESS)
```

Returns the value 18.

- Assume that START\_DATE is a column of type DATE.

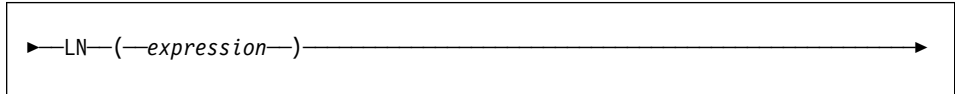
```
LENGTH(START_DATE)
```

Returns the value 4.

- This example returns the value 10.

```
LENGTH(CHAR(START_DATE, EUR))
```

## LN



► LN(-*expression*-) →

The schema is SYSFUN.

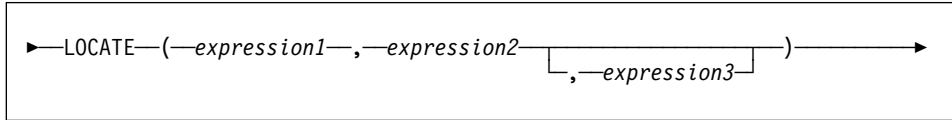
Returns the natural logarithm of the argument (same as LOG).

The argument can be of any built-in numeric data type. It is converted to 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.

## LOCATE

## LOCATE



The schema is SYSFUN.

Returns the starting position of the first occurrence of expression1 within expression2. If the optional expression3 is specified, it indicates the character position in expression2 at which the search is to begin. If expression1 is not found within expression2, the value 0 is returned.

If the first argument is a character string, then the second argument must be a character string with a maximum length of 1048576 bytes. If the first argument is a binary string, then the second argument must be a binary string with a maximum length of 1048576 bytes. The third argument must be is INTEGER or SMALLINT.

The result of the function is INTEGER. The result can be null; if any argument is null, the result is the null value.

Example:

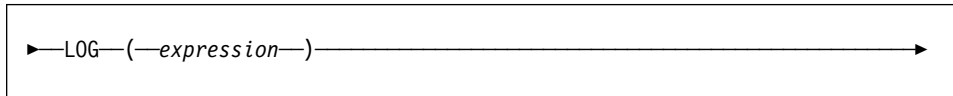
- Find the location of the letter 'N' (first occurrence) in the word 'DINING'.

**VALUES LOCATE ('N', 'DINING')**

This example returns the following:

```
1
-----
3
```

## LOG



► LOG(*expression*)

The schema is SYSFUN.

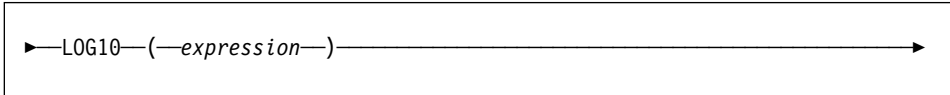
Returns the natural logarithm of the argument (same as LN).

The argument can be of any built-in numeric data type. It is converted to 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.

## LOG10

## LOG10



The schema is SYSFUN.

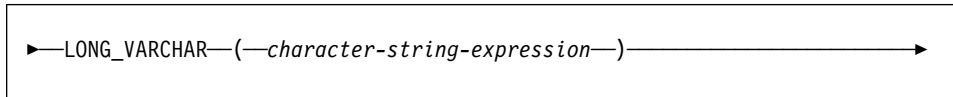
Returns the base 10 logarithm of the argument.

The argument can be of any built-in numeric type. It is converted to 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.



## LONG\_VARCHAR



The schema is SYSIBM.

The LONG\_VARCHAR function returns a LONG VARCHAR representation of a character string data type.

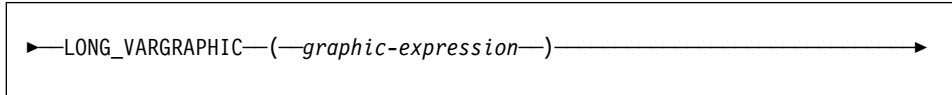
*character-string-expression*

An *expression* that returns a value that is a character string with a length no greater than 32700 bytes.

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

## LONG\_VARGRAPHIC

## LONG\_VARGRAPHIC



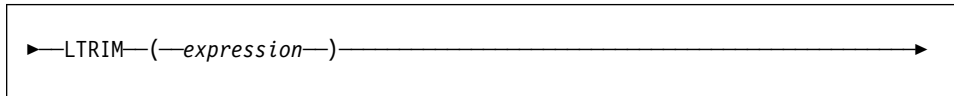
The schema is SYSIBM.

The LONG\_VARGRAPHIC function returns a LONG VARGRAPHIC representation of a double-byte character string.

*graphic-expression*

An *expression* that returns a value that is a graphic string with a length no greater than 16350 double byte characters .

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

**LTRIM**

```
▶LTRIM(expression)
```

The schema is SYSFUN.

Returns the characters of the argument with leading blanks removed.

The argument can be any built-in character string with a maximum length of 1048576 bytes (1M).

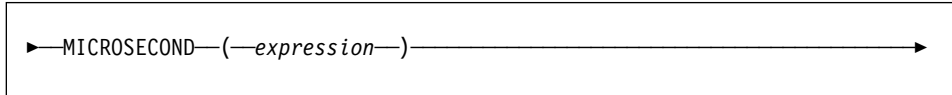
The result of the function is:

- VARCHAR(4000) if the argument is VARCHAR or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR.

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

## MICROSECOND

## MICROSECOND



The schema is SYSIBM.

The MICROSECOND function returns the microsecond part of a value.

The argument must be a timestamp, timestamp duration or a valid character string representation of a timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The other rules depend on the data type of the argument:

- If the argument is a timestamp or a valid string representation of a timestamp:
  - The integer ranges from 0 through 999999.
- If the argument is a duration:
  - The result reflects the microsecond part of the value which is an integer between -999999 through 999999. A nonzero result has the same sign as the argument.

Example:

- Assume a table TABLEA contains two columns, TS1 and TS2, of type TIMESTAMP. Select all rows in which the microseconds portion of TS1 is not zero and the seconds portion of TS1 and TS2 are identical.

```
SELECT * FROM TABLEA
WHERE MICROSECOND(TS1) <> 0 AND
SECOND(TS1) = SECOND(TS2)
```

## MIDNIGHT\_SECONDS

The schema is SYSFUN.

Returns an integer value in the range 0 to 86400 representing the number of seconds between midnight and the time value specified in the argument.

The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example:

- Find the number of seconds between midnight and 00:10:10, and midnight and 13:10:10.

```
VALUES (MIDNIGHT_SECONDS('00:10:10'), MIDNIGHT_SECONDS('13:10:10'))
```

This example returns the following:

```
1          2
-----
        610        47410
```

Since a minute is 60 seconds, there are 610 seconds between midnight and the specified time. The same follows for the second example. There are 3600 seconds in an hour, and 60 seconds in a minute, resulting in 47410 seconds between the specified time and midnight.

- Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
VALUES (MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00'))
```

This example returns the following:

```
1          2
-----
      86400          0
```

Note that these two values represent the same point in time, but return different MIDNIGHT\_SECONDS values.

## MINUTE

## MINUTE

► MINUTE(*expression*)

The schema is SYSIBM.

The MINUTE function returns the minute part of a value.

The argument must be a time, timestamp, time duration, timestamp duration or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The other rules depend on the data type of the argument:

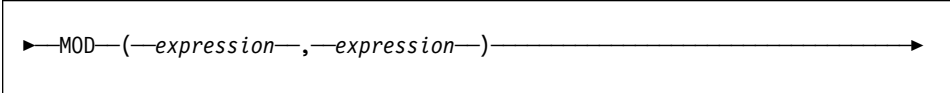
- If the argument is a time, timestamp or valid string representation of a time or timestamp:
  - The result is the minute part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:
  - The result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example:

- Using the CL\_SCHED sample table, select all classes with a duration less than 50 minutes.

```
SELECT * FROM CL_SCHED
WHERE HOUR(ENDING - STARTING) = 0 AND
MINUTE(ENDING - STARTING) < 50
```

## MOD



► MOD(*expression*, *expression*)

The schema is SYSFUN.

Returns the remainder of the first argument divided by the second argument. The result is negative only if first argument is negative.

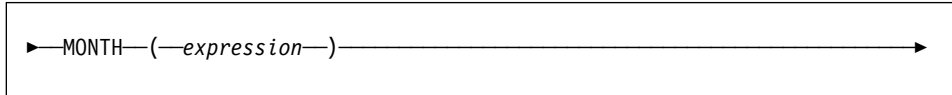
The result of the function is:

- SMALLINT if both arguments are SMALLINT
- INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT.

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

## MONTH

## MONTH



The schema is SYSIBM.

The MONTH function returns the month part of a value.

The argument must be a date, timestamp, date duration, timestamp duration or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The other rules depend on the data type of the argument:

- If the argument is a date, timestamp, or a valid string representation of a date or timestamp:
  - The result is the month part of the value, which is an integer between 1 and 12.
- If the argument is a date duration or timestamp duration:
  - The result is the month part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example:

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

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



**MONTHNAME**

```
▶ MONTHNAME(expression)
```

The schema is SYSFUN.

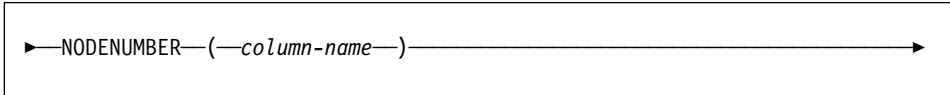
Returns a mixed case character string containing the name of month (e.g. January) for the month portion of the argument, based on the locale when the database was started.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

## NODENUMBER

### NODENUMBER



The schema is SYSIBM.

The NODENUMBER function returns the partition number of the row. For example, if used in a SELECT clause, it returns the partition number for each row of the table that was used to form the result of the SELECT statement.

The partition number returned on transition variables and tables is derived from the current transition values of the partitioning key columns. For example, in a before insert trigger, the function will return the projected partition number given the current values of the new transition variables. However, the values of the partitioning key columns may be modified by a subsequent before insert trigger. Thus, the final partition number of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column of a table. The column can have any data type.<sup>35</sup> If *column-name* references a column of a view the expression in the view for the column must reference a column of the underlying base table and the view must be deletable. A nested or common table expression follows the same rules as a view. See “Notes” on page 589 for the definition of a deletable view.

The specific row (and table) for which the partition number is returned by the NODENUMBER function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER and is never null. Since row-level information is returned, the results are the same, regardless of which column is specified for the table. If there is no db2nodes.cfg file, the result is 0.

The NODENUMBER function cannot be used on replicated tables or within check constraints (SQLSTATE 42881).

Examples:

- Count the number of rows where the row for an EMPLOYEE is on a different partition from the employee's department description in DEPARTMENT.

```
SELECT COUNT(*) FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DEPTNO=E.WORKDEPT
AND NODENUMBER(E.LASTNAME) <>NODENUMBER(D.DEPTNO)
```

<sup>35</sup> This function may not be used as a source function when creating a user-defined function. Since it accepts any data types as an argument, it is not necessary to create additional signatures to support user-defined distinct types.

## NODENUMBER

- Join the EMPLOYEE and DEPARTMENT tables where the rows of the two tables are on the same partition.

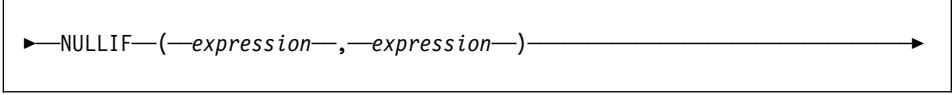
```
SELECT * FROM DEPARTMENT D, EMPLOYEE E
        WHERE NODENUMBER(E.LASTNAME) = NODENUMBER(D.DEPTNO)
```

- Log the employee number and the projected partition number of the new row into a table called EMPINSERTLOG1 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG1
  BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AS NEWTABLE
  FOR EACH MODE ROW MODE DB2SQL
  INSERT INTO EMPINSERTLOG1
    VALUES(NEWTABLE.EMPNO, NODENUMBER(NEWTABLE.EMPNO))
```

## NULLIF

## NULLIF



```
→ NULLIF ( expression , expression ) →
```

The schema is SYSIBM.

The NULLIF function returns a null value if the arguments are equal, otherwise it returns the value of the first argument.

The arguments must be comparable (see “Assignments and Comparisons” on page 70). They can be of either a built-in (other than a long string or DATALINK) or distinct data type (other than based on a long string or DATALINK).<sup>36</sup> The attributes of the result are the attributes of the first argument.

The result of using NULLIF(e1,e2) is the same as using the expression

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

Note that when e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first argument.

Example:

- Assume host variables PROFIT, CASH, and LOSSES have DECIMAL data types with the values 4500.00, 500.00, and 5000.00 respectively:

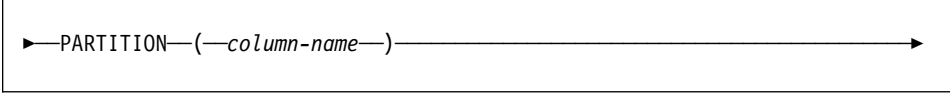
```
NULLIF (:PROFIT + :CASH , :LOSSES )
```

Returns a null value.

---

<sup>36</sup> This function may not be used as a source function when creating a user-defined function. Since it accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

## PARTITION



The schema is SYSIBM.

The PARTITION function returns the partitioning map index of the row obtained by applying the partitioning function on the partitioning key value of the row. For example, if used in a SELECT clause, it returns the partitioning map index for each row of the table that was used to form the result of the SELECT statement.

The partitioning map index returned on transition variables and tables is derived from the current transition values of the partitioning key columns. For example, in a before insert trigger, the function will return the projected partitioning map index given the current values of the new transition variables. However, the values of the partitioning key columns may be modified by a subsequent before insert trigger. Thus, the final partitioning map index of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column of a table. The column can have any data type.<sup>37</sup> If *column-name* references a column of a view the expression in the view for the column must reference a column of the underlying base table and the view must be deletable. A nested or common table expression follows the same rules as a view. See “Notes” on page 589 for the definition of a deletable view.

The specific row (and table) for which the partitioning map index is returned by the PARTITION function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER in the range 0 to 4095. For a table with no partitioning key, the result is always 0. A null value is never returned. Since row-level information is returned, the results are the same, regardless of which column is specified for the table.

The PARTITION function cannot be used on replicated tables or within check constraints(SQLSTATE 42881).

Example:

- List the employee numbers (EMPNO) from the EMPLOYEE table for all rows with a partitioning map index of 100.

<sup>37</sup> This function may not be used as a source function when creating a user-defined function. Since it accepts any data type as an arguments, it is not necessary to create additional signatures to support user-defined distinct types.

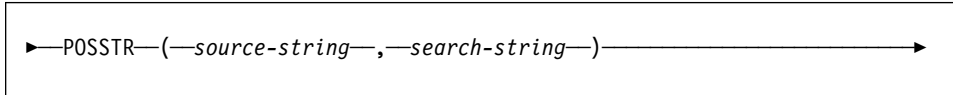
## PARTITION

```
SELECT EMPNO FROM EMPLOYEE
WHERE PARTITION(PHONENO) = 100
```

- Log the employee number and the projected partitioning map index of the new row into a table called EMPINSERTLOG2 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG2
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AS NEWTABLE
FOR EACH MODE ROW MODE DB2SQL
INSERT INTO EMPINSERTLOG2
VALUES(NEWTABLE.EMPNO, PARTITION(NEWTABLE.EMPNO))
```

POSSTR



The schema is SYSIBM.

The POSSTR function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). Numbers for the *search-string* position start at 1 (not 0).

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

*source-string*

An expression that specifies the source string in which the search is to take place.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable (including a locator variable or a file reference variable)
- a scalar function
- a large object locator
- a column name
- an expression concatenating any of the above

*search-string*

An expression that specifies the string that is to be searched for.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable
- a scalar function whose operands are any of the above
- an expression concatenating any of the above

with the restrictions that:

- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARCHARIC or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- The actual length of *search-string* cannot be more than 4000 bytes.

Note that these rules are the same as those for the *pattern-expression* described in “LIKE Predicate” on page 146.

Both *search-string* and *source-string* have zero or more contiguous positions. If the strings are character or binary strings, a position is a byte. If the strings are graphic strings, a position is a graphic (DBCS) character.

## POSSTR

The POSSTR function accepts mixed data strings. However, POSSTR operates on a strict byte-count basis, oblivious to changes between single and multi-byte characters.

The following rules apply:

- The data types of *source-string* and *search-string* must be compatible, otherwise an error is raised (SQLSTATE 42884).
  - If *source-string* is a character string, then *search-string* must be a character string, but not a CLOB or LONG VARCHAR, with an actual length of 4000 bytes or less.
  - If *source-string* is a graphic string, then *search-string* must be a graphic string, but not a DBCLOB or LONG VARGRAPHIC, with an actual length of 2000 double-byte characters or less.
  - If *source-string* is a binary string, then *search-string* must be a binary string with an actual length of 4000 bytes or less.
- If *search-string* has a length of zero, the result returned by the function is 1.
- Otherwise:
  - If *source-string* has a length of zero, the result returned by the function is zero.
  - Otherwise:
    - If the value of *search-string* is equal to an identical length substring of contiguous positions from the value of *source-string*, then the result returned by the function is the starting position of the first such substring within the *source-string* value.
    - Otherwise, the result returned by the function is 0.

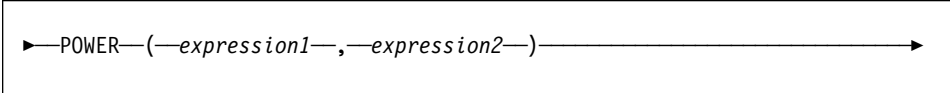
### Example

- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD BEER' within the NOTE\_TEXT column for all entries in the IN\_TRAY table that contain these words.

```
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 SYSFUN.

Returns the value of *expression1* to the power of *expression2*.

The arguments can be of any built-in numeric data type. DECIMAL and REAL arguments are converted to double-precision floating-point number.

The result of the function is:

- INTEGER if both arguments are INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT
- DOUBLE otherwise.

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

## QUARTER

## QUARTER

► `QUARTER` (`expression`) ►

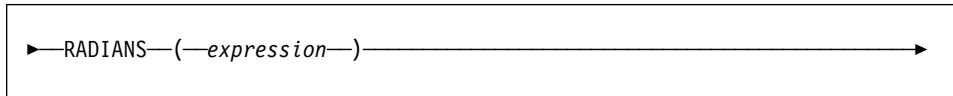
The schema is SYSFUN.

Returns an integer value in the range 1 to 4 representing the quarter of the year for the date specified in the argument.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## RADIANS

A diagram showing the function signature `RADIANS(expression)` with a right-pointing arrow. The text is enclosed in a rectangular box.

The schema is SYSFUN.

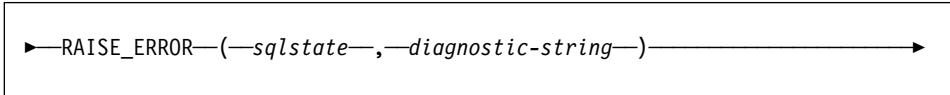
Returns the number of radians converted from argument which is expressed in degrees.

The argument can be of any built-in numeric data types. It is converted to 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.

## RAISE\_ERROR

## RAISE\_ERROR



The schema is SYSIBM.

The RAISE\_ERROR function causes the statement that includes the function to return an error with the specified SQLSTATE, SQLCODE -438 and *diagnostic-string*. The RAISE\_ERROR function always returns NULL with an undefined data type.

### *sqlstate*

A character string containing exactly 5 characters. It must be of type CHAR defined with a length of 5 or type VARCHAR defined with a length of 5 or greater. The *sqlstate* value must follow the rules for application-defined SQLSTATEs as follows:

- 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' since these are not error classes.
- If the SQLSTATE class (first two characters) starts with the character '0' through '6' or 'A' through 'H', then 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', then the subclass (last three characters) can be any of '0' through '9' or 'A' through 'Z'.

If the SQLSTATE does not conform to these rules an error occurs (SQLSTATE 428B3).

### *diagnostic-string*

An expression of type CHAR or VARCHAR that returns a character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.

In order to use this function in a context where Rules for Result Data Types do not apply (such as alone in a select list), a cast specification must be used to give the null returned value a data type. A CASE expression is where the RAISE\_ERROR function will be most useful.

Example:

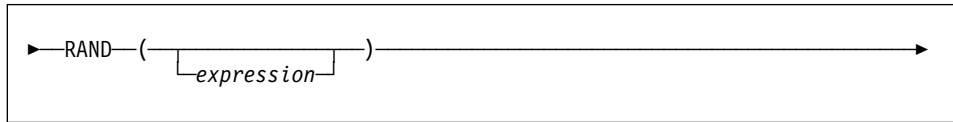
List employee numbers and education levels as Post Graduate, Graduate and Diploma. If an education level is greater than 20, raise an error.

## RAISE\_ERROR

```
SELECT EMPNO,  
       CASE WHEN EDUCLVL < 16 THEN 'Diploma'  
            WHEN EDUCLVL < 18 THEN 'Graduate'  
            WHEN EDUCLVL < 21 THEN 'Post Graduate'  
            ELSE RAISE_ERROR('70001',  
                             'EDUCLVL has a value greater than 20')  
       END  
FROM EMPLOYEE
```

## RAND

## RAND



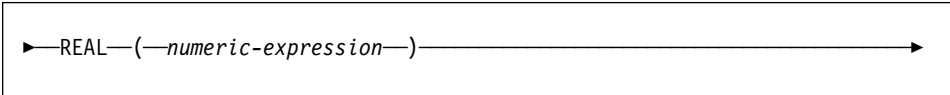
The schema is SYSFUN.

Returns a random floating point value between 0 and 1 using the argument as the optional seed value.

An argument is not required, but if it is specified it can be either INTEGER or SMALLINT.

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.

## REAL



►—REAL—(—*numeric-expression*—)————→

The schema is SYSIBM.

The REAL function returns a single-precision floating-point representation of a number.

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. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable.

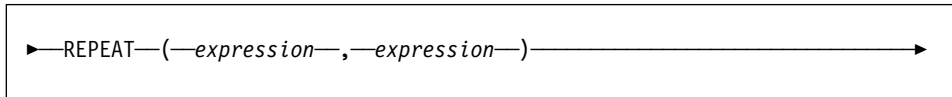
Example:

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. The result is desired in single-precision floating point. Therefore, REAL is applied to SALARY so that the division is carried out in floating point (actually double precision) and then REAL is applied to the complete expression to return the result in single-precision floating point.

```
SELECT EMPNO, REAL(REAL(SALARY)/COMM)
FROM EMPLOYEE
WHERE COMM > 0
```

## REPEAT

## REPEAT



The schema is SYSFUN.

Returns a character string composed of the first argument repeated the number of times specified by the second argument.

The first argument is a character string or binary string with a maximum length of 1048576 bytes. The second argument can be SMALLINT or INTEGER.

The result of the function is:

- VARCHAR(4000) if the first argument is VARCHAR or CHAR
- CLOB(1M) if the first argument is CLOB or LONG VARCHAR
- BLOB(1M) if the first argument is BLOB.

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

Example:

- List the phrase 'REPEAT THIS' five times.

```
VALUES CHAR(REPEAT('REPEAT THIS', 5), 60)
```

This example return the following:

1

-----  
REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS

As mentioned, the output of the REPEAT function is VARCHAR(4000). For the above example the function CHAR has been used to limit the output of REPEAT to 60 bytes.



## REPLACE

```
►—REPLACE—(—expression1—,—expression2—,—expression3—)—————►
```

The schema is SYSFUN.

Replaces all occurrences of *expression2* in *expression1* with *expression3*.

The first argument is a character string or binary string with a maximum length of 1048576 bytes. CHAR is converted to VARCHAR and LONG VARCHAR is converted to CLOB(1M). The second and third arguments are identical to the first argument.

The result of the function is:

- VARCHAR(4000) if the first, second and third arguments are VARCHAR or CHAR
- CLOB(1M) if the first, second and third arguments are CLOB or LONG VARCHAR
- BLOB(1M) if the first, second and third arguments are BLOB.

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

Example:

- Replace all occurrence of the letter 'N' in the word 'DINING' with 'VID'.

```
VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 10)
```

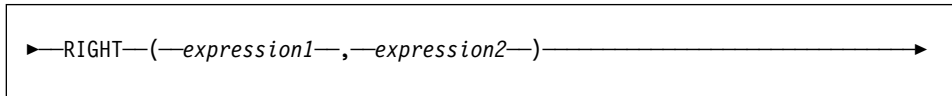
This example returns the following:

```
1
-----
DIVIDIVIDG
```

As mentioned, the output of the REPLACE function is VARCHAR(4000). For the above example the function CHAR has been used to limit the output of REPLACE to 10 bytes.

## RIGHT

## RIGHT



▶—RIGHT—(—*expression1*—,—*expression2*—)————▶

The schema is SYSFUN.

Returns a string consisting of the rightmost *expression2* bytes in *expression1*.

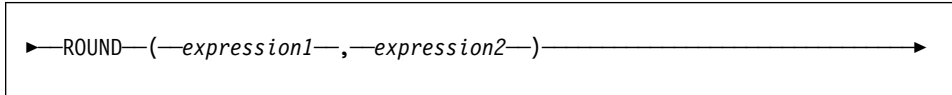
The first argument is a character string or binary string with maximum length of 1048576 bytes. The second argument can be INTEGER or SMALLINT.

The result of the function is:

- VARCHAR(4000) if the first argument is VARCHAR or CHAR
- CLOB(1M) if the first argument is CLOB or LONG VARCHAR
- BLOB(1M) if the first argument is BLOB.

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

ROUND



The schema is SYSFUN.

Returns the expression1 rounded to expression2 places right of the decimal point. If expression2 is negative, expression1 is rounded to the absolute value of expression2 places to the left of the decimal point.

The first argument can be of any built-in numeric data type. The second argument can be INTEGER or SMALLINT. DECIMAL and REAL are converted to double-precision floating-point number for processing by the function.

The result of the function is:

- INTEGER if the first argument is INTEGER or SMALLINT
- BIGINT if the first argument is BIGINT
- DOUBLE if the first argument is DOUBLE, DECIMAL or REAL.

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

Example:

- Display the number 973.726 rounded to 2, 1, 0, -1 and -2 decimal places respectively.

```
VALUES (DECIMAL(ROUND(873.726,2),6,3), DECIMAL(ROUND(873.726,1),6,3),
        DECIMAL(ROUND(873.726,0),6,3), DECIMAL(ROUND(873.726,-1),6,3),
        DECIMAL(ROUND(873.726,-2),6,3))
```

The above example returns:

1	2	3	4	5
-----	-----	-----	-----	-----
873.730	873.700	874.000	870.000	900.000

As mentioned, the output of the ROUND function is DOUBLE. For the above example the function DECIMAL has been used to limit the output of ROUND.

## RTRIM

## RTRIM

► RTRIM(*expression*)

The schema is SYSFUN.

Returns the characters of the argument with trailing blanks removed.

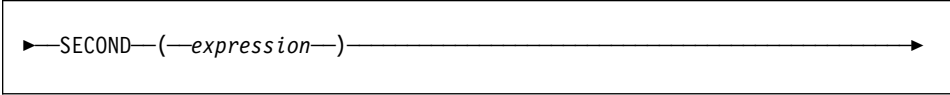
The argument can be of any built-in character string data types.

The result of the function is:

- VARCHAR(4000) if the argument is VARCHAR or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR.

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

## SECOND



► `SECOND` (`expression`) ►

The schema is SYSIBM.

The SECOND function returns the seconds part of a value.

The argument must be a time, timestamp, time duration, timestamp duration or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The other rules depend on the data type of the argument:

- If the argument is a time, timestamp or valid string representation of a time or timestamp:
  - The result is the seconds part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:
  - The result is the seconds part of the value, which is an integer between –99 and 99. A nonzero result has the same sign as the argument.

Examples:

- Assume that the host variable TIME\_DUR (decimal(6,0)) has the value 153045.

`SECOND(:TIME_DUR)`

Returns the value 45.

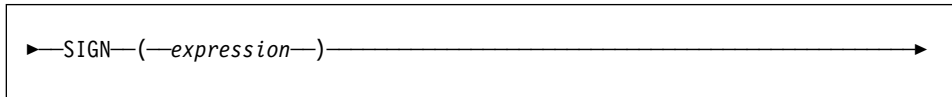
- Assume that the column RECEIVED (timestamp) has an internal value equivalent to 1988-12-25-17.12.30.000000.

`SECOND(RECEIVED)`

Returns the value 30.

## SIGN

## SIGN



The schema is SYSFUN.

Returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.

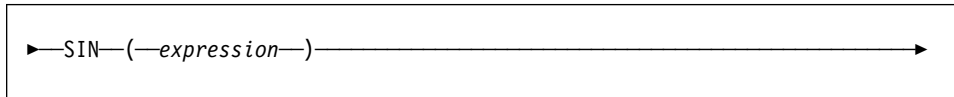
The argument can be of any built-in numeric data types. DECIMAL and REAL are converted to double-precision floating-point number for processing by the function.

The result of the function is:

- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DOUBLE otherwise.

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

## SIN



The schema is SYSFUN.

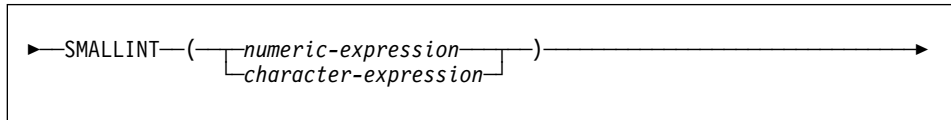
Returns the sine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data types. It is converted to 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.

## SMALLINT

## SMALLINT



The schema is SYSIBM.

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

### *numeric-expression*

An expression that returns a value of any built-in numeric data type.

If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. The decimal part of the argument is truncated if present.

### *character-expression*

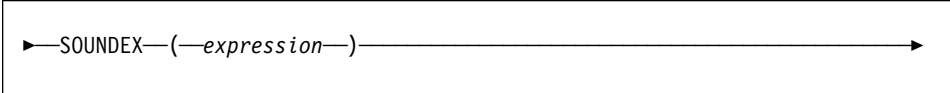
An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). However, the value of the constant must be in the range of small integers (SQLSTATE 22003). The character string cannot be a long string.

If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a small integer column or variable.

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



## SOUNDEX



The schema is SYSFUN.

Returns a 4 character code representing the sound of the words in the argument. The result can be used to compare with the sound of other strings.

The argument can be a character string that is either a CHAR or VARCHAR.

The result of the function is CHAR(4). The result can be null; if the argument is null, the result is the null value.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds. The comparison can be done directly or by passing the strings as arguments to the DIFFERENCE function (see “DIFFERENCE” on page 218).

Example:

Using the EMPLOYEE table, find the EMPNO and LASTNAME of the employee with a surname that sounds like 'Loucesy'.

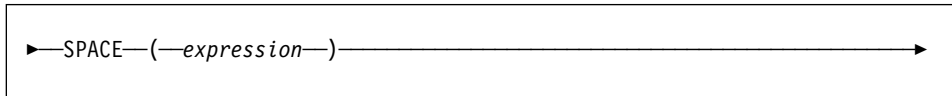
```
SELECT EMPNO, LASTNAME FROM EMPLOYEE
WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy')
```

This example returns the following:

```
EMPNO  LASTNAME
-----  -
000110  LUCCHESI
```

## SPACE

## SPACE



▶ `SPACE`—(*expression*)—▶

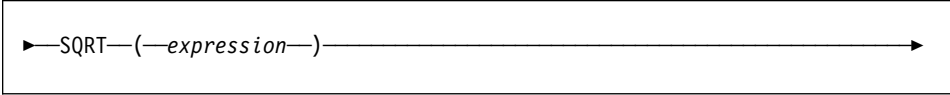
The schema is SYSFUN.

Returns a character string consisting of blanks with length specified by the second argument.

The argument can be SMALLINT or INTEGER.

The result of the function is VARCHAR(4000). The result can be null; if the argument is null, the result is the null value.

## SQRT



► SQRT ( *expression* ) ►

The schema is SYSFUN.

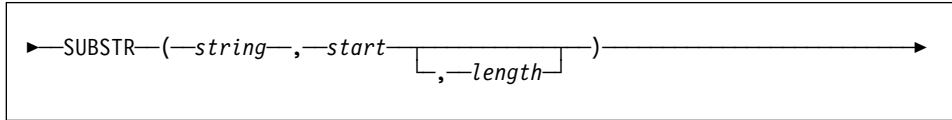
Returns the square root of the argument.

The argument can be any built-in numeric data type. It has to be converted to double-precision floating-point number for processing by the function.

The result of the function is double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

## SUBSTR

## SUBSTR



The schema is SYSIBM.

The SUBSTR function returns a substring of a string.

If *string* is a character string, the result of the function is a character string represented in the code page of its first argument. If it is a binary string, the result of the function is a binary string. If it is a graphic string, the result of the function is a graphic string represented in the code page of its first argument. If any argument of the SUBSTR function can be null, the result can be null; if any argument is null, the result is the null value.

### *string*

An expression that specifies the string from which the result is derived.

If *string* is either a character string or a binary string, a substring of *string* is zero or more contiguous bytes of *string*. If *string* is a graphic string, a substring of *string* is zero or more contiguous double-byte characters of *string*.

### *start*

An expression that specifies the position of the first byte of the result for a character string or a binary string or the position of the first character of the result for a graphic string. *start* must be an integer between 1 and the length or maximum length of *string*, depending on whether *string* is fixed-length or varying-length (SQLSTATE 22011, if out of range).

### *length*

An expression that specifies the length of the result. If specified, *length* must be a binary integer in the range 0 to *n*, where *n* equals (the length attribute of *string*) – *start* + 1 (SQLSTATE 22011, if out of range).

If *length* is explicitly specified, *string* is effectively padded on the right with the necessary number of blank characters (single-byte for character strings; double-byte for graphic strings) so that the specified substring of *string* always exists. The default for *length* is the number of bytes from the byte specified by the *start* to the last byte of *string* in the case of character string or binary string or the number of double-byte characters from the character specified by the *start* to the last character of *string* in the case of a graphic string. However, if *string* is a varying-length string with a length less than *start*, the default is zero and the result is the empty string. (For example, the column NAME with a data type of VARCHAR(18) and a value of 'MCKNIGHT' will yield an empty string with SUBSTR(NAME,10)).

Table 14 on page 287 shows that the result type and length of the SUBSTR function depend on the type and attributes of its inputs.

## SUBSTR

Table 14. Data Type and Length of SUBSTR Result

String Argument Data Type	Length Argument	Result Data Type
CHAR(A)	constant ( $\ell < 255$ )	CHAR( $\ell$ )
CHAR(A)	not specified but <i>start</i> argument is a constant	CHAR(A- <i>start</i> +1)
CHAR(A)	not a constant	VARCHAR(A)
VARCHAR(A)	constant ( $\ell < 255$ )	CHAR( $\ell$ )
VARCHAR(A)	constant ( $254 < \ell < 4001$ )	VARCHAR( $\ell$ )
VARCHAR(A)	not a constant or not specified	VARCHAR(A)
LONG VARCHAR	constant ( $\ell < 255$ )	CHAR( $\ell$ )
LONG VARCHAR	constant ( $254 < \ell < 4001$ )	VARCHAR( $\ell$ )
LONG VARCHAR	constant ( $\ell > 4000$ )	LONG VARCHAR
LONG VARCHAR	not a constant or not specified	LONG VARCHAR
CLOB(A)	constant ( $\ell$ )	CLOB( $\ell$ )
CLOB(A)	not a constant or not specified	CLOB(A)
GRAPHIC(A)	constant ( $\ell < 128$ )	GRAPHIC( $\ell$ )
GRAPHIC(A)	not specified but <i>start</i> argument is a constant	GRAPHIC(A- <i>start</i> +1)
GRAPHIC(A)	not a constant	VARGRAPHIC(A)
VARGRAPHIC(A)	constant ( $\ell < 128$ )	GRAPHIC( $\ell$ )
VARGRAPHIC(A)	constant ( $127 < \ell < 2001$ )	VARGRAPHIC( $\ell$ )
VARGRAPHIC(A)	not a constant	VARGRAPHIC(A)
LONG VARGRAPHIC	constant ( $\ell < 128$ )	GRAPHIC( $\ell$ )
LONG VARGRAPHIC	constant ( $127 < \ell < 2001$ )	VARGRAPHIC( $\ell$ )
LONG VARGRAPHIC	constant ( $\ell > 2000$ )	LONG VARGRAPHIC
LONG VARGRAPHIC	not a constant or not specified	LONG VARGRAPHIC
DBCLOB(A)	constant ( $\ell$ )	DBCLOB( $\ell$ )
DBCLOB(A)	not a constant or not specified	DBCLOB(A)
BLOB(A)	constant ( $\ell$ )	BLOB( $\ell$ )
BLOB(A)	not a constant or not specified	BLOB(A)

If *string* is a fixed-length string, omission of *length* is an implicit specification of  $\text{LENGTH}(\text{string}) - \text{start} + 1$ . If *string* is a varying-length string, omission of *length* is an implicit specification of zero or  $\text{LENGTH}(\text{string}) - \text{start} + 1$ , whichever is greater.

## SUBSTR

Examples:

- Assume the host variable NAME (varchar(50)) has a value of 'BLUE JAY' and the host variable SURNAME\_POS (int) has a value of 6.

```
SUBSTR(:NAME, :SURNAME_POS):ehp2s
```

Returns the value 'JAY'

```
SUBSTR(:NAME, :SURNAME_POS,1)
```

Returns the value 'J'.

- Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION '.

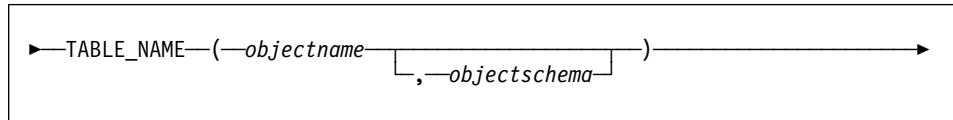
```
SELECT * FROM PROJECT  
WHERE SUBSTR(PROJNAME,1,10) = 'OPERATION '
```

The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

### Notes:

1. In dynamic SQL, *string*, *start*, and *length* may be represented by a parameter marker (?). If a parameter marker is used for *string*, the data type of the operand will be VARCHAR, and the operand will be nullable.
2. Though not explicitly stated in the result definitions above, it follows from these semantics that if *string* is a mixed single- and multi-byte character string, the result may contain fragments of multi-byte characters, depending upon the values of *start* and *length*. That is, the result could possibly begin with the second byte of a double-byte character, and/or end with the first byte of a double-byte character. The SUBSTR function does not detect such fragments, nor provides any special processing should they occur.

## TABLE\_NAME



The schema is SYSIBM.

The TABLE\_NAME function returns an unqualified name of the object found after any alias chains have been resolved. The specified *objectname* (and *objectschema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object.

*objectname*

A character expression representing the unqualified name (usually of an existing alias) to be resolved. *objectname* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 19 characters.

*objectschema*

A character expression representing the schema used to qualify the supplied *objectname* value before resolution. *objectschema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 9 characters.

If *objectschema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(18). If *objectname* can be null, the result can be null; if *objectname* is null, the result is the null value. If *objectschema* is the null value, the default schema name is used. The result is the character string representing an unqualified name. The result name could represent one of the following:

**table** The value for *objectname* was either a table name (the input value is returned) or an alias name that resolved to the table whose name is returned.

**view** The value for *objectname* was either a view name (the input value is returned) or an alias name that resolved to the view whose name is returned.

**undefined object**

The value for *objectname* was either an undefined object (the input value is returned) or an alias name that resolved to the undefined object whose name is returned.

Therefore, if a non-null value is given to this function, a value is always returned, even if no object with the result name exists.

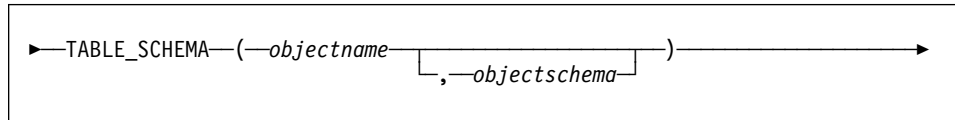
Examples:

## **TABLE\_NAME**

See Examples section in “TABLE\_SCHEMA” on page 291.



## TABLE\_SCHEMA



The schema is SYSIBM.

The TABLE\_SCHEMA function returns the schema name of the object found after any alias chains have been resolved. The specified *objectname* (and *objectschema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object.

*objectname*

A character expression representing the unqualified name (usually of an existing alias) to be resolved. *objectname* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 19 characters.

*objectschema*

A character expression representing the schema used to qualify the supplied *objectname* value before resolution. *objectschema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 9 characters.

If *objectschema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is CHAR(8). If *objectname* can be null, the result can be null; if *objectname* is null, the result is the null value. If *objectschema* is the null value, the default schema name is used. The result is the character string representing a schema name. The result schema could represent the schema name for one of the following:

- table** The value for *objectname* was either a table name (the input or default value of *objectschema* is returned) or an alias name that resolved to a table for which the schema name is returned.
- view** The value for *objectname* was either a view name (the input or default value of *objectschema* is returned) or an alias name that resolved to a view for which the schema name is returned.

**undefined object**

The value for *objectname* was either an undefined object (the input or default value of *objectschema* is returned) or an alias name that resolved to an undefined object for which the schema name is returned.

Therefore, if a non-null *objectname* value is given to this function, a value is always returned, even if the object name with the result schema name does not exist. For example, TABLE\_SCHEMA('DEPT', 'PEOPLE') returns 'PEOPLE ' if the catalog entry is not found.

## TABLE\_SCHEMA

Examples:

- PBIRD tries to select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A1 defined on the table HEDGES.T1.

```
SELECT NPAGES, CARD FROM SYSCAT.TABLES
WHERE TABNAME = TABLE_NAME ('A1')
AND TABSCHEMA = TABLE_SCHEMA ('A1')
```

The requested statistics for HEDGES.T1 are retrieved from the catalog.

- Select the statistics for an object called HEDGES.X1 from SYSCAT.TABLES using HEDGES.X1. Use TABLE\_NAME and TABLE\_SCHEMA since it is not known whether HEDGES.X1 is an alias or a table.

```
SELECT NPAGES, CARD FROM SYSCAT.TABLES
WHERE TABNAME = TABLE_NAME ('X1', 'HEDGES')
AND TABSCHEMA = TABLE_SCHEMA ('X1', 'HEDGES')
```

Assuming that HEDGES.X1 is a table, the requested statistics for HEDGES.X1 are retrieved from the catalog.

- Select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A2 defined on HEDGES.T2 where HEDGES.T2 does not exist.

```
SELECT NPAGES, CARD FROM SYSCAT.TABLES
WHERE TABNAME = TABLE_NAME ('A2', 'PBIRD')
AND TABSCHEMA = TABLE_SCHEMA ('A2', 'PBIRD')
```

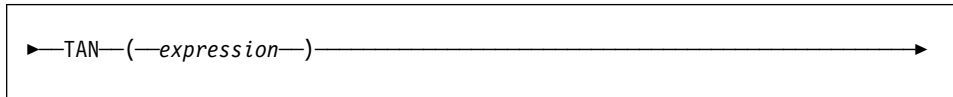
The statement returns 0 records as no matching entry is found in SYSCAT.TABLES where TABNAME = 'T2' and TABSCHEMA = 'HEDGES'.

- Select the qualified name of each entry in SYSCAT.TABLES along with the final referenced name for any alias entry.

```
SELECT TABSCHEMA AS SCHEMA, TABNAME AS NAME,
TABLE_SCHEMA (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_SCHEMA,
TABLE_NAME (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_NAME
FROM SYSCAT.TABLES
```

The statement returns the qualified name for each object in the catalog and the final referenced name (after alias has been resolved) for any alias entries. For all non-alias entries, BASE\_TABNAME and BASE\_TABSCHEMA are null so the REAL\_SCHEMA and REAL\_NAME columns will contain nulls.

## TAN



The schema is SYSFUN.

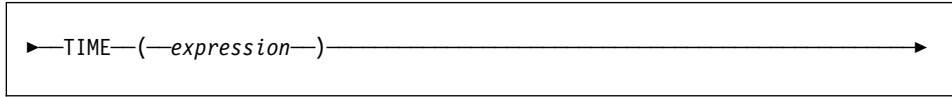
Returns the tangent of the argument, where the argument is an angle expressed in radians.

The argument can be any built-in numeric data type. It has to be converted to double-precision floating-point number for processing by the function.

The result of the function is double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

## TIME

## TIME



The schema is SYSIBM.

The TIME function returns a time from a value.

The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is a 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:

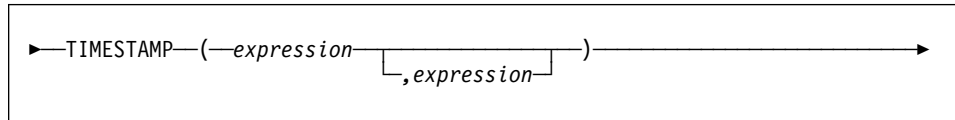
- If the argument is a time:
  - The result is that time.
- If the argument is a timestamp:
  - The result is the time part of the timestamp.
- If the argument is a character string:
  - The result is the time represented by the character string.

Example:

- Select all notes from the IN\_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

```
SELECT * FROM IN_TRAY
WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
```

TIMESTAMP



The schema is SYSIBM.

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

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

- If only one argument is specified:
  - It must be a timestamp, a valid character string representation of a timestamp, or a character string of length 14 that is neither a CLOB nor a LONG VARCHAR.

A character string of length 14 must be a string of digits that represents a valid date and time in the form *yyyxxddhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.
- If both arguments are specified:
  - The first argument must be a date or a valid character string representation of a date and the second argument must be a time or a valid string representation of a time.

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. If the argument is a character string of length 14, the timestamp has a microsecond part of zero.

Example:

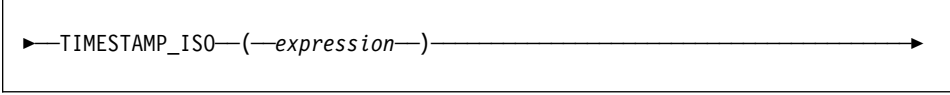
- Assume the column START\_DATE (date) has a value equivalent to 1988-12-25, and the column START\_TIME (time) has a value equivalent to 17.12.30.

**TIMESTAMP**(START\_DATE, START\_TIME)

## TIMESTAMP

Returns the value '1988-12-25-17.12.30.000000'.

## TIMESTAMP\_ISO



►—TIMESTAMP\_ISO—(—*expression*—)————→

The schema is SYSFUN.

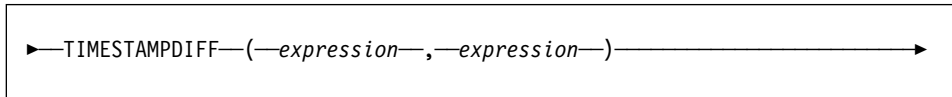
Returns a timestamp value based on date, time or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of CURRENT DATE for the date elements and zero for the fractional time element.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is TIMESTAMP. The result can be null; if the argument is null, the result is the null value.

## TIMESTAMPDIFF

## TIMESTAMPDIFF



The schema is SYSFUN.

Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.

The first argument can be either INTEGER or SMALLINT. Valid values of interval (the first argument) are:

<b>1</b>	Fractions of a second
<b>2</b>	Seconds
<b>4</b>	Minutes
<b>8</b>	Hours
<b>16</b>	Days
<b>32</b>	Weeks
<b>64</b>	Months
<b>128</b>	Quarters
<b>256</b>	Years

The second argument is the result of subtracting two timestamps types and converting the result to CHAR(22).

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

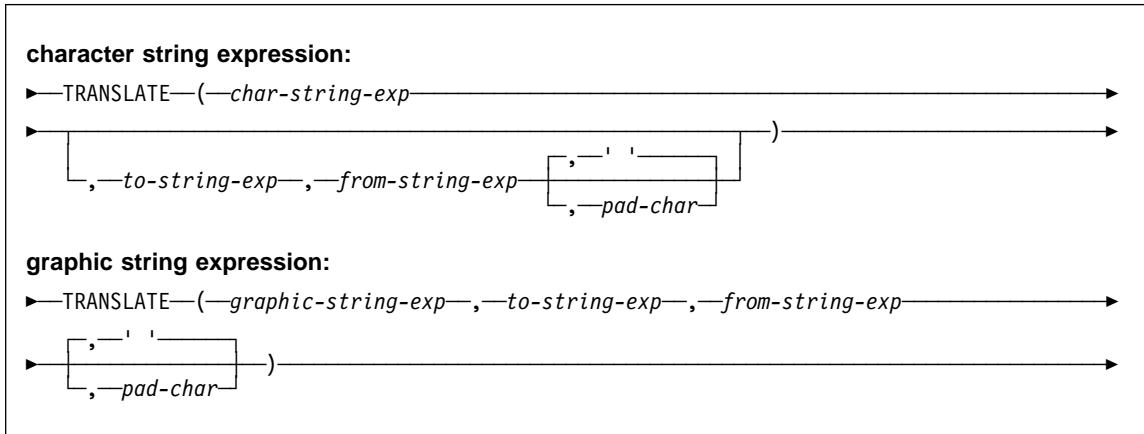
The following assumptions may be used in estimating the difference:

- there are 365 days in a year
- there are 30 days in a month
- there are 24 hours in a day
- there are 60 minutes in an hour
- there are 60 seconds in a minute

These assumptions are used when converting the information in the second argument, which is a timestamp duration, to the interval type specified in the first argument. The returned estimate may vary by a number of days. For example, if the number of days (interval 16) is requested for a difference in timestamps for '1997-03-01-00.00.00' and '1997-02-01-00.00.00', the result is 30. This is because the difference between the timestamps is 1 month so the assumption of 30 days in a month applies.



TRANSLATE



The schema is SYSIBM.

The TRANSLATE function returns a value in which one or more characters in a string expression may have been translated into other characters.

The result of the function has the same data type and code page as the first argument. The length attribute of the result is the same as that of the first argument. If any specified expression can be NULL, the result can be NULL. If any specified expression is NULL, the result will be NULL.

*char-string-exp* or *graphic-string-exp*

A string to be translated.

*to-string-exp*

Is a string of characters to which certain characters in the *char-string-exp* will be translated.

If the *to-string-exp* is not present and the data type is not graphic, all characters in the *char-string-exp* will be monocased (that is, the characters a-z will be translated to the characters A-Z, and characters with diacritical marks will be translated to their upper case equivalents if they exist. For example, in code page 850, é maps to É, but ÿ is not mapped since code page 850 does not include Ÿ).

*from-string-exp*

Is a string of characters which, if found in the *char-string-exp*, will be translated to the corresponding character in the *to-string-exp*. If the *from-string-exp* contains duplicate characters, the first one found will be used, and the duplicates will be ignored. If the *to-string-exp* is longer than the *from-string-exp*, the surplus characters will be ignored. If the *to-string-exp* is present, the *from-string-exp* must also be present.

## TRANSLATE

### *pad-char-exp*

Is a single character that will be used to pad the *to-string-exp* if the *to-string-exp* is shorter than the *from-string-exp*. The *pad-char-exp* must have a length attribute of one, or an error is returned. If not present, it will be taken to be a single-byte blank.

The arguments may be either strings of data type CHAR or VARCHAR, or graphic strings of data type GRAPHIC or VARGRAPHIC. They may not have data type LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, or DBCLOB.

With *graphic-string-exp*, only the *pad-char-exp* is optional (if not provided, it will be taken to be the double-byte blank), and each argument, including the pad character, must be of graphic data type.

The result is the string that occurs after translating all the characters in the *char-string-exp* or *graphic-string-exp* that occur in the *from-string-exp* to the corresponding character in the *to-string-exp* or, if no corresponding character exists, to the pad character specified by the *pad-char-exp*.

The code page of the result of TRANSLATE is always the same as the code page of the first operand, which is never converted. Each of the other operands is converted to the code page of the first operand unless it or the first operand is defined as FOR BIT DATA (in which case there is no conversion).

If the arguments are of data type CHAR or VARCHAR, the corresponding characters of the *to-string-exp* and the *from-string-exp* must have the same number of bytes. For example, it is not valid to translate a single-byte character to a multi-byte character or vice versa. An error will result if an attempt is made to do this. The *pad-char-exp* must not be the first byte of a valid multi-byte character, or SQLSTATE 42815 is returned. If the *pad-char-exp* is not present, it will be taken to be a single-byte blank.

If only the *char-string-exp* is specified, single-byte characters will be moncased and multi-byte characters will remain unchanged.

Examples:

- Assume the host variable SITE (VARCHAR(30)) has a value of 'Hanauma Bay'.

```
TRANSLATE (:SITE)
```

Returns the value 'HANAUMA BAY'.

```
TRANSLATE (:SITE 'j', 'B')
```

Returns the value 'Hanauma jay'.

```
TRANSLATE (:SITE, 'ei', 'aa')
```

Returns the value 'Heneume Bey'.

```
TRANSLATE (:SITE, 'bA', 'Bay', '%')
```

Returns the value 'HAnAumA bA%'.

```
TRANSLATE (:SITE, 'r', 'Bu')
```

Returns the value 'Hana ma ray'.

### TRUNC or TRUNCATE

► `TRUNC or TRUNCATE` (`expression`, `expression`) ►

The schema is SYSFUN.

Returns argument1 truncated to argument2 places right of decimal point. If argument2 is negative, argument1 is truncated to the absolute value of argument2 places to the left of the decimal point.

The first argument can be any built-in numeric data type. The second argument has to be an INTEGER or SMALLINT. DECIMAL and REAL are converted to double-precision floating-point number for processing by the function.

The result of the function is:

- INTEGER if the first argument is INTEGER or SMALLINT
- BIGINT if the first argument is BIGINT
- DOUBLE if the first argument is DOUBLE, DECIMAL or DOUBLE.

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

## TYPE\_ID

### TYPE\_ID

►TYPE\_ID(—*expression*—)◄

The schema is SYSIBM.

The TYPE\_ID function returns the internal type identifier of the dynamic data type of the *expression*.

The argument must be a user-defined structured type.<sup>38</sup>

The data type of the result of the function is INTEGER. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

The value returned by the TYPE\_ID function is not portable across databases. The value may be different, even though the type schema and type name of the dynamic data type are the same. When coding for portability, use the TYPE\_SCHEMA and TYPE\_NAME functions to determine the the type schema and type name.

Examples:

- A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO\_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the internal type identifier of the row that corresponds to the reference.

```
SELECT TASK, WHO_RESPONSIBLE->NAME,  
        TYPE_ID(DEREF(WHO_RESPONSIBLE))  
FROM ACTIVITIES
```

The Deref function is used to return the object corresponding to the row.

<sup>38</sup> This function may not be used as a source function when creating a user-defined function. Since it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

**TYPE\_NAME**

►TYPE\_NAME(—*expression*—)—————►

The schema is SYSIBM.

The TYPE\_NAME function returns the unqualified name of the dynamic data type of the *expression*.

The argument must be a user-defined structured type.<sup>39</sup>

The data type of the result of the function is VARCHAR(18). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE\_SCHEMA function to determine the schema name of the type name returned by TYPE\_NAME.

Examples:

- A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO\_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the type of the row that corresponds to the reference.

```
SELECT TASK, WHO_RESPONSIBLE->NAME,
       TYPE_NAME(DEREF(WHO_RESPONSIBLE)),
       TYPE_SCHEMA(DEREF(WHO_RESPONSIBLE))
FROM ACTIVITIES
```

The DEREf function is used to return the object corresponding to the row.

<sup>39</sup> This function may not be used as a source function when creating a user-defined function. Since it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

## TYPE\_SCHEMA

### TYPE\_SCHEMA

►TYPE\_SCHEMA(—*expression*—)—————►

The schema is SYSIBM.

The TYPE\_SCHEMA function returns the schema name of the dynamic data type of the *expression*.

The argument must be a user-defined structured type.<sup>40</sup>

The data type of the result of the function is CHAR(8). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE\_NAME function to determine the type name associated with the schema name returned by TYPE\_SCHEMA.

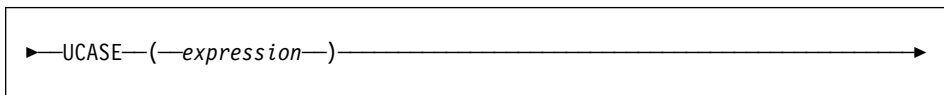
Examples:

See Examples section in “TYPE\_NAME” on page 303.

---

<sup>40</sup> This function may not be used as a source function when creating a user-defined function. Since it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

## UCASE

A diagram showing the function signature UCASE(expression) with a right-pointing arrow. The text is enclosed in a rectangular box with a thin black border. The text inside the box is "▶ UCASE ( *expression* ) ▶", where the arrow points to the right from the opening parenthesis.

The schema is SYSFUN.

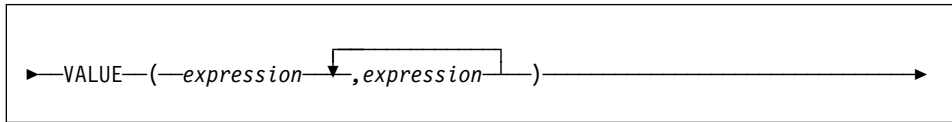
Returns a string in which all the characters have been converted to upper case characters. All characters in the expression are monocased. That is a-z are translated to A-Z, and characters with diacritical marks are translated to their upper case if they exist. For example, in code page 850, é maps to É, but ÿ is not mapped since code page 850 does not include Ÿ.

The argument can be CHAR or VARCHAR.

The result of the function is VARCHAR(4000). The result can be null; if the argument is null, the result is the null value.

## VALUE

## VALUE



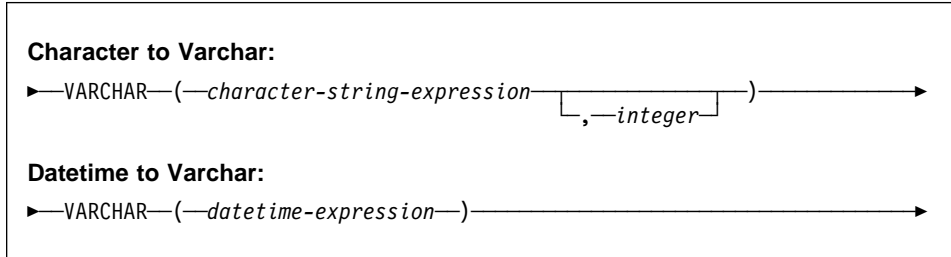
The schema is SYSIBM.

The VALUE function returns the first argument that is not null.

VALUE is a synonym for COALESCE. See “COALESCE” on page 202 for details.



VARCHAR



The schema is SYSIBM.

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

The result of the function is a varying-length string (VARCHAR data type). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

**Character to Varchar**

*character-string-expression*

An expression whose value must be of a character-string data type with a maximum length no greater than 4 000 bytes.

*integer*

The length attribute for the resulting varying-length character string. The value must be between 0 and 4 000. If this argument is not specified, the length of the result is the same as the length of the argument.

**Datetime to Varchar**

*datetime-expression*

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

Example:

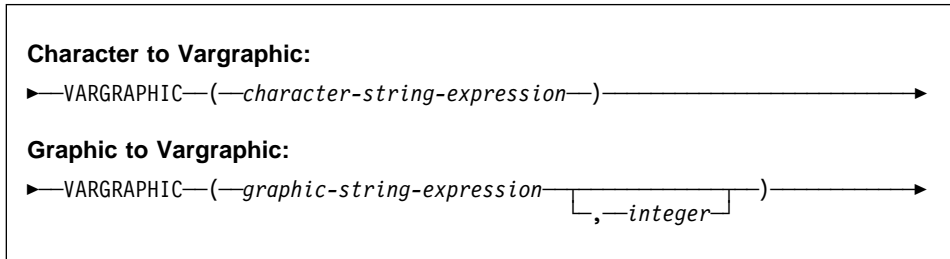
- Using the EMPLOYEE table, set the host variable JOB\_DESC (varchar(8)) to the VARCHAR equivalent of the job description (JOB defined as CHAR(8)) for employee Delores Quintana.

```

SELECT VARCHAR(JOB)
  INTO :JOB_DESC
  FROM EMPLOYEE
  WHERE LASTNAME = 'QUINTANA'
    
```

## VARGRAPHIC

## VARGRAPHIC



The schema is SYSIBM.

The VARGRAPHIC function returns a graphic string representation of a:

- character string value, converting single byte characters to double byte characters
- graphic string value, if the first argument is any type of graphic string.

The result of the function is a varying length graphic string (VARGRAPHIC data type). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

### Character to Vargraphic

#### *character-string-expression*

An expression whose value must be of a character string data type other than LONG VARCHAR or CLOB, and whose maximum length must not be greater than 2000 bytes.

The length attribute of the result is equal to the length attribute of the argument.

Let S denote the value of the *character-string-expression*. Each single-byte character in S is converted to its equivalent double-byte representation or to the double-byte substitution character in the result; each double-byte character in S is mapped 'as-is'. If the first byte of a double-byte character appears as the last byte of S, it is converted into the double-byte substitution character. The sequential order of the characters in S is preserved.

The following are additional considerations for the conversion.

- The conversion to double-byte code points by the VARGRAPHIC function is based on the code page of the operand.
- Double-byte characters of the operand are not converted (see Appendix O, "Japanese and Traditional-Chinese EUC Considerations" on page 973 for exception). All other characters are converted to their corresponding double-byte depiction. If there is no corresponding double-byte depiction, the double-byte substitution character for the code page is used.

## VARGRAPHIC

- No warning or error code is generated if one or more double-byte substitution characters are returned in the result.

### Graphic to Vargraphic

#### *graphic-string-expression*

An expression that returns a value that is a graphic string.

#### *integer*

The length attribute for the resulting varying length graphic string. The value must be between 0 and 2000. If this argument is not specified, the length of the result is the same as the length of the argument.

If the length of the *graphic-string-expression* is greater than the length attribute of the result, truncation is performed and a warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the *graphic-string-expression* was not a long string (LONG VARGRAPHIC or DBCLOB).

## WEEK

## WEEK

► `WEEK` (`expression`) ►

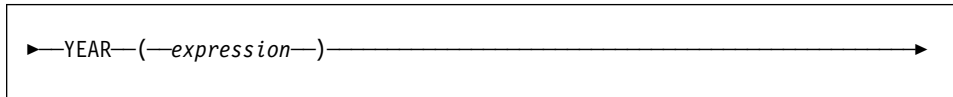
The schema is SYSFUN.

Returns the week of the year of the argument as an integer value in range 1-54 . The week starts with Sunday.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## YEAR



The schema is SYSIBM.

The YEAR function returns the year part of a value.

The argument must be a date, timestamp, date duration, timestamp duration or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp:
  - The result is the 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.

Examples:

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

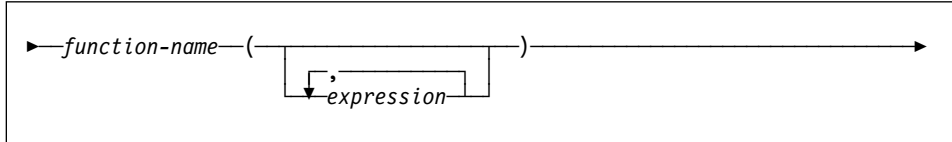
```
SELECT * FROM PROJECT
WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
```
- Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.
 

```
SELECT * FROM PROJECT
WHERE YEAR(PRENDATE - PRSTDATE) < 1
```

## User-Defined Functions

---

### User-Defined Functions



User-defined functions are extensions or additions to the existing built-in functions of the SQL language. A user-defined function can be a scalar function, which returns a single value each time it is called, a column function, which is passed a set of like values and returns a single value for the set, or a table function, which returns a table. (Note that a UDF can be a column function only when it is sourced on an existing column function. Similarly, a UDF can be a table function only if it is an external function.)

A user-defined scalar or column function registered with the database can be referenced in the same contexts that any built-in function can appear.

A user-defined table function registered with the database can be referenced only in the FROM clause of a SELECT, as described in “from-clause” on page 321.

A user-defined function is referenced by means of a qualified or unqualified function name, followed by parentheses enclosing the function arguments (if any).

Arguments of the function must correspond in number and position to the parameters specified in the user-defined function as it was registered with the database. In addition, the arguments must be of data types promotable to the data types of the corresponding defined parameters. (see “CREATE FUNCTION” on page 467).

The result of the function is as specified in the RETURNS clause specified when the user-defined function was registered. The RETURNS clause determines if a function is a table function or not.

If the NOT NULL CALL clause was specified (or defaulted to) when the function was registered then, if any argument is null, the result is null. For table functions, this is interpreted to mean a return table with no rows (empty table).

There are a collection of user-defined functions provided in the SYSFUN schema (see Table 13 on page 156).

Examples:

- Assume that a scalar function called ADDRESS was written to extract the home address from a script format resume. The ADDRESS function expects a CLOB argument and returns a VARCHAR(4000). The following example illustrates the invocation of the ADDRESS function.

```
SELECT EMPNO, ADDRESS(RESUME) FROM EMP_RESUME
WHERE RESUME_FORMAT = 'SCRIPT'
```

## User-Defined Functions

- Assume a table T2 with a numeric column A and the ADDRESS function described in the previous example. The following example illustrates an attempt to invoke the ADDRESS function with an incorrect argument.

```
SELECT ADDRESS(A) FROM T2
```

An error (SQLSTATE 42884) is raised since there is no function with a matching name and with a parameter promotable from the argument.

- Assume a table function WHO was written to return information about the sessions on the server machine which were active at the time the statement is executed. The following example illustrates the invocation of WHO in a FROM clause (TABLE keyword with mandatory correlation variable).

```
SELECT ID, START_DATE, ORIG_MACHINE  
FROM TABLE( WHO() ) AS QQ  
WHERE START_DATE LIKE 'MAY%'
```

The column names of the WHO() table are defined in the CREATE FUNCTION statement.

## User-Defined Functions



---

## Chapter 5. Queries

A *query* specifies a result table.

A query is a component of certain SQL statements. The three forms of a query are:

- subselect
- fullselect
- select-statement.

There is another SQL statement that can be used to retrieve at most a single row described under “SELECT INTO” on page 705.

### Authorization

For each table or view referenced in the query, the authorization ID of the statement must have at least one of the following:

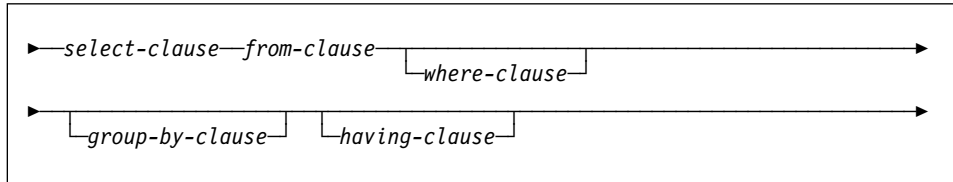
- SYSADM or DBADM authority
- CONTROL privilege
- SELECT privilege.

Group privileges are not checked for queries contained in static SQL statements.

## subselect

---

### subselect



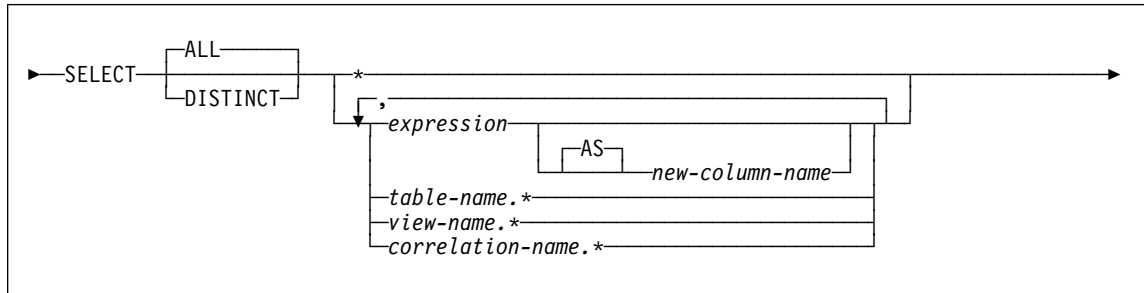
The *subselect* is a component of the fullselect.

A subselect specifies a result table derived from the tables or views identified in the 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 the names or expressions specified in the **SELECT** clause, and R is the result of the previous operation of the subselect. For example, if the only clauses specified are **SELECT**, **FROM**, and **WHERE**, 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. If **DISTINCT** is used, no string column of the result table can have a maximum length that is greater than 254 bytes. **DISTINCT** may be used more than once in a subselect. This includes **SELECT DISTINCT**, the use of **DISTINCT** in a column function of the select list or **HAVING** clause, and subqueries of the subselect.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value of the second. (For determining duplicates, 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 program containing the **SELECT** clause is bound. Hence, \* (the asterisk) does not identify any columns that have been added to a table after the statement containing the table reference has been bound.

*expression*

Specifies the values of a result column. May be any expression of the type described in Chapter 3, but commonly the expressions used include column names. Each column name used in the select list must unambiguously identify a column of R.

## select-clause

*new-column-name* or **AS** *new-column-name*

Names or renames the result column. The name must not be qualified and does not have to be unique. Subsequent usage of column-name is limited as follows:

- A new-column-name specified in the AS clause can be used in the order-by-clause, provided the name is unique.
- A new-column-name specified in the AS clause of the select list cannot be used in any other clause within the subselect (where-clause, group-by-clause or having-clause).
- A new-column-name specified in the AS clause cannot be used in the update-clause.
- A new-column-name specified in the AS clause is known outside the fullselect of nested table expressions, common table expressions and CREATE VIEW.

*name*.\*

Represents a list of names that identify the columns of *name*. The *name* may be a table name, view name, or correlation name, and must designate a table or view named in the FROM clause. The first name in the list identifies the first column of the table or view, the second name in the list identifies the second column of the table or view, and so on.

The list of names is established when the statement containing the SELECT clause is bound. Therefore, \* does not identify any columns that have been added to a table after the statement has been bound.

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 when the statement is prepared) and cannot exceed 500 .

### Limitations on String Columns

For limitations on the select list, see “Restrictions Using Varying-Length Character Strings” on page 57.

### Applying the Select List

Some of the results of applying the select list to R depend on whether or not GROUP BY or HAVING is used. The results are described in two separate lists:

**If GROUP BY or HAVING is used:**

- An expression *X* (not a column function) used in the select list must have a GROUP BY clause with:
  - a *grouping-expression* in which each column-name unambiguously identifies a column of R (see “group-by-clause” on page 329) or
  - each column of R referenced in *X* as a separate *grouping-expression*.

- 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.

**If neither GROUP BY nor HAVING is used:**

- Either the select list must not include any column functions, or each *column-name* in the select list must be specified within a column function or must be a correlated column reference.
- If the select does not include column functions, then the select list 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, then R is the source of the arguments of the functions and the result of applying the select list is one row.

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 do not allow null values if they are derived from:

- A column that does not allow null values
- A constant
- The COUNT or COUNT\_BIG function
- A host variable that does not have an indicator variable
- A scalar function or expression that does not include an operand that allows nulls.

Result columns allow null values if they are derived from:

- Any column function except COUNT or COUNT\_BIG
- A column that allows null values
- A scalar function or expression that includes an operand that allows nulls
- A NULLIF function with arguments containing equal values.
- A host variable that has an indicator variable.
- A result of a set operation if at least one of the corresponding items in the select list is nullable.
- An arithmetic expression or view column that is derived from an arithmetic expression and the database is configured with DFT\_SQLMATHWARN set to yes
- A dereference operation.

**Names of result columns:**

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and the result column is derived from a column, then the result column name is the unqualified name of that column.
- If the AS clause is not specified and the result column is derived using a dereference operation, then the result column name is the unqualified name of the target column of the dereference operation.

## select-clause

- All other result column names are unnamed.<sup>41</sup>

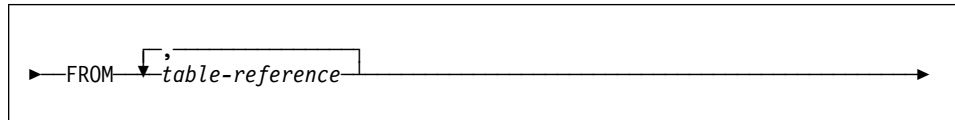
**Data types of result columns:** Each column of the result of SELECT acquires a data type from the expression from which it is derived.

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 constant	DECIMAL, with the precision and scale of the constant
a floating-point constant	DOUBLE
the name of any numeric variable	the same as the data type of the variable, with the same precision and scale for DECIMAL variables.
an expression	For a description of data type attributes, see "Expressions" on page 117.
any function	(see Chapter 4 to determine the data type of the result.)
a hexadecimal constant representing $n$ bytes	VARCHAR( $n$ ). The codepage is the database codepage.
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 variable	the same as the data type of the variable, with the same length attribute. If the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.
a character string constant of length $n$	VARCHAR( $n$ )
a graphic string constant of length $n$	VARGRAPHIC( $n$ )
the name of a datetime column	the same as the data type of the column.

---

<sup>41</sup> The system assigns temporary numbers (as character strings) to these columns. These system assigned names cannot be used in an order-by clause.

## from-clause

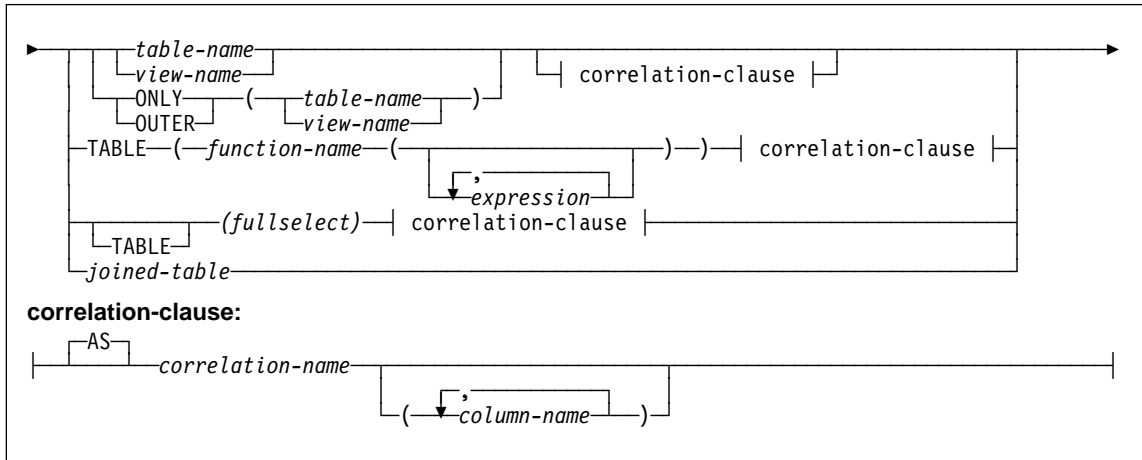


The FROM clause specifies an intermediate result table.

If one table-reference is specified, the intermediate result table is simply the result of that table-reference. If more than one table-reference is specified, the intermediate result table consists of all possible combinations of the rows of the specified table-references (the Cartesian product). Each row of the result is a row from the first table-reference concatenated with a row from the second table-reference, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual table-references. For a description of *table-reference*, see “table-reference” on page 322.

## table-reference

## table-reference



Each *table-name* or *view-name* specified as a table-reference must identify an existing table or view at the application server or the table-name of a common table expression (see “common-table-expression” on page 356) defined preceding the fullselect containing the table-reference. If the *table-name* references a typed table, the name denotes the UNION ALL of the table with all its subtables, with only the columns of the *table-name*. Similarly, if the *view-name* references a typed view, the name denotes the UNION ALL of the view with all its subviews, with only the columns of the *view-name*.

The use of `ONLY(table-name)` or `ONLY(view-name)` means that the rows of the proper subtables or subviews are not included. If the *table-name* used with `ONLY` does not have subtables, then `ONLY(table-name)` is equivalent to specifying *table-name*. If the *view-name* used with `ONLY` does not have subviews, then `ONLY(view-name)` is equivalent to specifying *view-name*.

The use of `OUTER(table-name)` or `OUTER(view-name)` represents a virtual table. If the *table-name* or *view-name* used with `OUTER` does not have subtables or subviews, then specifying `OUTER` is equivalent to not specifying `OUTER`. `OUTER(table-name)` is derived from *table-name* as follows:

- The columns include the columns of *table-name* followed by the additional columns introduced by each of its subtables (if any). The additional columns are added on the right, traversing the subtable hierarchy in depth-first order. Subtables that have a common parent are traversed in creation order of their types.
- The rows include all the rows of *table-name* and all the rows of its subtables. Null values are returned for columns that are not in the subtable for the row.

The previous points also apply to `OUTER(view-name)`, substituting *view-name* for *table-name* and subview for subtable.



Since OUTER(table-name) or OUTER(view-name) includes rows and columns from every subtable of table-name or subview of view-name, SELECT privilege is required on each of these subtables or subviews.

Each *function-name* together with the types of its arguments, specified as a table reference must resolve to an existing table function at the application server.

A fullselect in parentheses followed by a correlation name is called a *nested table expression*.

A *joined-table* specifies an intermediate result set that is the result of one or more join operations. For more information, see “joined-table” on page 326.

The exposed names of all table references should be unique. An exposed name is:

- A *correlation-name*,
- A *table-name* that is not followed by a *correlation-name*,
- A *view-name* that is not followed by a *correlation-name*.

Each *correlation-name* is defined as a designator of the immediately preceding *table-name*, *view-name*, *function-name* reference or nested table expression. Any qualified reference to a column for a table, view, table function or nested table expression must use the exposed name. If the same table name or view name is specified twice, at least one specification should be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table or view. When a *correlation-name* is specified, *column-names* can also be specified to give names to the columns of the *table-name*, *view-name*, *function-name* reference or nested table expression. For more information, see “Correlation Names” on page 99.

In general, table functions and nested table expressions can be specified on any from-clause. Columns from the table functions and nested table expressions can be referenced in the select list and in the rest of the subselect using the correlation name which must be specified. The scope of this correlation name is the same as correlation names for other table or view names in the FROM clause. A nested table expression can be used:

- in place of a view to avoid creating the view (when general use of the view is not required)
- when the desired result table is based on host variables.

### Table Function References

In general, a table function together with its argument values can be referenced in the FROM clause of a SELECT in exactly the same way as a table or view. There are, however, some special considerations which apply.

- Table Function Column Names

Unless alternate column names are provided following the correlation-name, the column names for the table function are those specified in the RETURNS clause of the CREATE FUNCTION statement. This is analogous to the names of the columns of a table, which are of course defined in the CREATE TABLE statement.

## table-reference

See “CREATE FUNCTION (External Table)” on page 484 for details about creating a table function.

- Table Function Resolution

The arguments specified in a table function reference, together with the function name, are used by an algorithm called *function resolution* to determine the exact function to be used. This is no different from what happens with other functions (such as scalar functions), used in a statement. Function resolution is covered in “Function Resolution” on page 112.

- Table Function Arguments

As with scalar function arguments, table function arguments can in general be any valid SQL expression. So the following examples are valid syntax:

```
Example 1: SELECT c1
           FROM TABLE( tf1('Zachary') ) AS z
           WHERE c2 = 'FLORIDA';
```

```
Example 2: SELECT c1
           FROM TABLE( tf2 (:hostvar1, CURRENT DATE) ) AS z;
```

```
Example 3: SELECT c1
           FROM t
           WHERE c2 IN
              (SELECT c3 FROM
               TABLE( tf5(t.c4) ) AS z    -- correlated reference
              )                          -- to previous FROM clause
```

### Correlated References in table-references

Correlated references can be used in nested table expressions or as arguments to table functions. The basic rule that applies for both these cases is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. This hierarchy includes the table-references that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the TABLE keyword must appear before the fullselect. So the following examples are valid syntax:

## table-reference

Example 1: **SELECT** t.c1, z.c5  
**FROM** t, **TABLE**( tf3(t.c2) ) **AS** z -- t precedes tf3 in FROM  
**WHERE** t.c3 = z.c4; -- so t.c2 is known

Example 2: **SELECT** t.c1, z.c5  
**FROM** t, **TABLE**( tf4(2 \* t.c2) ) **AS** z -- t precedes tf3 in FROM  
**WHERE** t.c3 = z.c4; -- so t.c2 is known

Example 3: **SELECT** d.deptno, d.deptname,  
empinfo.avgsal, empinfo.empcount  
**FROM** department d,  
**TABLE** (**SELECT** **AVG**(e.salary) **AS** avgsal,  
**COUNT**(\*) **AS** empcount  
**FROM** employee e -- department precedes and  
**WHERE** e.workdept=d.deptno -- **TABLE** is specified  
) **AS** empinfo; -- so d.deptno is known

But the following examples are not valid:

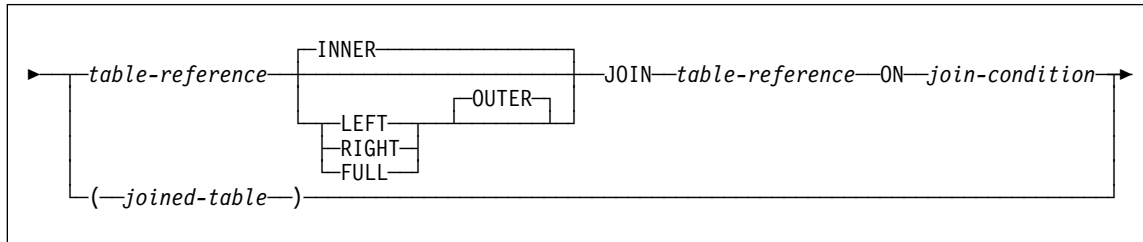
Example 4: **SELECT** t.c1, z.c5  
**FROM** **TABLE**( tf6(t.c2) ) **AS** z, t -- cannot resolve t in t.c2!  
**WHERE** t.c3 = z.c4; -- compare to Example 1 above.

Example 5: **SELECT** a.c1, b.c5  
**FROM** **TABLE**( tf7a(b.c2) ) **AS** a, **TABLE**( tf7b(a.c6) ) **AS** b  
**WHERE** a.c3 = b.c4; -- cannot resolve b in b.c2!

Example 6: **SELECT** d.deptno, d.deptname,  
empinfo.avgsal, empinfo.empcount  
**FROM** department d,  
(**SELECT** **AVG**(e.salary) **AS** avgsal,  
**COUNT**(\*) **AS** empcount  
**FROM** employee e -- department precedes but  
**WHERE** e.workdept=d.deptno -- **TABLE** is not specified  
) **AS** empinfo; -- so d.deptno is unknown

## joined-table

## joined-table



A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.

Inner joins can be thought of as the cross product of the tables (combine each row of the left table with every row of the right table), keeping only the rows where the join-condition is true. The result table may be missing rows from either or both of the joined tables. Outer joins include the inner join and preserve these missing rows. There are three types of outer joins:

1. **left outer join** includes rows from the left table that were missing from the inner join.
2. **right outer join** includes rows from the right table that were missing from the inner join.
3. **full outer join** includes rows from both the left and right tables that were missing from the inner join.

If a join-operator is not specified, INNER is implicit. The order in which multiple joins are performed can affect the result. Joins can be nested within other joins. The order of processing for joins is generally from left to right, but based on the position of the required join-condition. Parentheses are recommended to make the order of nested joins more readable. For example:

```
tb1 left join tb2 on tb1.c1=tb2.c1
  right join tb3 left join tb4 on tb3.c1=tb4.c1
    on tb1.c1=tb3.c1
```

is the same as:

```
(tb1 left join tb2 on tb1.c1=tb2.c1)
  right join (tb3 left join tb4 on tb3.c1=tb4.c1)
    on tb1.c1=tb3.c1
```

A joined table can be used in any context in which any form of the SELECT statement is used. A view or a cursor is read-only if its SELECT statement includes a joined table.

A *join-condition* is a *search-condition* except that:

- it cannot contain any subqueries, scalar or otherwise

- it cannot include any dereference operations or the Deref function where the reference value is other than the object identifier column.
- any column referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause)
- any function referenced in an expression of the *join-condition* of a full outer join must be deterministic and have no external action.

An error occurs if the join-condition does not comply with these rules (SQLSTATE 42972).

Column references are resolved using the rules for resolution of column name qualifiers. The same rules that apply to predicates apply to *join-conditions* (see “Predicates” on page 135).

### Join Operations

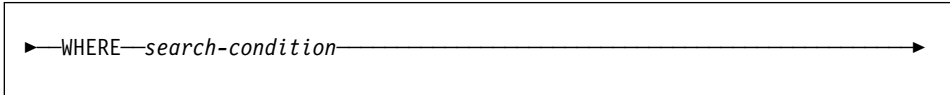
A *join-condition* specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of the JOIN operator of the *join-condition*. For all possible combinations of rows of T1 and T2, a row of T1 is paired with a row of T2 if the *join-condition* is true. When a row of T1 is joined 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 result of the join operations:

- The result of T1 INNER JOIN T2 consists of their paired rows where the join-condition is true.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows where the join-condition is true 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 where the join-condition is true 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 T2, the concatenation of that row with the null row of T1 and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T1 and T2 allow null values.

## where-clause

### where-clause



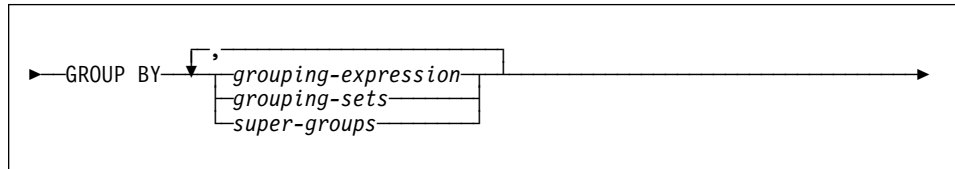
The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the subselect.

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 of the subselect.

In its simplest form, a GROUP BY clause contains a *grouping expression*. A *grouping expression* is an *expression* used in defining the grouping of R. Each *column-name* included in *grouping-expression* must unambiguously identify a column of R (SQLSTATE 42702 or 42703). The length attribute of each *grouping-expression* must not be more than 254 bytes (SQLSTATE 42907). A *grouping-expression* cannot include a scalar-fullselect (SQLSTATE 42822) or any function that is variant or has an external action (SQLSTATE 42845).

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of these forms, see “grouping-sets” on page 330 and “super-groups” on page 331, respectively.

The result of GROUP BY is a set of groups of rows. Each row in this result represents the set of rows for which the *grouping-expression* is equal. For grouping, all null values from a *grouping-expression* are considered equal.

A *grouping-expression* can be used in a search condition in a HAVING clause, in an expression in a SELECT clause or in a *sort-key-expression* of an ORDER BY clause (see “order-by-clause” on page 358 for details). In each case, the reference specifies only one value for each group. For example, if the *grouping-expression* is *iscol1+col2* then an allowed expression in the select list would be *col1+col2+3*. Associativity rules for expressions would disallow the similar expression, *3+col1+col2*, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, *3+(col1+col2)* would also be allowed in the select list. If the concatenation operator is used, the *grouping-expression* must be used exactly as the expression was specified in the select list.

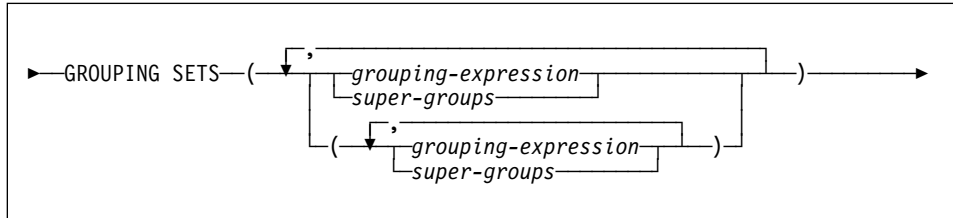
If the *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and may not all have the same length. In that case, a reference to the *grouping-expression* still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

As noted, there are some cases where the GROUP BY clause cannot refer directly to a column that is specified in the SELECT clause as an expression (scalar-fullselect, variant or external action functions). To group using such an expression, use a nested table expression or a common table expression to first provide a result table with the

## group-by-clause

expression as a column of the result. For an example using nested table expressions, see Example A9 on page 338.

## grouping-sets



A *grouping-sets* specification allows multiple grouping clauses to be specified in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element or can be a list of elements delimited by parentheses, where an element is either a grouping-expression or a super-group. Using *grouping-sets* allows the groups to be computed with a single pass over the base table.

The *grouping-sets* specification allows either a simple *grouping-expression* to be used, or the more complex forms of *super-groups*. For a description of *super-groups*, see “super-groups” on page 331.

Note that grouping sets are the fundamental building block for GROUP BY operations. A simple group by with a single column can be considered a grouping set with one element. For example:

```
GROUP BY a
```

is the same as

```
GROUP BY GROUPING SET((a))
```

and

```
GROUP BY a,b,c
```

is the same as

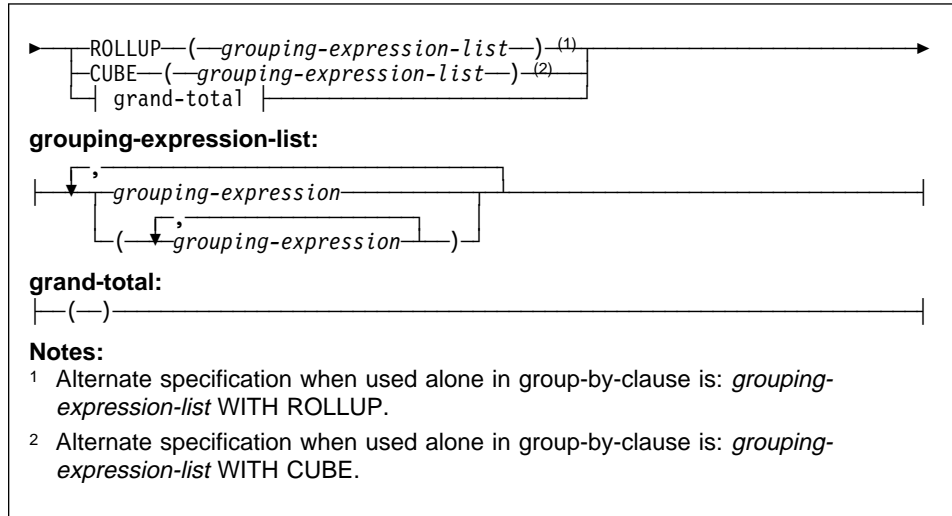
```
GROUP BY GROUPING SET((a,b,c))
```

Non-aggregation columns from the select list of the subselect that are excluded from a grouping set will return a null for such columns for each row generated for that grouping set. This reflects the fact that aggregation was done without considering the values for those columns. See “GROUPING” on page 176 for how to distinguish rows with nulls in actual data from rows with nulls generated from grouping sets.

Example C2 on page 342 through Example C7 on page 346 illustrate the use of grouping sets.



super-groups



**ROLLUP** ( *grouping-expression-list* )

A *ROLLUP grouping* is an extension to the GROUP BY clause that produces a result set that contains *sub-total* rows in addition to the "regular" grouped rows. *Sub-total* rows<sup>42</sup> are "super-aggregate" rows that contain further aggregates whose values are derived by applying the same column functions that were used to obtain the grouped rows.

A ROLLUP grouping is a series of *grouping-sets*. The general specification of a ROLLUP with *n* elements

**GROUP BY ROLLUP**(*C*<sub>1</sub>,*C*<sub>2</sub>,...,*C*<sub>*n*-1</sub>,*C*<sub>*n*</sub>)

is equivalent to

**GROUP BY GROUPING SETS**((*C*<sub>1</sub>,*C*<sub>2</sub>,...,*C*<sub>*n*-1</sub>,*C*<sub>*n*</sub>)  
 (*C*<sub>1</sub>,*C*<sub>2</sub>,...,*C*<sub>*n*-1</sub>)  
 ...  
 (*C*<sub>1</sub>,*C*<sub>2</sub>)  
 (*C*<sub>1</sub>)  
 ( ) )

Notice that the *n* elements of the ROLLUP translate to *n*+1 grouping sets.

Note that the order in which the *grouping-expressions* is specified is significant for ROLLUP. For example:

**GROUP BY ROLLUP**(*a*,*b*)

is equivalent to

<sup>42</sup> These are called sub-total rows, because that is their most common use, however any column function can be used for the aggregation. For instance, MAX and AVG are used in Example C8 on page 348.

## group-by-clause

```
GROUP BY GROUPING SETS((a,b)
                        (a)
                        ( ) )
```

while

```
GROUP BY ROLLUP(b,a)
```

is the same as

```
GROUP BY GROUPING SETS((b,a)
                        (b)
                        ( ) )
```

The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example C3 on page 343 illustrates the use of ROLLUP.

### **CUBE** ( *grouping-expression-list* )

A *CUBE grouping* is an extension to the GROUP BY clause that produces a result set that contains all the rows of a ROLLUP aggregation and, in addition, contains "cross-tabulation" rows. *Cross-tabulation* rows are additional "super-aggregate" rows that are not part of an aggregation with sub-totals.

Like a ROLLUP, a CUBE grouping can also be thought of as a series of *grouping-sets*. In the case of a CUBE, all permutations of the cubed *grouping-expression-list* are computed along with the grand total. Therefore, the  $n$  elements of a CUBE translate to  $2^{**}n$  (2 to the power  $n$ ) *grouping-sets*. For instance, a specification of

```
GROUP BY CUBE(a,b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                        (a,b)
                        (a,c)
                        (b,c)
                        (a)
                        (b)
                        (c)
                        ( ) )
```

Notice that the 3 elements of the CUBE translate to 8 grouping sets.

The order of specification of elements does not matter for CUBE. 'CUBE (DayOfYear, Sales\_Person)' and 'CUBE (Sales\_Person, DayOfYear)' yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example C4 on page 343 illustrates the use of CUBE.

### *grouping-expression-list*

A *grouping-expression-list* is used within a CUBE or ROLLUP clause to define the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple *grouping-expressions*.

The rules for a *grouping-expression* are described in “group-by-clause” on page 329. For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However the clause:

```
GROUP BY ROLLUP(Province, County, City)
```

results in unwanted sub-total rows for the County. In the clause

```
GROUP BY ROLLUP(Province, (County, City))
```

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the desired result. In other words, the two element ROLLUP

```
GROUP BY ROLLUP(Province, (County, City))
```

generates

```
GROUP BY GROUPING SETS((Province, County, City)
                          (Province)
                          () )
```

while the 3 element ROLLUP would generate

```
GROUP BY GROUPING SETS((Province, County, City)
                          (Province, County)
                          (Province)
                          () )
```

Example C2 on page 342 also utilizes composite column values.

### grand-total

Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This may be separately specified with empty parentheses within the GROUPING SET clause. It may also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. Example C4 on page 343 uses the grand-total syntax.

### Combining Grouping Sets

This can be used to combine any of the types of GROUP BY clauses. When simple *grouping-expression* fields are combined with other groups, they are “appended” to the beginning of the resulting *grouping sets*. When ROLLUP or CUBE expressions are combined, they operate like “multipliers” on the remaining expression, forming additional grouping set entries according to the definition of either ROLLUP or CUBE.

For instance, combining *grouping-expression* elements acts as follows:

```
GROUP BY a, ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                          (a,b)
                          (a) )
```

Or similarly,

```
GROUP BY a, b, ROLLUP(c,d)
```

## group-by-clause

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                          (a,b,c)
                          (a,b) )
```

Combining of *ROLLUP* elements acts as follows:

```
GROUP BY ROLLUP(a), ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                          (a,b)
                          (a)
                          (b,c)
                          (b)
                          () )
```

Similarly,

```
GROUP BY ROLLUP(a), CUBE(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                          (a,b)
                          (a,c)
                          (a)
                          (b,c)
                          (b)
                          (c)
                          () )
```

Combining of *CUBE* and *ROLLUP* elements acts as follows:

```
GROUP BY CUBE(a,b), ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                          (a,b,c)
                          (a,b)
                          (a,c,d)
                          (a,c)
                          (a)
                          (b,c,d)
                          (b,c)
                          (b)
                          (c,d)
                          (c)
                          () )
```

Like a simple *grouping-expression*, combining grouping sets also eliminates duplicates within each grouping set. For instance,

```
GROUP BY a, ROLLUP(a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b)
                          (a) )
```

A more complete example of combining grouping sets is to construct a result set that eliminates certain rows that would be returned for a full CUBE aggregation.

For example, consider the following GROUP BY clause:

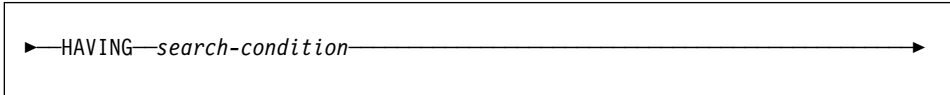
```
GROUP BY Region,
          ROLLUP(Sales_Person, WEEK(Sales_Date)),
          CUBE(YEAR(Sales_Date), MONTH (Sales_Date))
```

The column listed immediately to the right of GROUP BY is simply grouped, those within the parenthesis following ROLLUP are rolled up, and those within the parenthesis following CUBE are cubed. Thus, the above clause results in a cube of MONTH within YEAR which is then rolled up within WEEK within Sales\_Person within the Region aggregation. It does not result in any grand total row or any cross-tabulation rows on Region, Sales\_Person or WEEK(Sales\_Date) so produces fewer rows than the clause:

```
GROUP BY ROLLUP (Region, Sales_Person, WEEK(Sales_Date),
                  YEAR(Sales_Date), MONTH(Sales_Date) )
```

## having-clause

## having-clause



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 of the subselect. If this clause is not GROUP BY, R is considered a single group with no grouping columns.

Each *column-name* in the search condition must do one of the following:

- Unambiguously identify a grouping column of R.
- Be specified within a column function.
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table or view identified in an outer subselect.

A group of R to which the search condition is applied supplies the argument for each column 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 A6 on page 337 and Example A7 on page 338.

A correlated reference to a group of R must either identify a grouping column or be contained within a column function.

When HAVING is used without GROUP BY, the select list can only be a column name within a column function, a correlated column reference, a literal, or a special register.

## Examples of subselects

*Example A1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

*Example A2:* Join the EMP\_ACT and EMPLOYEE tables, select all the columns from the EMP\_ACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

```
SELECT EMP_ACT.*, LASTNAME
FROM EMP_ACT, EMPLOYEE
WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO
```

*Example A3:* Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee surname (LASTNAME), department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the DEPARTMENT table) and department name (DEPTNAME) of all employees who were born (BIRTHDATE) earlier than 1930.

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
FROM EMPLOYEE, DEPARTMENT
WHERE WORKDEPT = DEPTNO
AND YEAR(BIRTHDATE) < 1930
```

*Example A4:* Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27000.

```
SELECT JOB, MIN(SALARY), MAX(SALARY)
FROM EMPLOYEE
GROUP BY JOB
HAVING COUNT(*) > 1
AND MAX(SALARY) >= 27000
```

*Example A5:* Select all the rows of EMP\_ACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

```
SELECT *
FROM EMP_ACT
WHERE EMPNO IN
    (SELECT EMPNO
     FROM EMPLOYEE
     WHERE WORKDEPT = 'E11')
```

*Example A6:* From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.

```
SELECT WORKDEPT, MAX(SALARY)
FROM EMPLOYEE
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                     FROM EMPLOYEE)
```

The subquery in the HAVING clause would only be executed once in this example.

*Example A7:* Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.

```
SELECT WORKDEPT, MAX(SALARY)
FROM EMPLOYEE EMP_COR
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                      FROM EMPLOYEE
                      WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

In contrast to Example A6 on page 337, the subquery in the HAVING clause would need to be executed for each group.

*Example A8:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments.

This query must first create a nested table expression (DINFO) in order to get the AVGSALARY and EMPCOUNT columns, as well as the DEPTNO column that is used in the WHERE clause.

```
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
FROM EMPLOYEE THIS_EMP,
     (SELECT OTHERS.WORKDEPT AS DEPTNO,
          AVG(OTHERS.SALARY) AS AVGSALARY,
          COUNT(*) AS EMPCOUNT
      FROM EMPLOYEE OTHERS
      GROUP BY OTHERS.WORKDEPT
     ) AS DINFO
WHERE THIS_EMP.JOB = 'SALESREP'
AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

Using a nested table expression for this case saves the overhead of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the query, only the rows for the department of the sales representatives need to be considered by the view.

*Example A9:* Display the average education level and salary for 5 random groups of employees.

This query requires the use of a nested table expression to set a random value for each employee so that it can subsequently be used in the GROUP BY clause.

```
SELECT RANDID , AVG(EDLEVEL), AVG(SALARY)
FROM (
  SELECT EDLEVEL, SALARY, INTEGER(RAND()*5) AS RANDID
  FROM EMPLOYEE
 ) AS EMPRAND
GROUP BY RANDID
```



## Examples of Joins

*Example B1:* This example illustrates the results of the various joins using tables J1 and J2. These tables contain rows as shown.

```
SELECT * FROM J1
```

W	X
A	11
B	12
C	13

```
SELECT * FROM J2
```

Y	Z
A	21
C	22
D	23

The following query does an inner join of J1 and J2 matching the first column of both tables.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y
```

W	X	Y	Z
A	11	A	21
C	13	C	22

In this inner join example the row with column W='C' from J1 and the row with column Y='D' from J2 are not included in the result because they do not have a match in the other table. Note that the following alternative form of an inner join query produces the same result.

```
SELECT * FROM J1, J2 WHERE W=Y
```

The following left outer join will get back the missing row from J1 with nulls for the columns of J2. Every row from J1 is included.

```
SELECT * FROM J1 LEFT OUTER JOIN J2 ON W=Y
```

W	X	Y	Z
A	11	A	21
B	12	-	-
C	13	C	22

The following right outer join will get back the missing row from J2 with nulls for the columns of J1. Every row from J2 is included.

```
SELECT * FROM J1 RIGHT OUTER JOIN J2 ON W=Y
```

W	X	Y	Z
A	11	A	21
C	13	C	22
-	-	D	23

The following full outer join will get back the missing rows from both J1 and J2 with nulls where appropriate. Every row from both J1 and J2 is included.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
```

W	X	Y	Z
A	11	A	21
C	13	C	22
-	-	D	23
B	12	-	-

*Example B2:* Using the tables J1 and J2 from the previous example, examine what happens when an additional predicate is added to the search condition.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=13
```

W	X	Y	Z
C	13	C	22

The additional condition caused the inner join to select only 1 row compared to the inner join in Example B1 on page 339.

Notice what the impact of this is on the full outer join.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=13
```

W	X	Y	Z
-	-	A	21
C	13	C	22
-	-	D	23
A	11	-	-
B	12	-	-

The result now has 5 rows (compared to 4 without the additional predicate) since there was only 1 row in the inner join and all rows of both tables must be returned.

The following query illustrates that placing the same additional predicate in WHERE clause has completely different results.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
WHERE X=13
```

W	X	Y	Z
C	13	C	22

The WHERE clause is applied after the intermediate result of the full outer join. This intermediate result would be the same as the result of the full outer join query in Example B1 on page 339. The WHERE clause is applied to this intermediate result and eliminates all but the row that has X=13. Choosing the location of a predicate when performing outer joins can have significant impact on the results. Consider what happens if the predicate was X=12 instead of X=13. The following inner join returns no rows.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=12
```

Hence, the full outer join would return 6 rows, 3 from J1 with nulls for the columns of J2 and 3 from J2 with nulls for the columns of J1.,

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=12
```

W	X	Y	Z
-	-	A	21
-	-	C	22
-	-	D	23
A	11	-	-
B	12	-	-
C	13	-	-

If the additional predicate is in the WHERE clause instead, 1 row is returned.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
WHERE X=12
```

W	X	Y	Z
B	12	-	-

*Example B3:* List every department with the employee number and last name of the manager, including departments without a manager.

```
SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
ON MGRNO = EMPNO
```

*Example B4:* List every employee number and last name with the employee number and last name of their manager, including employees without a manager.

```
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E LEFT OUTER JOIN
DEPARTMENT INNER JOIN EMPLOYEE M
ON MGRNO = M.EMPNO
ON E.WORKDEPT = DEPTNO
```

The inner join determines the last name for any manager identified in the DEPARTMENT table and the left outer join guarantees that each employee is listed even if a corresponding department is not found in DEPARTMENT.

## Examples of Grouping Sets, Cube, and Rollup

The queries in Example C1 on page 342 through Example C4 on page 343 use a subset of the rows in the SALES tables based on the predicate 'WEEK(SALES\_DATE) = 13'.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SALES AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
```

which results in:

WEEK	DAY_WEEK	SALES_PERSON	UNITS_SOLD
13	6	LUCCHESSI	3
13	6	LUCCHESSI	1
13	6	LEE	2
13	6	LEE	2
13	6	LEE	3
13	6	LEE	5
13	6	GOUNOT	3
13	6	GOUNOT	1
13	6	GOUNOT	7
13	7	LUCCHESSI	1
13	7	LUCCHESSI	2
13	7	LUCCHESSI	1
13	7	LEE	7
13	7	LEE	3
13	7	LEE	7
13	7	LEE	4
13	7	GOUNOT	2
13	7	GOUNOT	18
13	7	GOUNOT	1

*Example C1:* Here is a query with a basic GROUP BY clause over 3 columns:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

WEEK	DAY_WEEK	SALES_PERSON	UNITS_SOLD
13	6	GOUNOT	11
13	6	LEE	12
13	6	LUCCHESSI	4
13	7	GOUNOT	21
13	7	LEE	21
13	7	LUCCHESSI	4

*Example C2:* Produce the result based on two different grouping sets of rows from the SALES table.

```

SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY GROUPING SETS ( (WEEK(SALES_DATE), SALES_PERSON),
                          (DAYOFWEEK(SALES_DATE), SALES_PERSON))
ORDER BY WEEK, DAY_WEEK, SALES_PERSON

```

This results in:

WEEK	DAY_WEEK	SALES_PERSON	UNITS_SOLD
13	-	GOUNOT	32
13	-	LEE	33
13	-	LUCCHESSI	8
-	6	GOUNOT	11
-	6	LEE	12
-	6	LUCCHESSI	4
-	7	GOUNOT	21
-	7	LEE	21
-	7	LUCCHESSI	4

The rows with WEEK 13 are from the first grouping set and the other rows are from the second grouping set.

*Example C3:* If you use the 3 distinct columns involved in the grouping sets of Example C2 on page 342 and perform a ROLLUP, you can see grouping sets for (WEEK, DAY\_WEEK, SALES\_PERSON), (WEEK, DAY\_WEEK), (WEEK) and grand total.

```

SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON

```

This results in:

WEEK	DAY_WEEK	SALES_PERSON	UNITS_SOLD
13	6	GOUNOT	11
13	6	LEE	12
13	6	LUCCHESSI	4
13	6	-	27
13	7	GOUNOT	21
13	7	LEE	21
13	7	LUCCHESSI	4
13	7	-	46
13	-	-	73
-	-	-	73

*Example C4:* If you run the same query as Example C3 on page 343 only replace ROLLUP with CUBE, you can see additional grouping sets for (WEEK, SALES\_PERSON), (DAY\_WEEK, SALES\_PERSON), (DAY\_WEEK), (SALES\_PERSON) in the result.

```

SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY CUBE ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON

```

This results in:

WEEK	DAY_WEEK	SALES_PERSON	UNITS_SOLD
13	6	GOUNOT	11
13	6	LEE	12
13	6	LUCCHESSI	4
13	6	-	27
13	7	GOUNOT	21
13	7	LEE	21
13	7	LUCCHESSI	4
13	7	-	46
13	-	GOUNOT	32
13	-	LEE	33
13	-	LUCCHESSI	8
13	-	-	73
-	6	GOUNOT	11
-	6	LEE	12
-	6	LUCCHESSI	4
-	6	-	27
-	7	GOUNOT	21
-	7	LEE	21
-	7	LUCCHESSI	4
-	7	-	46
-	-	GOUNOT	32
-	-	LEE	33
-	-	LUCCHESSI	8
-	-	-	73

*Example C5:* Obtain a result set which includes a grand-total of selected rows from the SALES table together with a group of rows aggregated by SALES\_PERSON and MONTH.

```

SELECT SALES_PERSON,
       MONTH(SALES_DATE) AS MONTH,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( (SALES_PERSON, MONTH(SALES_DATE)),
                        ( )
                      )
ORDER BY SALES_PERSON, MONTH

```

This results in:

SALES_PERSON	MONTH	UNITS_SOLD
GOUNOT	3	35
GOUNOT	4	14
GOUNOT	12	1
LEE	3	60
LEE	4	25
LEE	12	6
LUCCHESSI	3	9
LUCCHESSI	4	4
LUCCHESSI	12	1
-	-	155

*Example C6:* This example shows two simple ROLLUP queries followed by a query which treats the two ROLLUPs as grouping sets in a single result set and specifies row ordering for each column involved in the grouping sets.

*Example C6-1:*

```

SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) )
ORDER BY WEEK, DAY_WEEK

```

results in:

WEEK	DAY_WEEK	UNITS_SOLD
13	6	27
13	7	46
13	-	73
14	1	31
14	2	43
14	-	74
53	1	8
53	-	8
-	-	155

*Example C6-2:*

```

SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( MONTH(SALES_DATE), REGION );
ORDER BY MONTH, REGION

```

results in:

MONTH	REGION	UNITS_SOLD
3	Manitoba	22
3	Ontario-North	8
3	Ontario-South	34
3	Quebec	40
3	-	104
4	Manitoba	17
4	Ontario-North	1
4	Ontario-South	14
4	Quebec	11
4	-	43

```

12 Manitoba                2
12 Ontario-South          4
12 Quebec                  2
12 -                       8
- -                       155

```

*Example C6-3:*

```

SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) ),
                        ROLLUP( MONTH(SALES_DATE), REGION ) )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION

```

results in:

WEEK	DAY_WEEK	MONTH	REGION	UNITS_SOLD
13	6	- -	- -	27
13	7	- -	- -	46
13	-	- -	- -	73
14	1	- -	- -	31
14	2	- -	- -	43
14	-	- -	- -	74
53	1	- -	- -	8
53	-	- -	- -	8
-	-	3	Manitoba	22
-	-	3	Ontario-North	8
-	-	3	Ontario-South	34
-	-	3	Quebec	40
-	-	3	-	104
-	-	4	Manitoba	17
-	-	4	Ontario-North	1
-	-	4	Ontario-South	14
-	-	4	Quebec	11
-	-	4	-	43
-	-	12	Manitoba	2
-	-	12	Ontario-South	4
-	-	12	Quebec	2
-	-	12	-	8
-	-	- -	- -	155
-	-	- -	- -	155

Using the two ROLLUPs as grouping sets causes the result to include duplicate rows. There are even two grand total rows.

Observe how the use of ORDER BY has affected the results:

- In the first grouped set, week 53 has been repositioned to the end.
- In the second grouped set, month 12 has now been positioned to the end and the regions now appear in alphabetic order.
- Null values are sorted high.

*Example C7:* In queries that perform multiple rollups in a single pass (such as Example C6-3 on page 346) you may want to be able to indicate which grouping set produced each row. The following steps demonstrate how to provide a column (called



GROUP) which indicates the origin of each row in the result set. By origin, we mean which one of the two grouping sets produced the row in the result set.

*Step 1:* Introduce a way of "generating" new data values, using a query which selects from a VALUES clause (which is an alternate form of a fullselect). This query shows how a table can be derived called "X" having 2 columns "R1" and "R2" and 1 row of data.

```
SELECT R1,R2
FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2);
```

results in:

```
R1      R2
-----
GROUP 1 GROUP 2
```

*Step 2:* Form the cross product of this table "X" with the SALES table. This add columns "R1" and "R2" to every row.

```
SELECT R1, R2, WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SALES AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
```

This add columns "R1" and "R2" to every row.

*Step 3:* Now we can combine these columns with the grouping sets to include these columns in the rollup analysis.

```
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE)) ),
                        (R2, ROLLUP( MONTH(SALES_DATE), REGION ) ) )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

R1	R2	WEEK	DAY_WEEK	MONTH	REGION	UNITS_SOLD
GROUP 1	-	13	6	-	-	27
GROUP 1	-	13	7	-	-	46
GROUP 1	-	13	-	-	-	73
GROUP 1	-	14	1	-	-	31
GROUP 1	-	14	2	-	-	43
GROUP 1	-	14	-	-	-	74
GROUP 1	-	53	1	-	-	8
GROUP 1	-	53	-	-	-	8
-	GROUP 2	-	-	-	3 Manitoba	22
-	GROUP 2	-	-	-	3 Ontario-North	8
-	GROUP 2	-	-	-	3 Ontario-South	34
-	GROUP 2	-	-	-	3 Quebec	40
-	GROUP 2	-	-	-	3 -	104
-	GROUP 2	-	-	-	4 Manitoba	17

-	GROUP 2	-	-	4 Ontario-North	1
-	GROUP 2	-	-	4 Ontario-South	14
-	GROUP 2	-	-	4 Quebec	11
-	GROUP 2	-	-	4 -	43
-	GROUP 2	-	-	12 Manitoba	2
-	GROUP 2	-	-	12 Ontario-South	4
-	GROUP 2	-	-	12 Quebec	2
-	GROUP 2	-	-	12 -	8
-	GROUP 2	-	-	-	155
GROUP 1	-	-	-	-	155

*Step 4:* Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```

SELECT COALESCE(R1,R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES, (VALUES ('GROUP 1', 'GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE)) ),
                        (R2, ROLLUP( MONTH(SALES_DATE), REGION ) ) )
ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION;

```

results in:

GROUP	WEEK	DAY_WEEK	MONTH	REGION	UNITS_SOLD
GROUP 1	13	6	-	-	27
GROUP 1	13	7	-	-	46
GROUP 1	13	-	-	-	73
GROUP 1	14	1	-	-	31
GROUP 1	14	2	-	-	43
GROUP 1	14	-	-	-	74
GROUP 1	53	1	-	-	8
GROUP 1	53	-	-	-	8
GROUP 1	-	-	-	-	155
GROUP 2	-	-	-	3 Manitoba	22
GROUP 2	-	-	-	3 Ontario-North	8
GROUP 2	-	-	-	3 Ontario-South	34
GROUP 2	-	-	-	3 Quebec	40
GROUP 2	-	-	-	3 -	104
GROUP 2	-	-	-	4 Manitoba	17
GROUP 2	-	-	-	4 Ontario-North	1
GROUP 2	-	-	-	4 Ontario-South	14
GROUP 2	-	-	-	4 Quebec	11
GROUP 2	-	-	-	4 -	43
GROUP 2	-	-	-	12 Manitoba	2
GROUP 2	-	-	-	12 Ontario-South	4
GROUP 2	-	-	-	12 Quebec	2
GROUP 2	-	-	-	12 -	8
GROUP 2	-	-	-	-	155

*Example C8:* The following example illustrates the use of various column functions when performing a CUBE. The example also makes use of cast functions and rounding to produce a decimal result with reasonable precision and scale.

```

SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD,
       MAX(SALES) AS BEST_SALE,
       CAST(ROUND(AVG(DECIMAL(SALES)),2) AS DECIMAL(5,2)) AS AVG_UNITS_SOLD
FROM SALES
GROUP BY CUBE(MONTH(SALES_DATE),REGION)
ORDER BY MONTH, REGION

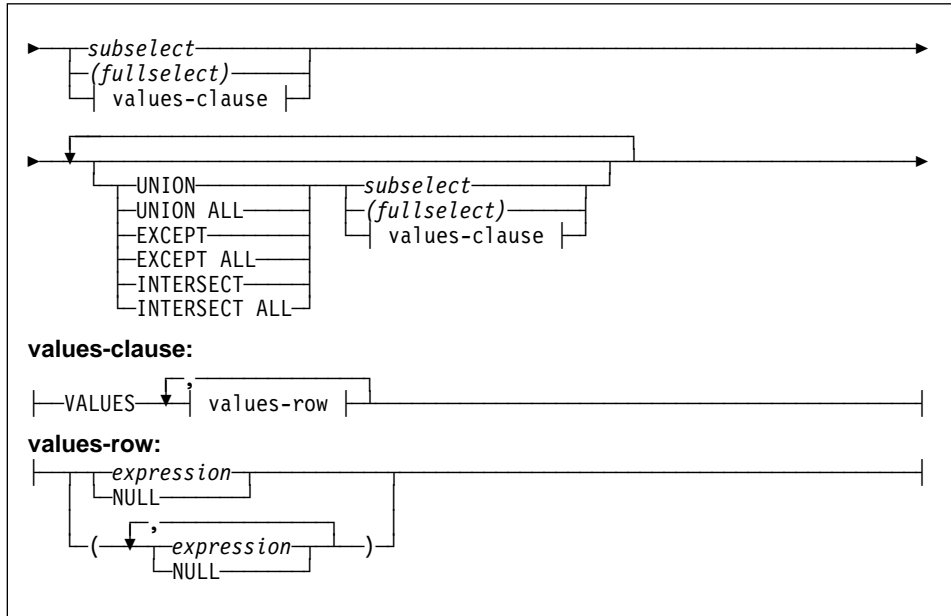
```

This results in:

MONTH	REGION	UNITS_SOLD	BEST_SALE	AVG_UNITS_SOLD
3	Manitoba	22	7	3.14
3	Ontario-North	8	3	2.67
3	Ontario-South	34	14	4.25
3	Quebec	40	18	5.00
3	-	104	18	4.00
4	Manitoba	17	9	5.67
4	Ontario-North	1	1	1.00
4	Ontario-South	14	8	4.67
4	Quebec	11	8	5.50
4	-	43	9	4.78
12	Manitoba	2	2	2.00
12	Ontario-South	4	3	2.00
12	Quebec	2	1	1.00
12	-	8	3	1.60
-	Manitoba	41	9	3.73
-	Ontario-North	9	3	2.25
-	Ontario-South	52	14	4.00
-	Quebec	53	18	4.42
-	-	155	18	3.87

## fullselect

### fullselect



The *fullselect* is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn are components of a statement. A *fullselect* that is a component of a predicate is called a *subquery*. A *fullselect* that is enclosed in parentheses is sometimes called a subquery.

The set operators UNION, EXCEPT, and INTERSECT correspond to the relational operators union, difference, and intersection.

A *fullselect* specifies a result table. If a set operator is not used, the result of the *fullselect* is the result of the specified subselect or values-clause.

#### values-clause

Derives a result table by specifying the actual values, using expressions, for each column of a row in the result table. Multiple rows may be specified.

NULL can only be used with multiple *values-rows* and at least one row in the same column must not be NULL (SQLSTATE 42826).

A *values-row* is specified by:

- A single expression for a single column result table or,
- n expressions (or NULL) separated by commas and enclosed in parentheses, where n is the number of columns in the result table.

A multiple row VALUES clause must have the same number of expressions in each *values-row* (SQLSTATE 42826).

The following are examples of values-clauses and their meaning.

VALUES (1),(2),(3)	- 3 rows of 1 column
VALUES 1, 2, 3	- 3 rows of 1 column
VALUES (1, 2, 3)	- 1 row of 3 columns
VALUES (1,21),(2,22),(3,23)	- 3 rows of 2 columns

A values-clause that is composed of  $n$  *values-rows*,  $RE_1$  to  $RE_n$ , where  $n$  is greater than 1, is equivalent to

$RE_1$  UNION ALL  $RE_2$  ... UNION ALL  $RE_n$

This means that the corresponding expressions of each *values-row* must be comparable (SQLSTATE 42825) and the resulting data type is based on “Rules for Result Data Types” on page 82.

#### 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 the duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

#### EXCEPT or EXCEPT ALL

Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

#### INTERSECT or INTERSECT ALL

Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

The number of columns in the result tables R1 and R1 must be the same (SQLSTATE 42826).

The columns of the result are named as follows:

- If the  $n$ th column of R1 and the  $n$ th column of R2 have the same result column name, then the  $n$ th column of R has the result column name.
- If the  $n$ th column of R1 and the  $n$ th column of R2 have different result column names, a name is generated. This name cannot be used as the column name in an ORDER BY or UPDATE clause.

The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

## fullselect

Two rows are duplicates of one another if each value in the first is equal to the corresponding value of the second. (For determining duplicates, two null values are considered equal.)

When multiple operations are combined in an expression, operations within parentheses are performed first. If there are no parentheses, the operations are performed from left to right with the exception that all INTERSECT operations are performed before UNION or EXCEPT operations.

In the following example, the values of tables R1 and R2 are shown on the left. The other headings listed show the values as a result of various set operations on R1 and R2.

R1	R2	UNION		EXCEPT		INTER-SECT	
		ALL	UNION	ALL	EXCEPT	ALL	INTER-SECT
1	1	1	1	1	2	1	1
1	1	1	2	2	5	1	3
1	3	1	3	2		3	4
2	3	1	4	2		4	
2	3	1	5	4			
2	3	2		5			
3	4	2					
4		2					
4		3					
5		3					
		3					
		3					
		3					
		4					
		4					
		4					
		5					

For the rules on how the data types of the result columns are determined, see “Rules for Result Data Types” on page 82.

For the rules on how conversions of string columns are handled, see “Rules for String Conversions” on page 85.

## Examples of a fullselect

*Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

*Example 2:* List the employee numbers (EMPNO) of all employees in the EMPLOYEE table whose department number (WORKDEPT) either begins with 'E' or who are assigned to projects in the EMP\_ACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```

SELECT EMPNO
  FROM EMPLOYEE
 WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO
  FROM EMP_ACT
 WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')

```

*Example 3:* Make the same query as in example 2, and, in addition, “tag” the rows from the EMPLOYEE table with 'emp' and the rows from the EMP\_ACT table with 'emp\_act'. Unlike the result from example 2, this query may return the same EMPNO more than once, identifying which table it came from by the associated “tag”.

```

SELECT EMPNO, 'emp'
  FROM EMPLOYEE
 WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act' FROM EMP_ACT
 WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')

```

*Example 4:* Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

```

SELECT EMPNO
  FROM EMPLOYEE
 WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO
  FROM EMP_ACT
 WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')

```

*Example 5:* Make the same query as in Example 3, only include an additional two employees currently not in any table and tag these rows as "new".

```

SELECT EMPNO, 'emp'
  FROM EMPLOYEE
 WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act'
  FROM EMP_ACT
 WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
UNION
VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')

```

*Example 6:* This example of EXCEPT produces all rows that are in T1 but not in T2.

```

(SELECT * FROM T1)
EXCEPT ALL
(SELECT * FROM T2)

```

If no NULL values are involved, this example returns the same results as

```

SELECT ALL *
  FROM T1
 WHERE NOT EXISTS (SELECT * FROM T2
                   WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)

```

## fullselect

*Example 7:* This example of INTERSECT produces all rows that are in both tables T1 and T2, removing duplicates.

```
(SELECT * FROM T1)
INTERSECT
(SELECT * FROM T2)
```

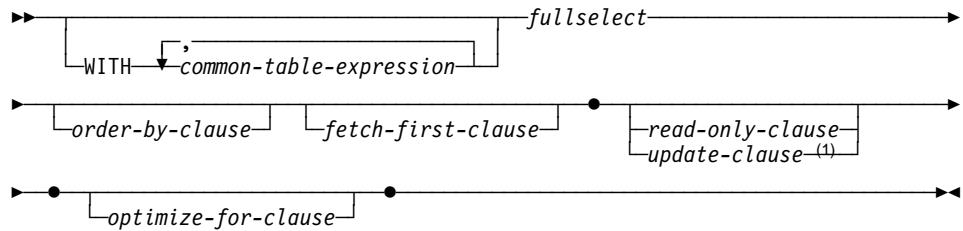
If no NULL values are involved, this example returns the same result as

```
SELECT DISTINCT * FROM T1
  WHERE EXISTS (SELECT * FROM T2
               WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

where C1, C2, and so on represent the columns of T1 and T2.



---

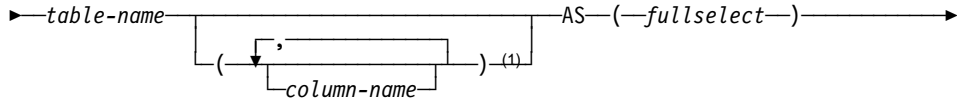
**select-statement**
**Note:**

- <sup>1</sup> The update-clause and the order-by-clause cannot both be specified in the same select-statement.

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 through the use of dynamic SQL statements using the command line processor (or similar tools), causing a result table to be displayed on the user's screen. In either case, the table specified by a *select-statement* is the result of the fullselect.

## common-table-expression

### common-table-expression



#### Note:

- <sup>1</sup> If a common table expression is recursive, or if the fullselect results in duplicate column names, column names must be specified.

A *common table expression* permits defining a result table with a *table-name* that can be specified as a table name in any FROM clause of the fullselect that follows. Multiple common table expressions can be specified following the single WITH keyword. Each common table expression specified can also be referenced by name in the FROM clause of subsequent common table expressions.

If a list of columns is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the fullselect used to define the common table expression.

The *table-name* of a *common table expression* must be different from any other common table expression *table-name* in the same statement (SQLSTATE 42726). If the common table expression is specified in an INSERT statement the *table-name* cannot be the same as the table or view name that is the object of the insert (SQLSTATE 42726). A common table expression *table-name* can be specified as a table name in any FROM clause throughout the fullselect. A *table-name* of a common table expression overrides any existing table, view or alias (in the catalog) with the same qualified name.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted (SQLSTATE 42835). A *cyclic reference* occurs when two common table expressions *dt1* and *dt2* are created such that *dt1* refers to *dt2* and *dt2* refers to *dt1*.

The *common table expression* is also optional prior to the fullselect in the CREATE VIEW and INSERT statements.

A *common table expression* can be used:

- In place of a view to avoid creating the view (when general use of the view is not required and positioned updates or deletes are not used)
- To enable grouping by a column that is derived from a scalar subselect or function that is not deterministic or has external action
- When the desired result table is based on host variables
- When the same result table needs to be shared in a *fullselect*
- When the result needs to be derived using recursion.

## common-table-expression

If a *fullselect* of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning. For an example, see Appendix M, “Recursion Example: Bill of Materials” on page 961.

The following must be true of a recursive common table expression:

- Each fullselect that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed (SQLSTATE 42925). Furthermore, the unions must use UNION ALL (SQLSTATE 42925).
- The column names must be specified following the *table-name* of the common table expression (SQLSTATE 42908).
- The first fullselect of the first union (the initialization fullselect) must not include a reference to any column of the common table expression in any FROM clause (SQLSTATE 42836).
- If a column name of the common table expression is referred to in the iterative fullselect, the data type, length, and code page for the column are determined based on the initialization fullselect. The corresponding column in the iterative fullselect must have the same data type and length as the data type and length determined based on the initialization fullselect and the code page must match (SQLSTATE 42825). However, for character string types, the length of the two data types may differ. In this case, the column in the iterative fullselect must have a length that would always be assignable to the length determined from the initialization fullselect.
- Each fullselect that is part of the recursion cycle must not include any column functions, group-by-clauses, or having-clauses (SQLSTATE 42836).

The FROM clauses of these fullselects can include at most one reference to a common table expression that is part of a recursion cycle (SQLSTATE 42836).

- Subqueries (scalar or quantified) must not be part of any recursion cycles (SQLSTATE 42836).

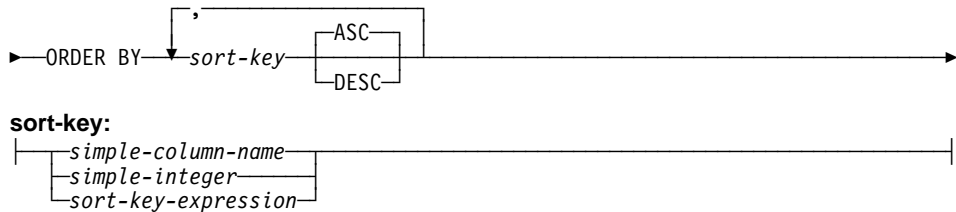
When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will terminate. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:

- In the iterative fullselect, an integer column incremented by a constant.
- A predicate in the where clause of the iterative fullselect in the form "counter\_col < constant" or "counter \_col < :hostvar".

A warning is issued if this syntax is not found in the recursive common table expression (SQLSTATE 01605).

## order-by-clause

### order-by-clause



The ORDER BY clause specifies an ordering of the rows of the result table. If a single sort specification (one *sort-key* with associated direction) is identified, the rows are ordered by the values of that sort specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified sort specification, then by the values of the second identified sort specification, and so on. The length attribute of each *sort-key* must not be more than 254 characters for a character column or 127 characters for a graphic column (SQLSTATE 42907).

A named column in the select list may be identified by a *sort-key* that is a *simple-integer* or a *simple-column-name*. An unnamed column in the select list must be identified by an *simple-integer* or, in some cases, by a *sort-key-expression* that matches the expression in the select list (see details of *sort-key-expression*). 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.<sup>43</sup>

Ordering is performed in accordance with the comparison rules described in Chapter 3. The null value is higher than all other values. If the ORDER BY clause does not completely order the rows, rows with duplicate values of all identified columns are displayed in an arbitrary order.

#### *simple-column-name*

Usually identifies a column of the result table. In this case, *simple-column-name* must be the column name of a named column in the select list.

The *simple-column-name* may also identify a column name of a table, view or nested table identified in the FROM clause if the query is a subselect. An error occurs if the subselect:

- specifies DISTINCT in the select-clause (SQLSTATE 42822)
- produces a grouped result and the *simple-column-name* is not a *grouping-expression* (SQLSTATE 42803).

Determining which column is used for ordering the result is described under "Column name in sort keys" (see "Notes" on page 359).

<sup>43</sup> The rules for determining the name of result columns for a fullselect that involves set operators (UNION, INTERSECT, or EXCEPT) can be found in "fullselect" on page 350.

### *simple-integer*

Must be greater than 0 and not greater than the number of columns in the result table (SQLSTATE 42805). The integer *n* identifies the *n*th column of the result table.

### *sort-key-expression*

An expression that is not simply a column name or an unsigned integer constant. The query to which ordering is applied must be a *subselect* to use this form of sort-key. The *sort-key-expression* cannot include a correlated scalar-fullselect (SQLSTATE 42703) or a function with an external action (SQLSTATE 42845).

Any column-name within a *sort-key-expression* must conform to the rules described under "Column names in sort keys" (see "Notes").

There are a number of special cases that further restrict the expressions that can be specified.

- DISTINCT is specified in the SELECT clause of the subselect (SQLSTATE 42822).

The sort-key-expression must match exactly with an expression in the select list of the subselect (scalar-fullselects are never matched).

- The subselect is grouped (SQLSTATE 42803).

The sort-key-expression can:

- be an expression in the select list of the subselect,
- include a *grouping-expression* from the GROUP BY clause of the subselect
- include a column function, constant or host variable.

### **ASC**

Uses the values of the column in ascending order. This is the default.

### **DESC**

Uses the values of the column in descending order.

### **Notes**

- **Column names in sort keys:**

- The column name is qualified.

The query must be a *subselect* (SQLSTATE 42877). The column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the subselect (SQLSTATE 42702). The value of the column is used to compute the value of the sort specification.

- The column name is unqualified.

- The query is a subselect.

If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the ordering

## order-by-clause

subselect (SQLSTATE 42702). If the column name is identical to one column, that column is used to compute the value of the sort specification. If the column name is not identical to a column of the result table, then it must unambiguously identify a column of some table, view or nested table in the FROM clause of the fullselect in the select-statement (SQLSTATE 42702).

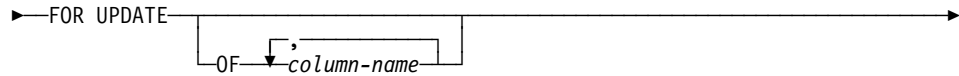
- The query is not a subselect ( it includes set operations such as union, except or intersect ).

The column name must not be identical to the name of more than one column of the result table (SQLSTATE 42702). The column name must be identical to exactly one column of the result table (SQLSTATE 42707) and this column is used to compute the value of the sort specification.

See “Column Name Qualifiers to Avoid Ambiguity” on page 101 for more information on qualified column names.

- **Limits:** The use of a *sort-key-expression* or a *simple-column-name* where the column is not in the select list may result in the addition of the column or expression to the temporary table used for sorting. This may result in reaching the limit of the number of columns in a table or the limit on the size of a row in a table. Exceeding these limits will result in an error if a temporary table is required to perform the sorting operation.

## update-clause



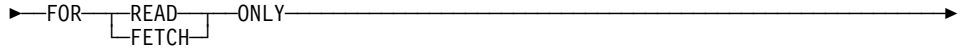
The FOR UPDATE clause identifies the columns that can be updated in a subsequent Positioned UPDATE statement. Each *column-name* must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect. If the FOR UPDATE clause is specified without column names, all updatable columns of the table or view identified in the first FROM clause of the fullselect are included.

The FOR UPDATE clause cannot be used if one of the following is true:

- The cursor associated with the select-statement is not deletable (see “Notes” on page 597).
- One of the selected columns is a non-updatable column of a catalog table and the FOR UPDATE clause has not been used to exclude that column.

## read-only-clause

### read-only-clause



The FOR READ ONLY clause indicates that the result table is read-only and therefore the cursor cannot be referred to in Positioned UPDATE and DELETE statements. FOR FETCH ONLY has the same meaning.

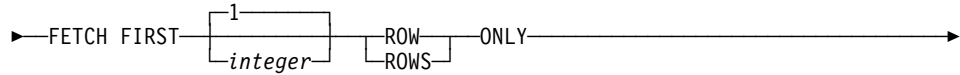
Some result tables are read-only by nature. (For example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY (or FOR FETCH ONLY) can possibly improve the performance of FETCH operations by allowing the database manager to do blocking and avoid exclusive locks. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors as if the FOR UPDATE clause was specified. It is recommended, therefore, that the FOR READ ONLY clause be used to improve performance except in cases where queries will be used in a Positioned UPDATE or DELETE statements.

A read-only result table must not be referred to in a Positioned UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY (FOR FETCH ONLY). See “DECLARE CURSOR” on page 595 for more information about read-only and updatable cursors.



**fetch-first-clause**



The *fetch-first-clause* sets a maximum number of rows that can be retrieved. It lets the database manager know that the application does not want to retrieve more than *integer* rows, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond *integer* rows is handled the same way as normal end of data (SQLSTATE 02000, SQLCODE +100). The value of *integer* must be a positive integer (not zero).

Limiting the result table to the first *integer* rows can improve performance. The database manager will cease processing the query once it has determined the first *integer* rows. If both the *fetch-first-clause* and the *optimize-for-clause* are specified, the lower of the *integer* values from these clauses will be used to influence the communications buffer size. The values are considered independently for optimization purposes.

Specification of the *fetch-first-clause* in a select-statement makes the cursor not deletable (read-only). This clause cannot be specified with the FOR UPDATE clause.

## optimize-for-clause

### optimize-for-clause

▶—OPTIMIZE FOR—*integer*—

ROWS
ROW

—▶

The OPTIMIZE FOR clause requests special processing of the *select statement*. If the clause is omitted, it is assumed that all rows of the result table will be retrieved; if it is specified, it is assumed that the number of rows retrieved will probably not exceed *n* where *n* is the value for *integer*. The value of *n* must be a positive integer. Use of the OPTIMIZE FOR clause influences query optimization based on the assumption that *n* rows will be retrieved. In addition, for cursors that are blocked, this clause will influence the number of rows that will be returned in each block (ie. no more than *n* rows will be returned in each block).

This clause does not limit the number of rows that can be fetched or affect the result in any other way than performance. Using OPTIMIZE FOR *n* ROWS can improve the performance if no more than *n* rows are retrieved, but may degrade performance if more than *n* rows are retrieved.

If the value of *n* multiplied by the size of the row, exceeds the size of the communication buffer<sup>44</sup> the OPTIMIZE FOR clause will have no impact on the data buffers.

---

<sup>44</sup> The size of the communication buffer is defined by the RQRIOLBK or the ASLHEAPSZ configuration parameter. See the *Administration Guide* for details.

## Examples of a select-statement

*Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

*Example 2:* Select the project name (PROJNAME), start date (PRSTDATE), and end date (PRENDATE) from the PROJECT table. Order the result table by the end date with the most recent dates appearing first.

```
SELECT PROJNAME, PRSTDATE, PRENDATE
FROM PROJECT
ORDER BY PRENDATE DESC
```

*Example 3:* Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the EMPLOYEE table. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
FROM EMPLOYEE
GROUP BY WORKDEPT
ORDER BY 2
```

*Example 4:* Declare a cursor named UP\_CUR to be used in a C program to update the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row.

```
EXEC SQL DECLARE UP_CUR CURSOR FOR
        SELECT PROJNO, PRSTDATE, PRENDATE
        FROM PROJECT
        FOR UPDATE OF PRSTDATE, PRENDATE;
```

*Example 5:* This example names the expression SAL+BONUS+COMM as TOTAL\_PAY

```
SELECT SALARY+BONUS+COMM AS TOTAL_PAY
FROM EMPLOYEE
ORDER BY TOTAL_PAY
```

*Example 6:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments. Also, list the average salary of the department with the highest average salary.

Using a common table expression for this case saves the overhead of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the fullselect, only the rows for the department of the sales representatives need to be considered by the view.

```

WITH
  DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
    (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
     FROM EMPLOYEE OTHERS
     GROUP BY OTHERS.WORKDEPT
    ),
  DINFOMAX AS
    (SELECT MAX(AVGSALARY) AS AVGMAX FROM DINFO)
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY,
       DINFO.AVGSALARY, DINFO.EMPCOUNT, DINFOMAX.AVGMAX
FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
WHERE THIS_EMP.JOB = 'SALESREP'
AND THIS_EMP.WORKDEPT = DINFO.DEPTNO

```

## Chapter 6. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements.

Table 15 (Page 1 of 3). SQL Statements

SQL Statement	Function	Page
ALTER BUFFERPOOL	Changes the definition of a buffer pool.	375
ALTER NODEGROUP	Changes the definition of a nodegroup.	377
ALTER TABLE	Changes the definition of a table.	380
ALTER TYPE (Structured)	Changes the definition of a structured type.	403
ALTER VIEW	Changes the definition of a view by altering a reference type column to add a scope.	405
ALTER TABLESPACE	Changes the definition of a table space.	399
BEGIN DECLARE SECTION	Marks the beginning of a host variable declaration section.	407
CALL	Calls a stored procedure.	409
CLOSE	Closes a cursor.	416
COMMENT ON	Replaces or adds a comment to the description of an object.	418
COMMIT	Terminates a unit of work and commits the database changes made by that unit of work.	426
Compound SQL	Combines one or more other SQL statements into an executable block.	428
CONNECT (Type 1)	Connects to an application server according to the rules for remote unit of work.	432
CONNECT (Type 2)	Connects to an application server according to the rules for application-directed distributed unit of work.	439
CREATE ALIAS	Defines an alias for a table, view, or another alias.	446
CREATE BUFFERPOOL	Creates a new buffer pool.	449
CREATE DISTINCT TYPE	Defines a distinct data type.	452
CREATE EVENT MONITOR	Specifies events in the database to monitor.	458
CREATE FUNCTION	Registers a user-defined function.	467
CREATE FUNCTION (External Scalar)	Registers a user-defined external scalar function.	468
CREATE FUNCTION (External Table)	Registers a user-defined external table function.	484
CREATE FUNCTION (Sourced)	Registers a user-defined sourced function.	497
CREATE INDEX	Defines an index on a table.	504
CREATE NODEGROUP	Defines a nodegroup.	508
CREATE PROCEDURE	Registers a stored procedure.	511
CREATE SCHEMA	Defines a schema.	519

Table 15 (Page 2 of 3). SQL Statements

SQL Statement	Function	Page
CREATE TABLE	Defines a table.	522
CREATE TABLESPACE	Defines a table space.	559
CREATE TRIGGER	Defines a trigger.	568
CREATE TYPE (Structured)	Defines a structured data type.	578
CREATE VIEW	Defines a view of one or more tables or views.	582
DECLARE CURSOR	Defines an SQL cursor.	595
DELETE	Deletes one or more rows from a table.	599
DESCRIBE	Describes the result columns of a prepared SELECT statement.	604
DISCONNECT	Terminates one or more connections when there is no active unit of work.	608
DROP	Deletes objects in the database.	611
END DECLARE SECTION	Marks the end of a host variable declaration section.	624
EXECUTE	Executes a prepared SQL statement.	626
EXECUTE IMMEDIATE	Prepares and executes an SQL statement.	631
EXPLAIN	Captures information about the chosen access plan.	633
FETCH	Assigns values of a row to host variables.	637
FREE LOCATOR	Removes the association between a locator variable and its value.	640
GRANT (Database Authorities)	Grants authorities on the entire database.	641
GRANT (Index Privileges)	Grants the CONTROL privilege on indexes in the database.	644
GRANT (Package Privileges)	Grants privileges on packages in the database.	646
GRANT (Schema Privileges)	Grants privileges on a schema.	649
GRANT (Table or View Privileges)	Grants privileges on tables and views.	652
INCLUDE	Inserts code or declarations into a source program.	659
INSERT	Inserts one or more rows into a table.	661
LOCK TABLE	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table.	666
OPEN	Prepares a cursor that will be used to retrieve values when the FETCH statement is issued.	668
PREPARE	Prepares an SQL statement (with optional parameters) for execution.	673
REFRESH TABLE	Refreshes the data in a summary table.	682
RELEASE	Places one or more connections in the release-pending state.	683
RENAME TABLE	Renames an existing table.	685
REVOKE (Database Authorities)	Revokes authorities from the entire database.	687
REVOKE (Index Privileges)	Revokes the CONTROL privilege on given indexes.	690

Table 15 (Page 3 of 3). SQL Statements

SQL Statement	Function	Page
REVOKE (Package Privileges)	Revokes privileges from given packages in the database.	692
REVOKE (Schema Privileges)	Revokes privileges on a schema.	695
REVOKE (Table or View Privileges)	Revokes privileges from given tables or views.	697
ROLLBACK	Terminates a unit of work and backs out the database changes made by that unit of work.	702
SELECT INTO	Specifies a result table of no more than one row and assigns the values to host variables.	705
SET CONNECTION	Changes the state of a connection from dormant to current, making the specified location the current server.	707
SET CONSTRAINTS	Sets the check pending state and checks data for constraint violations.	709
SET CURRENT DEGREE	Changes the value of the CURRENT DEGREE special register.	716
SET CURRENT EXPLAIN MODE	Changes the value of the CURRENT EXPLAIN MODE special register.	718
SET CURRENT EXPLAIN SNAPSHOT	Changes the value of the CURRENT EXPLAIN SNAPSHOT special register.	720
SET PATH	Changes the value of the CURRENT PATH special register.	731
SET CURRENT PACKAGESET	Sets the schema name for package selection.	722
SET CURRENT QUERY OPTIMIZATION	Changes the value of the CURRENT QUERY OPTIMIZATION special register.	724
SET CURRENT REFRESH AGE	Changes the value of the CURRENT REFRESH AGE special register.	727
SET EVENT MONITOR STATE	Activates or deactivates an event monitor.	729
SET SCHEMA	Changes the value of the CURRENT SCHEMA special register.	733
SET transition-variable	Assigns values to NEW transition variables.	735
SIGNAL SQLSTATE	Signals an error.	738
UPDATE	Updates the values of one or more columns in one or more rows of a table.	740
VALUES INTO	Specifies a result table of no more than one row and assigns the values to host variables.	748
WHENEVER	Defines actions to be taken on the basis of SQL return codes.	750

## How SQL Statements Are Invoked

The SQL statements described in this chapter are classified as *executable* or *nonexecutable*. The *Invocation* section in the description of each statement indicates whether or not the statement is executable.

An *executable statement* can be invoked in three ways:

- Embedded in an application program
- Dynamically prepared and executed
- Issued interactively.

**Note:** Statements embedded in REXX are prepared and executed dynamically.

Depending on the statement, some or all of these methods can be used. The *Invocation* section in the description of each statement tells which methods can be used.

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 355.) It is not included in this chapter because it is used differently from other statements.

A *select-statement* can be invoked in three ways:

- Included in DECLARE CURSOR and implicitly executed by OPEN, FETCH and CLOSE
- Dynamically prepared, referenced in DECLARE CURSOR, and implicitly executed by OPEN, FETCH and CLOSE
- Issued interactively.

The first two methods are called, respectively, the *static* and the *dynamic* invocation of *select-statement*.

The different methods of invoking an SQL statement are discussed below in more detail. For each method, the discussion includes the mechanism of execution, interaction with host variables, and testing whether or not the execution was successful.

## Embedding a Statement in an Application Program

SQL statements can be included in a source program that will be submitted to the pre-compiler. Such statements are said to be *embedded* in the program. An embedded statement can be placed anywhere in the program where a host language statement is allowed. Each embedded statement must be preceded by the keywords EXEC and 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, 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 referenced in this way can be used in two ways:

- As input (the current value of the host variable is used in the execution of the statement)
- As output (the variable is assigned a new value as a result of executing the statement).



In particular, all references to host variables in expressions and predicates are effectively replaced by current values of the variables; that is, the variables are used as input. The treatment of other references is described individually for each statement.

All executable statements should be followed by a test of an SQL return code. Alternatively, the *WHENEVER* statement (which is itself nonexecutable) can be used to change the flow of control immediately after the execution of an embedded statement.

All objects referenced in DML statements must exist when the statements are bound to a DB2 Universal Database .

### **Nonexecutable statements**

An embedded nonexecutable statement is processed only by the precompiler. The precompiler reports any errors encountered in the statement. The statement is *never* processed during program execution. Therefore, such statements should not be followed by a test of an SQL return code.

## **Dynamic Preparation and Execution**

An 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 program (for example, input from a workstation ). The statement (other than a *select-statement*) so constructed can be prepared for execution by means of the (embedded) statement *PREPARE* and executed by means of the (embedded) statement *EXECUTE*. Alternatively, the (embedded) statement *EXECUTE IMMEDIATE* can be used to prepare and execute a statement in one step.

A statement that is going to be dynamically prepared must not contain references to host variables. It can instead contain parameter markers. (See “*PREPARE*” on page 673 for rules concerning the 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 626 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 the setting of an SQL return code in the *SQLCA* after the *EXECUTE* (or *EXECUTE IMMEDIATE*) statement. The SQL return code should be checked as described above. See “*SQL Return Codes*” on page 373 for more information.

## **Static Invocation of a *select-statement***

A *select-statement* can be included as a part of the (nonexecutable) statement *DECLARE CURSOR*. Such a statement is executed every time the cursor is opened by means of the (embedded) statement *OPEN*. After the cursor is open, the result table can be retrieved one row at a time by successive executions of the *FETCH* statement.

Used in this way, the *select-statement* can contain references to host variables. These references are effectively replaced by the values that the variables have at the moment of executing OPEN.

### **Dynamic Invocation of a select-statement**

An 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 program (for example, a query obtained from a workstation ). The statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE, and referenced by a (nonexecutable) statement DECLARE CURSOR. The statement is then executed every time the cursor is opened by means of the (embedded) statement OPEN. After the cursor is open, the result table can be retrieved one row at a time by successive executions of the FETCH statement.

Used in this way, the *select-statement* must not contain references to host variables. It can contain parameter markers instead. (See "PREPARE" on page 673 for rules concerning the parameter markers.) The parameter markers are effectively replaced by the values of the host variables specified in the OPEN statement. (See "OPEN" on page 668 for rules concerning this replacement.)

### **Interactive Invocation**

A capability for entering SQL statements from a workstation is part of the architecture of the database manager. A statement entered in this way is said to be issued interactively.

A statement issued interactively must be an executable statement that does not contain parameter markers or references to host variables, because these make sense only in the context of an application program.

---

## SQL Return Codes

An application program containing executable SQL statements can use either the SQLCODE or SQLSTATE values to handle return codes from SQL statements. There are two ways in which an application can get access to these values.

- Include a structure named SQLCA. An SQLCA is provided automatically in REXX. In other languages, an SQLCA can be obtained by using the INCLUDE SQLCA statement.

The SQLCA includes an integer variable named SQLCODE and a character string variable named SQLSTATE.

- When LANGLEVEL SQL92E is specified as a precompile option, a variable SQLCODE or SQLSTATE may be declared in the SQL declare section of the program. If neither of these variables is declared in the SQL declare section, it is assumed that a variable SQLCODE is declared elsewhere in the program. When using LANGLEVEL SQL92E, the program should not have an INCLUDE SQLCA statement.

Occasionally, warning conditions are mentioned in addition to error conditions with respect to return codes. A warning SQLCODE is a positive value and a warning SQLSTATE has the first two characters set to '01'.

## SQLCODE

An SQLCODE is set by the database manager after each SQL statement is executed. All database managers conform to the ISO/ANSI SQL standard, as follows:

- If SQLCODE = 0 and SQLWARN0 is blank, execution was successful.
- If SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data, because the cursor was positioned after the last row of the result table.
- If SQLCODE > 0 and not = 100, execution was successful with a warning.
- If SQLCODE = 0 and SQLWARN0 = 'W', execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

The meaning of SQLCODE values other than 0 and 100 is product-specific. See the *Messages Reference* for the product-specific meanings.

## SQLSTATE

SQLSTATE is also set by the database manager after execution of each SQL statement. Thus, application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE.

SQLSTATE provides application programs with common codes for common error conditions. 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 database managers and is based on the ISO/ANSI SQL92 standard. For a complete list of the possible values of SQLSTATE, see the *Messages Reference*.

---

## SQL Comments

Static SQL statements can include host language or SQL comments. SQL comments are introduced by two hyphens.

These rules apply to the use of SQL comments:

- 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).
- In COBOL, the hyphens must be preceded by a space.

*Example:* This example shows how to include comments in an SQL statement within a C program:

```
EXEC SQL
  CREATE VIEW PRJ_MAXPER      -- projects with most support personnel
  AS SELECT PROJNO, PROJNAME -- number and name of project
  FROM PROJECT
  WHEREDEPTNO = 'E21'      -- systems support dept code
  AND PRSTAFF > 1;
```

## ALTER BUFFERPOOL

The ALTER BUFFERPOOL statement is used to do the following:

- modify the size of the buffer pool on all partitions (or nodes) or on a single partition
- turn on or off the use of extended storage
- add this buffer pool definition to a new nodegroup.

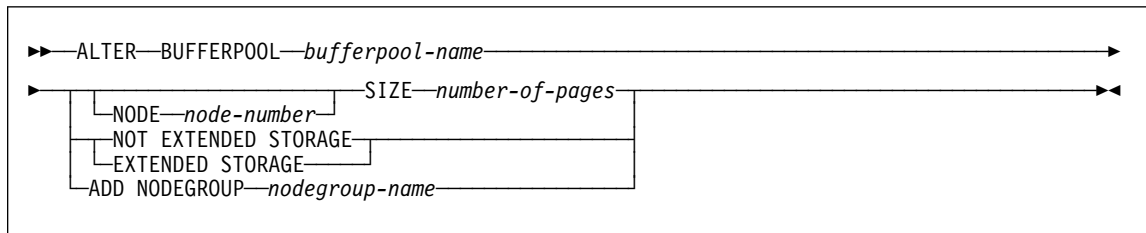
### Invocation

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 ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax



### Description

*bufferpool-name*

Names the buffer pool. This is a one-part name. It is an SQL identifier (either ordinary or delimited). It must be a buffer pool described in the catalog.

**NODE** *node-number*

Specifies the partition on which size of the buffer pool is modified. The partition must be in one of the nodegroups for the buffer pool (SQLSTATE 42729) . If this clause is not specified, then the size of the buffer pool is modified on all partitions on which the buffer pool exists that used the default size for the buffer pool (did not have a size specified in the *except-on-nodes-clause* of the CREATE buffer pool statement).

**SIZE** *number-of-pages*

The size of the buffer pool specified as the number of pages.<sup>45</sup>

**EXTENDED STORAGE**

<sup>45</sup> The size can be specified with a value of (-1) which will indicate that the buffer pool size should be taken from the BUFFPAGE database configuration parameter.

## ALTER BUFFERPOOL

If the extended storage configuration is turned on <sup>46</sup>, pages that are being migrated out of this buffer pool, will be cached in the extended storage.

### NOT EXTENDED STORAGE

Even if the extended storage configuration is turned on, pages that are being migrated out of this buffer pool, will NOT be cached in the extended storage.

### ADD NODEGROUP *nodegroup-name*

Adds this nodegroup to the list of nodegroups to which the buffer pool definition is applicable. Tables spaces in *nodegroup-name* may specify this buffer pool. The nodegroup must currently exist in the database (SQLSTATE 42704).

## Notes

- Although the buffer pool definition is transactional and the changes to the buffer pool definition will be reflected in the catalog tables on commit, no changes to the actual buffer pool will take effect until the next time the database is started. The current attributes of the buffer pool will exist until then, and there will not be any impact to the buffer pool in the interim. Tables created in table spaces of new nodegroups will use the default buffer pool.
- There should be enough real memory on the machine for the total of all the buffer pools, as well as for the rest of the database manager and application requirements.

---

<sup>46</sup> Extended storage configuration is turned on by setting the database configuration parameters NUM\_ESTORE\_SEGS and ESTORE\_SEG\_SIZE to non-zero values. See *Administration Guide* for details.

## ALTER NODEGROUP

The ALTER NODEGROUP statement is used to:

- add one or more partitions or nodes to a nodegroup
- drop one or more partitions from a nodegroup.

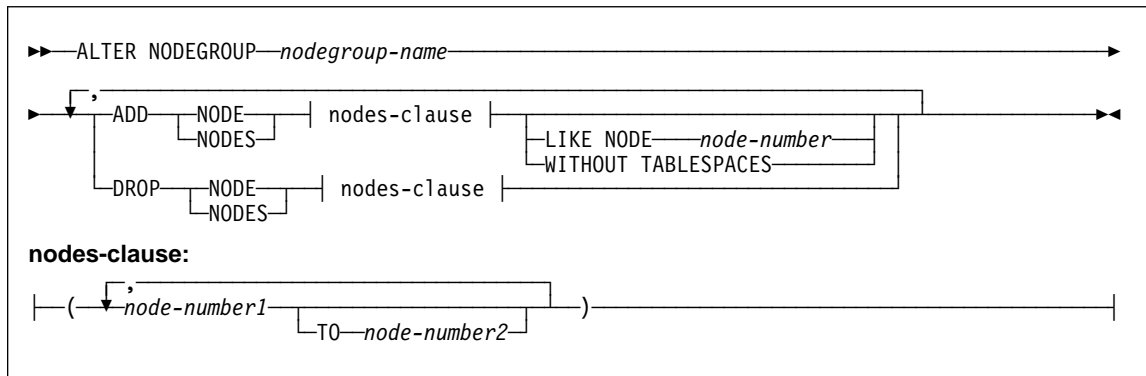
### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be prepared dynamically.

### Authorization

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax



### Description

#### *nodegroup-name*

Names the nodegroup. This is a one-part name. It is an SQL identifier (either ordinary or delimited). It must be a nodegroup described in the catalog.

IBMCATGROUP and IBMTEMPGROUP cannot be specified (SQLSTATE 42832).

#### **ADD NODE**

Specifies the specific partition or partitions to add to the nodegroup. NODES is a synonym for NODE. Any specified partition must not already be defined in the nodegroup (SQLSTATE 42728).

#### **DROP NODE**

Specifies the specific partition or partitions to drop from the nodegroup. NODES is a synonym for NODE. Any specified partition must already be defined in the nodegroup (SQLSTATE 42729).

#### *nodes-clause*

Specifies the partition or partitions to be added or dropped.

## ALTER NODEGROUP

*node-number1*

Specify a specific partition number.

**TO** *node-number2*

Specify a range of partition numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9).

**LIKE NODE** *node-number*

Specifies that the containers for the existing table spaces in the nodegroup will be the same as the containers on the specified *node-number*. The partition specified must be a partition that existed in the nodegroup prior to this statement and is not included in a DROP NODE clause of the same statement.

**WITHOUT TABLESPACES**

Specifies that the default table spaces are not created on the newly added partition or partitions . The ALTER TABLESPACE using the FOR NODE clause must be used to define containers for use with the table spaces that are defined on this nodegroup. If this option is not specified, the default containers are specified on newly added partitions for each table space defined on the nodegroup.

### Rules

- Each partition or node specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729). See “Data Partitioning Across Multiple Partitions” on page 41 for information about this file.
- Each *node-number* listed in the ON NODES clause must be for a unique partition (SQLSTATE 42728).
- A valid partition number is between 0 and 999 inclusive (SQLSTATE 42729).
- A partition cannot appear in both the ADD and DROP clauses (SQLSTATE 42728).
- There must be at least one partition remaining in the nodegroup. The last partition cannot be dropped from a nodegroup (SQLSTATE 428C0) .
- If neither the LIKE NODE clause nor the WITHOUT TABLESPACES clause is specified when adding a partition , the default is to use the lowest partition number of the existing partitions in the nodegroup (say it is 2) and proceed as if LIKE NODE 2 had been specified. For an existing partition to be used as the default it must have containers defined for all the table spaces in the nodegroup (column IN\_USE of SYSCAT.NODEGROUPDEF is not 'I').

### Notes

- When a partition or node is added to a nodegroup, a catalog entry is made for the partition (see SYSCAT.NODEGROUPDEF). The partitioning map is changed immediately to include the new partition along with an indicator (IN\_USE) that the partition is in the partitioning map if either:
  - no table spaces are defined in the nodegroup or
  - no tables are defined in the table spaces defined in the nodegroup and the WITHOUT TABLESPACES clause was not specified.



## ALTER NODEGROUP

The partitioning map is not changed and the indicator (IN\_USE) is set to indicate that the partition is not included in the partitioning map if either:

- tables exist in table spaces in the nodegroup or
- table spaces exist in the nodegroup and the WITHOUT TABLESPACES clause was specified.

To change the partitioning map, the REDISTRIBUTE NODEGROUP command must be used. This redistributes any data, changes the partitioning map, and changes the indicator. Table space containers need to be added before attempting to redistribute data if the WITHOUT TABLESPACES clause was specified.

- When a partition is dropped from a nodegroup, the catalog entry for the partition (see SYSCAT.NODEGROUPDEF) is updated. If there are no tables defined in the table spaces defined in the nodegroup, the partitioning map is changed immediately to exclude the dropped partition and the entry for the partition in the nodegroup is dropped. If tables exist, the partitioning map is not changed and the indicator (IN\_USE) is set to indicate that the partition is waiting to be dropped. The REDISTRIBUTE NODEGROUP command must be used to redistribute the data and drop the entry for the partition from the nodegroup.

### Example

Assume that you have a six- partition database that has the following partitions : 0, 1, 2, 5, 7, and 8. Two partitions are added to the system with partition numbers 3 and 6.

- Assume that you want to add both partitions or nodes 3 and 6 to a nodegroup called MAXGROUP and have the table space containers like those on partition 2. The statement is as follows:

```
ALTER NODEGROUP MAXGROUP  
ADD NODES (3,6) LIKE NODE 2
```

- Assume that you want to drop partition 1 and add partition 6 to nodegroup MEDGROUP. You will define the table space containers separately for partition 6 using ALTER TABLESPACE. The statement is as follows:

```
ALTER NODEGROUP MEDGROUP  
ADD NODE(6) WITHOUT TABLESPACES  
DROP NODE(1)
```

## ALTER TABLE

---

### ALTER TABLE

The ALTER TABLE statement modifies existing tables by:

- Adding one or more columns to a table
- Adding or dropping a primary key
- Adding or dropping one or more unique or referential constraints
- Adding or dropping one or more check constraint definitions
- Altering the length of a VARCHAR column
- Altering a reference type column to add a scope
- Adding or dropping a partitioning key
- Changing table attributes such as the data capture option, pctfree, lock size, or append mode.
- Setting the table to not logged initially state.

### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- ALTER privilege on the table to be altered
- CONTROL privilege on the table to be altered
- ALTERIN privilege on the schema of the table
- SYSADM or DBADM authority.

To create or drop a foreign key, the privileges held by the authorization ID of the statement must include one of the following on the parent table:

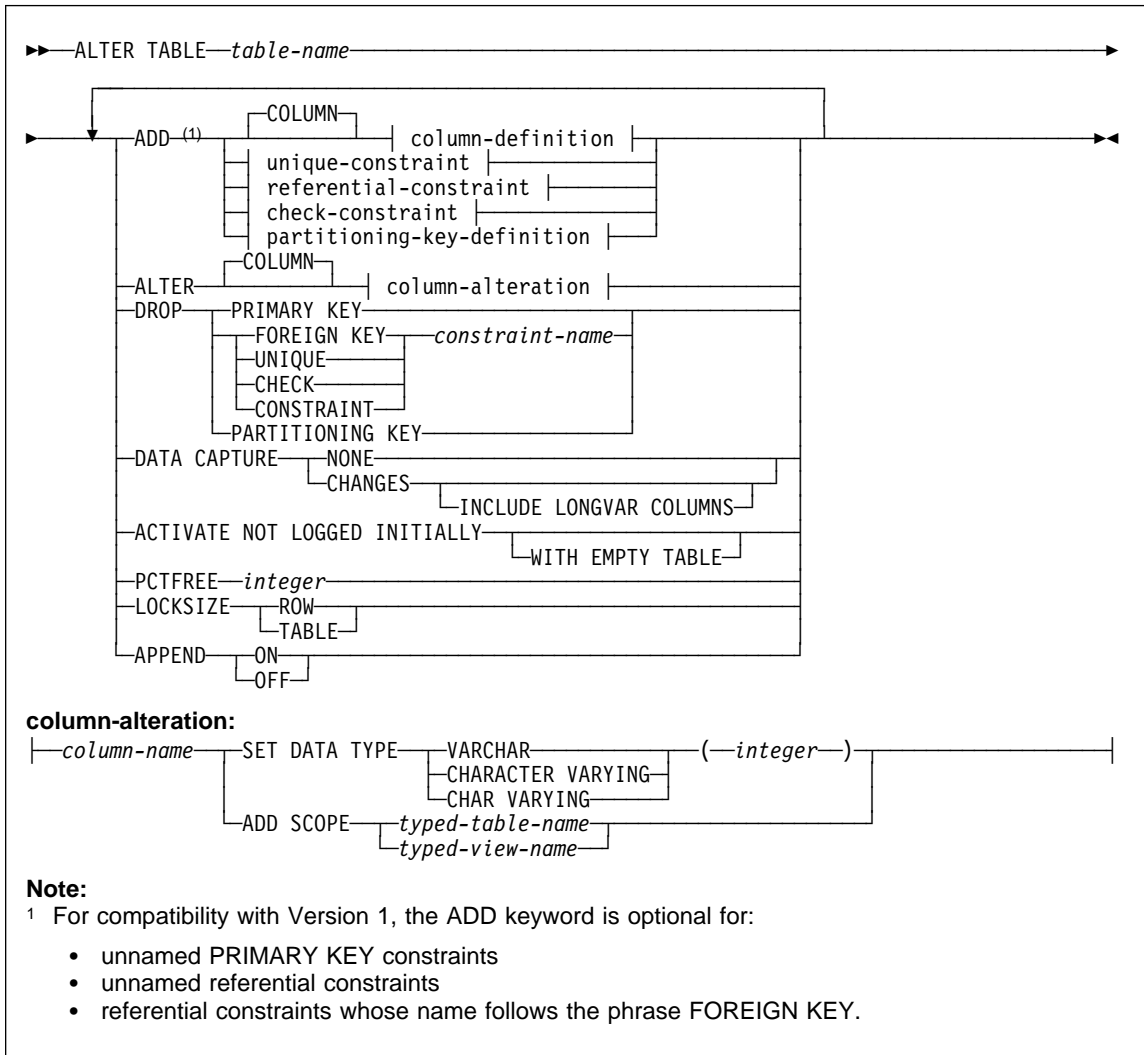
- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SYSADM or DBADM authority.

To drop a primary key or unique constraint of table T, the privileges held by the authorization ID of the statement must include at least one of the following on every table that is a dependent of this parent key of T:

- ALTER privilege on the table
- CONTROL privilege on the table
- ALTERIN privilege on the schema of the table
- SYSADM or DBADM authority.

### Syntax

## ALTER TABLE



## ALTER TABLE

### column-definition:

*column-name* data-type *column-options*

### column-options:

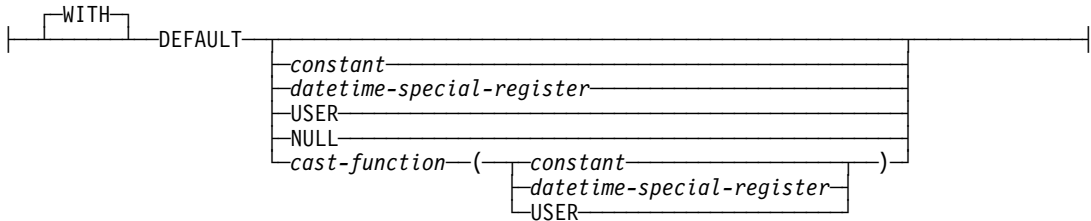
NOT NULL  
default-clause  
lob-options <sup>(1)</sup>  
datalink-options <sup>(2)</sup>  
SCOPE *typed-table-name2* <sup>(3)</sup>  
*typed-view-name2*  
CONSTRAINT <sup>(4)</sup> *constraint-name*  
PRIMARY KEY  
UNIQUE  
references-clause  
CHECK (*check-condition*)

### Notes:

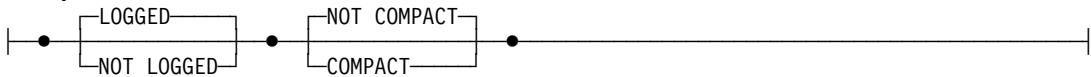
- 1 The lob-options clause only applies to large object types (BLOB, CLOB and DBCLOB) and distinct types based on large object types.
- 2 The datalink-options clause only applies to the DATALINK type and distinct types based on the DATALINK type.
- 3 The SCOPE clause only applies to the REF type.
- 4 For compatibility with Version 1, the CONSTRAINT keyword may be omitted in a *column-definition* defining a references-clause.

## ALTER TABLE

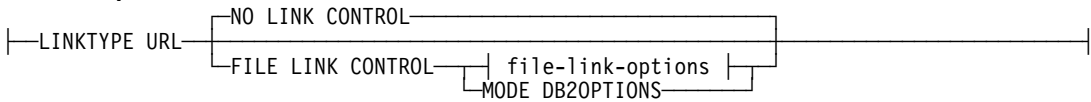
### default-clause:



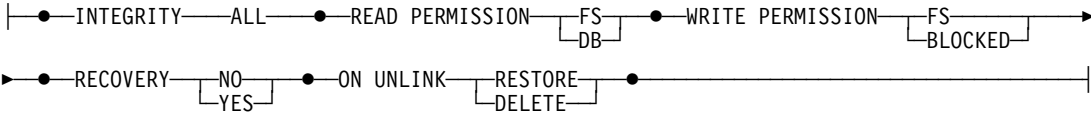
### lob-options:



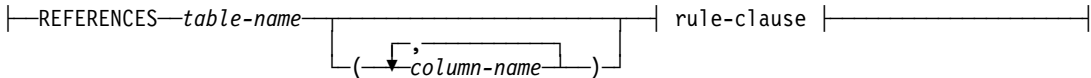
### datalink-options:



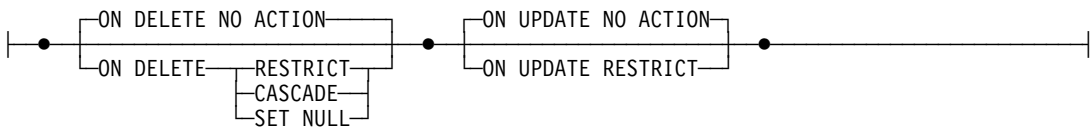
### file-link-options:



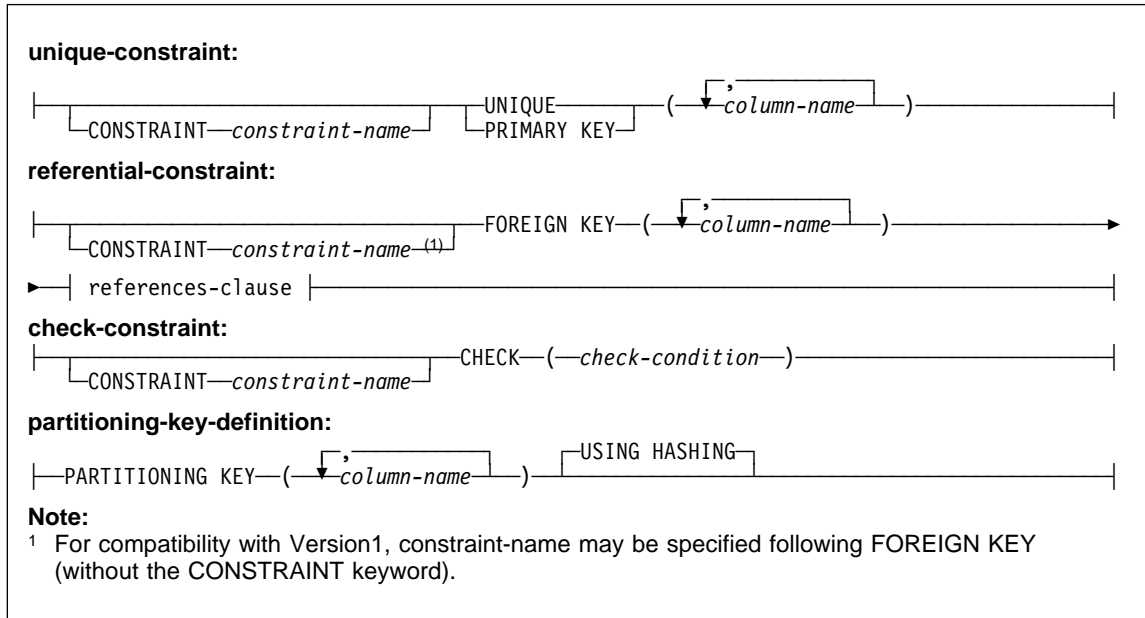
### references-clause:



### rule-clause:



## ALTER TABLE



### Description

*table-name*

Identifies the table to be changed. It must be a table described in the catalog and must not be a summary table, a view or a catalog table.

**ADD** *column-definition*

Adds a column to the table. The table must not be a typed table (SQLSTATE 428DH). If the table has existing rows, every value of the newly added column is its default value. The new column is the 'last' column of the table. That is, if initially there are *n* columns, the added column is column *n*+1. The value of *n* cannot be greater than 499.

Adding the new column must not make the total byte count of all columns exceed the maximum record size of 4005. See "Notes" on page 550 for more information.

*column-name*

Is the name of the column to be added to the table. The name cannot be qualified. Existing column names in the table cannot be used (SQLSTATE 42711).

*data-type*

Is one of the data types listed under "CREATE TABLE" on page 522.

**NOT NULL**

Prevents the column from containing null values. The *default-clause* must also be specified (SQLSTATE 42601).

## ALTER TABLE

### *default-clause*

Specifies a default value for the column.

### **WITH**

An optional keyword.

### **DEFAULT**

Provides a default value in the event a value is not supplied on INSERT or is specified as DEFAULT on INSERT or UPDATE. If a specific default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in Table 16. If a column is defined as a DATALINK, then a specific default value cannot be specified.

If a column is defined using a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

Data Type	Default Value
Numeric	0
Fixed-length character string	Blanks
Varying-length character string	A string of length 0
Fixed-length graphic string	Double-byte blanks
Varying-length graphic string	A string of length 0
Date	For existing rows, a date corresponding to January 1, 0001. For added rows, the current date.
Time	For existing rows, a time corresponding to 0 hours, 0 minutes, and 0 seconds. For added rows, the current time.
Timestamp	For existing rows, a date corresponding to January 1, 0001, and a time corresponding to 0 hours, 0 minutes, 0 seconds and 0 microseconds. For added rows, the current timestamp.
Binary string (blob)	A string of length 0

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column.

Specific types of values that can be specified with the DEFAULT keyword are as follows.

### *constant*

Specifies the constant as the default value for the column. The specified constant must:

## ALTER TABLE

- represent a value that could be assigned to the column in accordance with the rules of assignment as described in Chapter 3
- not be a floating-point constant unless the column is defined with a floating-point data type
- not have non-zero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column)
- be expressed with no more than 254 characters including the quote characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*.

### *datetime-special-register*

Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT or UPDATE as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified). For existing rows, the value is the current date, current time or current timestamp when the ALTER TABLE statement is processed.

### **USER**

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default for the column. If USER is specified, the data type of the column must be a character string with a length not less than the length attribute of USER. For existing rows, the value is the authorization ID of the ALTER TABLE statement.

### **NULL**

Specifies NULL as the default for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same column definition.

### *cast-function*

This form of a default value can only be used with columns defined as a distinct type, BLOB or datetime (DATE, TIME or TIMESTAMP) data type. For distinct type, with the exception of distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function may also be used. For a BLOB or a dis-



## ALTER TABLE

tinct type based on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM. An example using the *cast-function* is given in Example 8 on page 397.

### *constant*

Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the cast-function is BLOB, the constant must be a string constant.

### *datetime-special-register*

Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct type of the column must be the data type that corresponds to the specified special register.

### **USER**

Specifies the USER special register. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

If the value specified is not valid, an error (SQLSTATE 42894) is raised.

### *lob-options*

Specifies options for LOB data types. See *lob-options* in “CREATE TABLE” on page 522.

### *datalink-options*

Specifies options for DATALINK data types. See *datalink-options* in “CREATE TABLE” on page 522.

### **SCOPE**

Specify a scope for a reference type column.

### *typed-table-name2*

The name of a typed table. The data type of *column-name* must be REF(S), where S is the type of *typed-table-name2* (SQLSTATE 428DM). No checking is done of the default value for *column-name* to ensure that the value actually references an existing row in *typed-table-name2*.

### *typed-view-name2*

The name of a typed view. The data type of *column-name* must be REF(S), where S is the type of *typed-view-name2* (SQLSTATE 428DM). No checking is done of the default value for *column-name* to ensure that the values actually references an existing row in *typed-view-name2*.

### **CONSTRAINT** *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that was already specified within the same ALTER TABLE statement, or as the name of any other existing constraint on the table (SQLSTATE 42710).

## ALTER TABLE

If the constraint name is not specified by the user, an 18-character identifier unique within the identifiers of the existing constraints defined on the table, is generated<sup>47</sup> by the system.

When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* may be used as the name of an index that is created to support the constraint. See “Notes” on page 394 for details on index names associated with unique constraints.

### PRIMARY KEY

This 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 were specified as a separate clause. The column cannot contain null values, so the NOT NULL attribute must also be specified (SQLSTATE 42831).

See PRIMARY KEY within the description of the *unique-constraint* below.

### UNIQUE

This 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 were specified as a separate clause.

See UNIQUE within the description of the *unique-constraint* below.

### *references-clause*

This 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.

See *references-clause* on page 542 under CREATE TABLE.

### CHECK (*check-condition*)

This provides a shorthand method of defining a check constraint that applies to a single column. See CHECK (*check-condition*) on page 545 under CREATE TABLE.

### ADD *unique-constraint*

Defines a unique or primary key constraint. A primary key or unique constraint cannot be added to a table that is a subtable (SQLSTATE 429B3). If the table is a supertable at the top of the hierarchy, the constraint applies to the table and all its subtables.

### CONSTRAINT *constraint-name*

Names the primary key or unique constraint. For information on *constraint-name*, see page 387.

---

<sup>47</sup> The identifier is formed of "SQL" followed by a sequence of 15 numeric characters generated by a timestamp-based function.

### **UNIQUE** (*column-name*,...)

Defines a unique key composed of the identified columns. The identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the table and the same column must not be identified more than once. The name cannot be qualified. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may be used as part of a unique key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962). The set of columns in the unique key cannot be the same as the set of columns of the primary key or another unique key (SQLSTATE 01543). Any existing values in the set of identified columns must be unique (SQLSTATE 23515).

A check is performed to determine if an existing index matches the unique key definition (ignoring any INCLUDE columns in the index). An index definition matches if it identifies the same set of columns without regard to the order of the columns or the direction (ASC/DESC) specifications. If a matching index definition is found, the description of the index is changed to indicate that it is required by the system and it is changed to unique (after ensuring uniqueness) if it was a non-unique index. If the table has more than one matching index, an existing unique index is selected (the selection is arbitrary). If no matching index is found, a unique index will automatically be created for the columns, as described in CREATE TABLE. See “Notes” on page 394 for details on index names associated with unique constraints.

### **PRIMARY KEY** (*column-name*,...)

Defines a primary key composed of the identified columns. Each *column-name* must identify a column of the table, and the same column must not be identified more than once. The name cannot be qualified. The number of identified columns must not exceed 16 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. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may be used as part of a primary key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962). The set of columns in the primary key cannot be the same as the set of columns of a unique key (SQLSTATE 01543).<sup>48</sup> Any existing values in the set of identified columns must be unique (SQLSTATE 23515).

A check is performed to determine if an existing index matches the primary key definition (ignoring any INCLUDE columns in the index). An index definition matches if it identifies the same set of columns without regard to the order of the columns or the direction (ASC/DESC) specifications. If a matching index definition is found, the description of the index is changed to indicate that it is the primary index, as required by the system, and it is changed to unique (after ensuring uniqueness) if it was a non-unique index. If the table has

---

<sup>48</sup> If LANGLEVEL is SQL92E or MIA then an error is returned, SQLSTATE 42891.

## ALTER TABLE

more than one matching index, an existing unique index is selected (the selection is arbitrary). If no matching index is found, a unique index will automatically be created for the columns, as described in CREATE TABLE. See “Notes” on page 394 for details on index names associated with unique constraints.

Only one primary key can be defined on a table.

If the table has a partitioning key, the columns of a *unique-constraint* must be a superset of the partitioning key columns; column order is unimportant.

### **ADD** *referential-constraint*

Defines a referential constraint. See *referential-constraint* on page 542 under CREATE TABLE.

### **ADD** *check-constraint*

Defines a check constraint. See *check-constraint* in “CREATE TABLE” on page 522.

### **ADD** *partitioning-key-definition*

Defines a partitioning key. The table must be defined in a table space on a single-partition nodegroup and must not already have a partitioning key. A partitioning key cannot be added to a table that is a subtable (SQLSTATE 428DH).

### **PARTITIONING KEY** (*column-name...*)

Defines a partitioning key using the specified columns. Each *column-name* must identify a column of the table, and the same column must not be identified more than once. The name cannot be qualified. No LONG VARCHAR, LONG VARGRAPHIC, or LOB column may be used as part of a partitioning key (SQLSTATE 42962).

For restrictions related to the partitioning key, see “Rules” on page 394.

### **USING HASHING**

Specifies the use of the hashing function as the partitioning method for data distribution. This is the only partitioning method supported.

### **ALTER** *column-alteration*

Alters the characteristics of a column.

#### *column-name*

Is the name of the column to be altered in the table. The *column-name* must identify an existing column of the table (SQLSTATE 42703). The name cannot be qualified.

### **SET DATA TYPE VARCHAR** (*integer*)

Increase the length of an existing VARCHAR column. CHARACTER VARYING or CHAR VARYING can be used as synonyms for the VARCHAR keyword. The data type of *column-name* must be VARCHAR and the current maximum length defined for the column must not be greater than the value for *integer* (SQLSTATE 42837). The value for *integer* may range up to 4000. The table must not be a typed table (SQLSTATE 428DH).

Altering the column must not make the total byte count of all columns exceed the maximum record size of 4005 (SQLSTATE 54010). See “Notes” on page 550 for more information. If the column is used in a unique constraint or an index, the new size must not cause the sum of the lengths of the columns for the unique constraint or index to exceed 255 (SQLSTATE 54008).

### ADD SCOPE

Add a scope to an existing reference type column that does not already have a scope defined (SQLSTATE 428DK). If the table being altered is a typed table, the column must not be inherited from a supertable (SQLSTATE 428DJ). Refer to “ALTER TYPE (Structured)” on page 403 for examples.

#### *typed-table-name*

The name of a typed table. The data type of *column-name* must be REF(S), where S is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

#### *typed-view-name*

The name of a typed view. The data type of *column-name* must be REF(S), where S is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

### DROP PRIMARY KEY

Drops the definition of the primary key and all referential constraints dependent on this primary key. The table must have a primary key.

### DROP FOREIGN KEY *constraint-name*

Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint. For information on implications of dropping a referential constraint see “Notes” on page 394.

### DROP UNIQUE *constraint-name*

Drops the definition of the unique constraint "*constraint-name*" and all referential constraints dependent on this unique constraint. The *constraint-name* must identify an existing UNIQUE constraint. For information on implications of dropping a unique constraint see “Notes” on page 394.

### DROP CONSTRAINT *constraint-name*

Drops the constraint *constraint-name*. The *constraint-name* must identify an existing check constraint, referential constraint, primary key or unique constraint defined on the table. For information on implications of dropping a constraint see “Notes” on page 394.

### DROP CHECK *constraint-name*

Drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint defined on the table.

### DROP PARTITIONING KEY

Drops the partitioning key. The table must have a partitioning key and must be in a table space defined on a single-partition nodegroup.

## ALTER TABLE

### DATA CAPTURE

Indicates whether extra information for data replication is to be written to the log.

#### NONE

Indicates that no extra information will be logged.

### CHANGES

Indicates that extra information regarding SQL changes to this table will be written to the log. This option is required if this table will be replicated and the Capture program is used to capture changes for this table from the log.

If the table is defined to allow data on a partition other than the catalog partition (multiple partition nodegroup or nodgroup with partition other than the catalog partition), then this option is not supported (SQLSTATE 42997).

Further information about using replication can be found in the *Administration Guide* and the *DB2 Replication Guide and Reference*.

### INCLUDE LONGVAR COLUMNS

Allows data replication utilities to capture changes made to LONG VARCHAR or LONG VARGRAPHIC columns. The clause may be specified for tables that do not have any LONG VARCHAR or LONG VARGRAPHIC columns since it is possible to ALTER the table to include such columns.

### ACTIVATE NOT LOGGED INITIALLY

Activates the NOT LOGGED INITIALLY attribute of the table for this current unit of work. The table must have been originally created with the NOT LOGGED INITIALLY attribute (SQLSTATE 429AA).

Any changes made to the table by an INSERT, DELETE, UPDATE, CREATE INDEX, DROP INDEX, or ALTER TABLE in the same unit of work after the table is altered by this statement are not logged. Any changes made to the system catalog by the ALTER statement in which the NOT LOGGED INITIALLY attribute is activated are logged. Any subsequent changes made in the same unit of work to the system catalog information are logged.

At the completion of the current unit of work, the NOT LOGGED INITIALLY attribute is deactivated and all operations that are done on the table in subsequent units of work are logged.

If using this feature to avoid locks on the catalog tables while inserting data, it is important that only this clause be specified on the ALTER TABLE statement. Use of any other clause in the ALTER TABLE statement will result in catalog locks. If no other clauses are specified for the ALTER TABLE statement, then only a SHARE lock will be acquired on the system catalog tables. This can greatly reduce the possibility of concurrency conflicts for the duration of time between when this statement is executed and when the unit of work in which it was executed is ended.

For more information on the NOT LOGGED INITIALLY attribute, see the description of this attribute in "CREATE TABLE" on page 522.

## ALTER TABLE

**Note:** An error in any operation in the unit of work in which the NOT LOGGED INITIALLY attribute is active will result in the entire unit of work being rolled back (SQLSTATE 40506). Furthermore, the table for which the NOT LOGGED INITIALLY attribute was activated is **marked inaccessible after the rollback** has occurred and can only be dropped. Therefore, the opportunity for errors within the unit of work in which the NOT LOGGED INITIALLY attribute is activated should be minimized.

### WITH EMPTY TABLE

Causes all data currently in table to be removed. Once the data has been removed, it cannot be recovered except through use of the RESTORE facility. If the unit of work in which this Alter statement was issued is rolled back, the table data will NOT be returned to its original state.

When this action is requested, no DELETE triggers defined on the affected table are fired. Any indexes that exist on the table are also emptied.

### PCTFREE *integer*

Indicates what percentage of each page to leave as free space during load or reorganization. The value of *integer* can range from 0 to 99. The first row on each page is added without restriction. When additional rows are added, at least *integer* percent of free space is left on each page. The PCTFREE value is considered only by the LOAD and REORGANIZE TABLE utilities.

### LOCKSIZE

Indicates the size (granularity) of locks used when the table is accessed. Use of this option in the table definition will not prevent normal lock escalation from occurring.

### ROW

Indicates the use of row locks. This is the default lock size when a table is created.

### TABLE

Indicates the use of table locks. This means that the appropriate share or exclusive lock is acquired on the table and intent locks (except intent none) are not used. Use of this value may improve the performance of queries by limiting the number of locks that need to be acquired. However, concurrency is also reduced since all locks are held over the complete table.

Further information about locking can be found in the *Administration Guide*.

### APPEND

Indicates whether data is appended to the end of the table data or placed where free space is available in data pages.

### ON

Indicates that table data will be appended and information about free space on pages will not be kept. The table must not have a clustered index (SQLSTATE 428CA).

## ALTER TABLE

### OFF

Indicates that table data will be placed where there is available space. This is the default when a table is created.

The table should be reorganized after setting APPEND OFF since the information about available free space is not accurate and may result in poor performance during insert.

### Rules

- A partitioning key column of a table cannot be updated (SQLSTATE 42997).
- Any unique or primary key constraint defined on the table must be a superset of the partitioning key, if there is one (SQLSTATE 42997).
- A nullable column of a partitioning key cannot be included as a foreign key column when the relationship is defined with ON DELETE SET NULL (SQLSTATE 42997).
- A column can only be referenced in one ADD or ALTER COLUMN clause in a single ALTER TABLE statement (SQLSTATE 42711).
- A column length cannot be altered if the table has any summary tables that are dependent on the table (SQLSTATE 42997).

### Notes

- ADD column clauses are processed prior to all other clauses. Other clauses are processed in the order that they are specified.
- Any columns added via ALTER TABLE will not automatically be added to any existing view of the table.
- When an index is automatically created for a unique or primary key constraint, the database manager will try to use the specified constraint name as the index name with a schema name that matches the schema name of the table. If this matches an existing index name or no name for the constraint was specified, the index is created in the SYSIBM schema with a system-generated name formed of "SQL" followed by a sequence of 15 numeric characters generated by a timestamp based function.
- Any table that may be involved in a DELETE operation on table T is said to be *delete-connected* to T. Thus, a table is delete-connected to T if it is a dependent of T or it is a dependent of a table in which deletes from T cascade.
- A package has an insert (update/delete) usage on table T if records are inserted into (updated in/deleted from) T either directly by a statement in the package, or indirectly through constraints or triggers executed by the package on behalf of one of its statements. Similarly, a package has an update usage on a column if the column is modified directly by a statement in the package, or indirectly through constraints or triggers executed by the package on behalf of one of its statements.
- Any changes to primary key, unique keys, or foreign keys may have the following effect on packages, indexes, and other foreign keys.
  - If a primary key or unique key is added:



## ALTER TABLE

- There is no effect on packages, foreign keys, or existing unique keys.<sup>49</sup>
- If a primary key or unique key is dropped:
  - The index is dropped if it was automatically created for the constraint. Any packages dependent on the index are invalidated.
  - The index is set back to non-unique if it was converted to unique for the constraint and it is no longer system-required. Any packages dependent on the index are invalidated.
  - The index is set to no longer system required if it was an existing unique index used for the constraint. There is no effect on packages.
  - All dependent foreign keys are dropped. Further action is taken for each dependent foreign key, as specified in the next item.
- If a foreign key is added or dropped:
  - All packages with an insert usage on the object table are invalidated.
  - All packages with an update usage on at least one column in the foreign key are invalidated.
  - All packages with a delete usage on the parent table are invalidated.
  - All packages with an update usage on at least one column in the parent key are invalidated.
- Adding a column to a table will result in invalidation of all packages with insert usage on the altered table.
- Adding a check or referential constraint to a table that already exists and that is not in check pending state (see “SET CONSTRAINTS” on page 709) will cause the existing rows in the table to be immediately evaluated against the constraint. If the verification fails, an error (SQLSTATE 23512) is raised. If a table is in check pending state, adding a check or referential constraint will not immediately lead to the enforcement of the constraint. Instead, the corresponding constraint type flags used in the check pending operation will be updated. To begin enforcing the constraint, the SET CONSTRAINTS statement will need to be issued.
- Adding or dropping a check constraint will result in invalidation of all packages with either an insert usage on the object table or an update usage on at least one of the columns involved in the constraint.
- Adding a partitioning key will result in invalidation of all packages with an update usage on at least one of the columns of the partitioning key.
- A partitioning key that was defined by default as the first column of the primary key is not affected by dropping the primary key and adding a different primary key.

---

<sup>49</sup> If the primary or unique key uses an existing unique index that was created in a previous version and has not been converted to support deferred uniqueness, then the index is converted and packages with update usage on the associated table are invalidated.

## ALTER TABLE

- Altering a column to increase the length will invalidate all packages that reference the table (directly or indirectly through a referential constraint or trigger) with the altered column.
- Altering a column to increase the length will regenerate views (except typed views) that are dependent on the table. If an error occurs while regenerating a view, an error is returned (SQLSTATE 56098). Any typed views that are dependent on the table are marked inoperative.
- Altering a column to increase the length may cause errors in processing triggers when a statement that would involve the trigger is prepared or bound. This may occur when row length based on the sum of the lengths of the transition variables and transition table columns is too long. This error is reported using SQLCODE -670, SQLSTATE 54010. If such a trigger were dropped a subsequent attempt to create it would result in error SQLCODE -1424, SQLSTATE 54040.
- Changing the LOCKSIZE for a table will result in invalidation of all packages that have a dependency on the altered table. Further information about locking can be found in the *Administration Guide*.

## Examples

*Example 1:* Add a new column named RATING, which is one character long, to the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT  
ADD RATING CHAR(1)
```

*Example 2:* Add a new column named SITE\_NOTES to the PROJECT table. Create SITE\_NOTES as a varying-length column with a maximum length of 1000 characters. The values of the column do not have an associated character set and therefore should not be translated.

```
ALTER TABLE PROJECT  
ADD SITE_NOTES VARCHAR(1000) FOR BIT DATA
```

*Example 3:* Assume a table called EQUIPMENT exists defined with the following columns:

<b>Column Name</b>	<b>Data Type</b>
EQUIP_NO	INT
EQUIP_DESC	VARCHAR(50)
LOCATION	VARCHAR(50)
EQUIP_OWNER	CHAR(3)

Add a referential constraint to the EQUIPMENT table so that the owner (EQUIP\_OWNER) must be a department number (DEPTNO) that is present in the DEPARTMENT table. DEPTNO is the primary key of the DEPARTMENT table. If a department is removed from the DEPARTMENT table, the owner (EQUIP\_OWNER) values for all equipment owned by that department should become unassigned (or set to null). Give the constraint the name DEPTQUIP.

## ALTER TABLE

```
ALTER TABLE EQUIPMENT
ADD CONSTRAINT DEPTQUIP
FOREIGN KEY (EQUIP_OWNER)
REFERENCES DEPARTMENT
ON DELETE SET NULL
```

Also, an additional column is needed to allow the recording of the quantity associated with this equipment record. Unless otherwise specified, the EQUIP\_QTY column should have a value of 1 and must never be null.

```
ALTER TABLE EQUIPMENT
ADD COLUMN EQUIP_QTY
SMALLINT NOT NULL DEFAULT 1
```

*Example 4:* Alter table EMPLOYEE. Add the check constraint named REVENUE defined so that each employee must make a total of salary and commission greater than \$30,000.

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT REVENUE
CHECK (SALARY + COMM > 30000)
```

*Example 5:* Alter table EMPLOYEE. Drop the constraint REVENUE which was previously defined.

```
ALTER TABLE EMPLOYEE
DROP CONSTRAINT REVENUE
```

*Example 6:* Alter a table to log SQL changes in the default format.

```
ALTER TABLE SALARY1
DATA CAPTURE NONE
```

*Example 7:* Alter a table to log SQL changes in an expanded format.

```
ALTER TABLE SALARY2
DATA CAPTURE CHANGES
```

*Example 8:* Alter the EMPLOYEE table to add 4 new columns with default values.

```
ALTER TABLE EMPLOYEE
ADD COLUMN HEIGHT MEASURE DEFAULT MEASURE(1)
ADD COLUMN BIRTHDAY BIRTHDATE DEFAULT DATE('01-01-1850')
ADD COLUMN FLAGS BLOB(1M) DEFAULT BLOB(X'01')
ADD COLUMN PHOTO PICTURE DEFAULT BLOB(X'00')
```

The default values use various function names when specifying the default. Since MEASURE is a distinct type based on INTEGER, the MEASURE function is used. The HEIGHT column default could have been specified without the function since the source type of MEASURE is not BLOB or a datetime data type. Since BIRTHDATE is a distinct type based on DATE, the DATE function is used (BIRTHDATE cannot be used here). For the FLAGS and PHOTO columns the default is specified using the BLOB function even though PHOTO is a distinct type. To specify a default for BIRTHDAY, FLAGS and PHOTO columns, a function must be used because the type is a BLOB or a distinct type sourced on a BLOB or datetime data type.

## ALTER TABLE

*Example 9:* Assume that you have a table called CUSTOMERS that is defined with the following columns:

Column Name	Data Type
BRANCH_NO	SMALLINT
CUSTOMER_NO	DECIMAL(7)
CUSTOMER_NAME	VARCHAR(50)

In this table, the primary key is made up of the BRANCH\_NO and CUSTOMER\_NO columns. You want to partition the table, so you need to create a partitioning key for the table. The table must be defined in a table space on a single-node nodegroup. The primary key must be a superset of the partitioning columns: at least one of the columns of the primary key must be used as the partitioning key. Assume that you want to make BRANCH\_NO the partitioning key. You would do this with the following statement:

```
ALTER TABLE CUSTOMERS  
  ADD PARTITIONING KEY (BRANCH_NO)
```

---

### ALTER TABLESPACE

The ALTER TABLESPACE statement is used to modify an existing table space in the following ways.

- Add a container to a DMS table space (that is, one created with the MANAGED BY DATABASE option).
- Add a container to a SMS table space on a partition (or node) that currently has no containers.
- Modify the PREFETCHSIZE setting for a table space.
- Modify the BUFFERPOOL used for tables in the table space.
- Modify the OVERHEAD setting for a table space.
- Modify the TRANSFERRATE setting for a table space.

#### Invocation

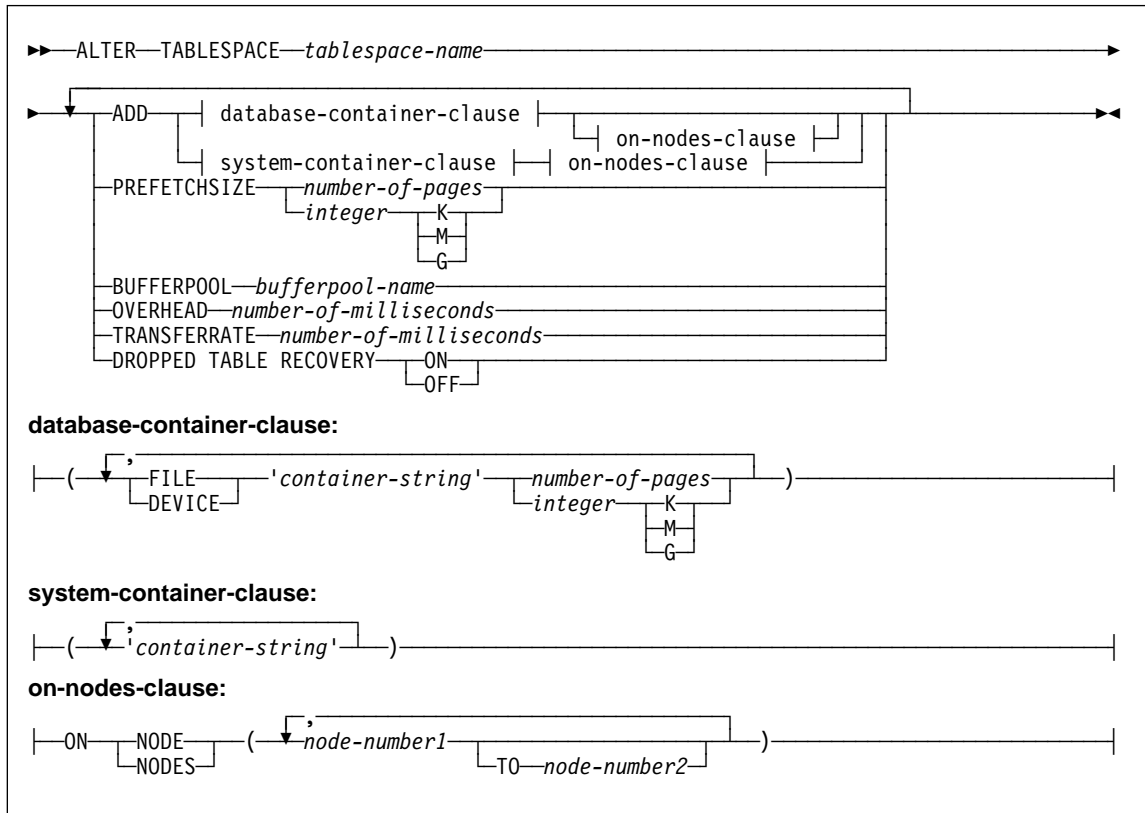
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

#### Authorization

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

#### Syntax

# ALTER TABLESPACE



## Description

*tablespace-name*

Names the table space. This is a one-part name. It is a long SQL identifier (either ordinary or delimited).

### ADD

ADD specifies that a new container is to be added to the table space.

*database-container-clause*

Adds one or more containers to a DMS table space. The table space must identify a DMS table space that already exists at the application server. See the description of *container-clause* on page 562.

*system-container-clause*

Adds one or more containers to an SMS table space on the specified partitions or nodes. The table space must identify an SMS table space that already exists at the application server. There must not be any containers on the specified partitions for the table space. (SQLSTATE 42921). See the description of *system-containers* on page 561.

## ALTER TABLESPACE

### *on-nodes-clause*

Specifies the partition or partitions for the added containers. See the description of *on-nodes-clause* on page 563.

### **PREFETCHSIZE** *number-of-pages*

Specifies the number of PAGESIZE pages that will be read from the table space when data prefetching is being performed. The prefetch size value can also be specified as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages value for prefetch size. Prefetching reads in data needed by a query prior to it being referenced by the query, so that the query need not wait for I/O to be performed.

### **BUFFERPOOL** *bufferpool-name*

The name of the buffer pool used for tables in this table space. The buffer pool must currently exist in the database (SQLSTATE 42704). The nodegroup of the table space must be defined for the bufferpool (SQLSTATE 42735).

### **OVERHEAD** *number-of-milliseconds*

Any numeric literal (integer, decimal, or floating point) that specifies the I/O controller overhead and disk seek and latency time, in milliseconds. The number should be an average for all containers that belong to the table space, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

### **TRANSFERRATE** *number-of-milliseconds*

Any numeric literal (integer, decimal, or floating point) that specifies the time to read one page (4K or 8K) into memory, in milliseconds. The number should be an average for all containers that belong to the table space, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

### **DROPPED TABLE RECOVERY**

Dropped tables in the specified table space may be recovered using the RECOVER DROPPED TABLE ON option of the ROLLFORWARD command.

## Notes

- Guidance on choosing optimal values for the PREFETCHSIZE, OVERHEAD, and TRANSFERRATE parameters, and information on rebalancing is provided in the *Administration Guide*.
- Once the new container has been added and the transaction is committed, the contents of the table space are automatically rebalanced across the containers. Access to the table space is not restricted during the rebalancing.
- If adding more than one container to a table space, it is recommended that they be added in the same statement so that the cost of rebalancing is incurred only once. An attempt to add containers to the same table space in separate ALTER TABLESPACE statements within a single transaction will result in an error (SQLSTATE 55041).

## ALTER TABLESPACE

- In a partitioned database if more than one partition resides on the same physical node, then the same device or specific path cannot be specified for such partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each partition or use a relative path name.
- Although the table space definition is transactional and the changes to the table space definition are reflected in the catalog tables on commit, the buffer pool with the new definition cannot be used until the next time the database is started. The buffer pool in use, when the ALTER TABLESPACE statement was issued, will continue to be used in the interim.

### Examples

*Example 1:* Add a device to the PAYROLL table space.

```
ALTER TABLESPACE PAYROLL
  ADD (DEVICE '/dev/rhdisk9' 10000)
```

*Example 2:* Change the prefetch size and I/O overhead for the ACCOUNTING table space.

```
ALTER TABLESPACE ACCOUNTING
  PREFETCHSIZE 64
  OVERHEAD 19.3
```



**ALTER TYPE (Structured)**

The ALTER TYPE statement is used to add or drop attributes of a user-defined structured type.

**Invocation**

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

**Authorization**

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the type.
- definer of the type as recorded in the DEFINER column of SYSCAT.DATATYPES

**Syntax**

```

▶▶ ALTER TYPE type-name {
    ADD ATTRIBUTE attribute-definition |
    DROP ATTRIBUTE attribute-name
}
  
```

**Description***type-name*

Identifies the structured type to be changed. It must be an existing type defined in the catalog (SQLSTATE 42704) and the type must be a structured type (SQLSTATE 428DP). The unqualified name must not be the same as the name of a built-in data type or BOOLEAN (SQLSTATE 42918). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

**ADD ATTRIBUTE**

Adds an attribute after the last attribute of the existing structured type.

*attribute-definition*

Defines the characteristics of the new attribute. See *attribute-definition* in "CREATE TYPE (Structured)" on page 578.

**DROP ATTRIBUTE**

Drops an attribute of the existing structured type.

*attribute-name*

The name of the attribute. The attribute must exist as an attribute of the type (SQLSTATE 42703).

## ALTER TYPE (Structured)

### Rules

- A type cannot be altered if it or one of its subtypes is the type of an existing table (SQLSTATE 55043).

### Notes

- When a type is altered, all packages are invalidated that depend on functions that use this type or a subtype of this type as a parameter or a result.

### Examples

*Example 1:* The ALTER TYPE statement can be used to permit a cycle of mutually referencing types and tables. Consider mutually referencing tables named EMPLOYEE and DEPARTMENT.

The following sequence would allow the types and tables to be created.

```
CREATE TYPE DEPT ...
CREATE TYPE EMP ... (including attribute named DEPTREF of type REF(DEPT))
ALTER TYPE DEPT ADD ATTRIBUTE MANAGER REF(EMP)
CREATE TABLE DEPARTMENT OF DEPT ...
CREATE TABLE EMPLOYEE OF EMP (DEPTREF WITH OPTIONS SCOPE DEPARTMENT)
ALTER TABLE DEPARTMENT ALTER COLUMN MANAGER ADD SCOPE EMPLOYEE
```

The following sequence would allow these tables and types to be dropped.

```
DROP TABLE EMPLOYEE (the MANAGER column in DEPARTMENT becomes unscoped)
DROP TABLE DEPARTMENT
ALTER TYPE DEPT DROP ATTRIBUTE MANAGER
DROP TYPE EMP
DROP TYPE DEPT
```

*Example 2:* The ALTER TYPE statement can be used to create a type with an attribute that references a subtype.

```
CREATE TYPE EMP ...
CREATE TYPE MGR UNDER EMP ...
ALTER TYPE EMP ADD ATTRIBUTE MANAGER REF(MGR)
```

## ALTER VIEW

The ALTER VIEW statement modifies an existing view by altering a reference type column to add a scope.

### Invocation

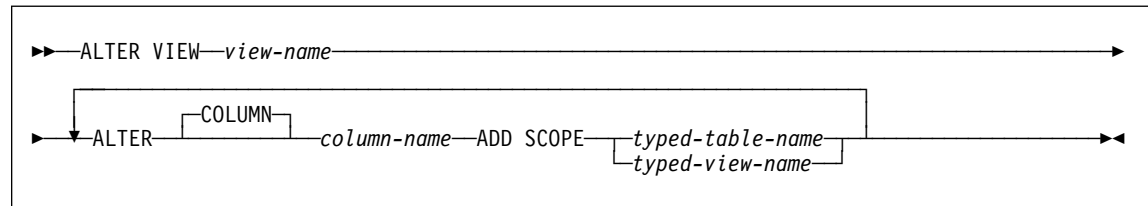
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the view
- Definer of the view to be altered
- CONTROL privilege on the view to be altered.

### Syntax



### Description

*view-name*

Identifies the view to be changed. It must be a view described in the catalog.

**ALTER COLUMN** *column-name*

Is the name of the column to be altered in the view. The *column-name* must identify an existing column of the view (SQLSTATE 42703). The name cannot be qualified.

**ADD SCOPE**

Add a scope to an existing reference type column that does not already have a scope defined (SQLSTATE 428DK). The column must not be inherited from a superview (SQLSTATE 428DJ).

*typed-table-name*

The name of a typed table. The data type of *column-name* must be REF(S), where S is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

## ALTER VIEW

| *typed-view-name*

| The name of a typed view. The data type of *column-name* must be REF(*S*),  
| where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is  
| done of any existing values in *column-name* to ensure that the values actually  
| reference existing rows in *typed-view-name*.

---

### 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. It must not be specified in REXX.

#### Authorization

None required.

#### Syntax



The diagram shows the syntax for the BEGIN DECLARE SECTION statement. It consists of a horizontal line with a double arrowhead pointing right at the end. The text "BEGIN DECLARE SECTION" is positioned on the line, starting with a double arrowhead pointing right at the beginning. This indicates that the statement is followed by a single-line comment.

#### Description

The BEGIN DECLARE SECTION statement may 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 (see “END DECLARE SECTION” on page 624).

#### Rules

- The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.
- SQL statements cannot be included within the declare section.
- Variables referenced in SQL statements must be declared in a declare section in all host languages other than REXX. Furthermore, the section must appear before the first reference to the variable. Generally, host variables are not declared in REXX with the exception of LOB locators and file reference variables. In this case, they are not declared within a BEGIN DECLARE SECTION.
- Variables declared outside a declare section must not have the same name as variables declared within a declare section.
- LOB data types must have their data type and length preceded with the SQL TYPE IS keywords.

#### Examples

*Example 1:* Define the host variables hv\_smint (smallint), hv\_vchar24 (varchar(24)), hv\_double (double), and hv\_blob\_50k (blob(51200)) in a C program.

## BEGIN DECLARE SECTION

```
EXEC SQL BEGIN DECLARE SECTION;
static short                               hv_smint;
static struct {
    short hv_vchar24_len;
    char  hv_vchar24_value[24];
}                                           hv_vchar24;
static double                              hv_double;
static SQL TYPE IS BLOB(50K)             hv_blob_50k;
EXEC SQL END DECLARE SECTION;
```

*Example 2:* Define the host variables HV-SMINT (smallint), HV-VCHAR24 (varchar(24)), HV-DEC72 (dec(7,2)), and HV-BLOB-50k (blob(51200)) in a COBOL program.

```
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 HV-SMINT          PIC S9(4)      COMP-4.
01 HV-VCHAR24.
   49 HV-VCHAR24-LENGTH PIC S9(4)      COMP-4.
   49 HV-VCHAR24-VALUE  PIC X(24).
01 HV-DEC72         PIC S9(5)V9(2)  COMP-3.
01 HV-BLOB-50K     USAGE SQL TYPE IS BLOB(50K).
    EXEC SQL END DECLARE SECTION END-EXEC.
```

*Example 3:* Define the host variables HVSMINT (smallint), HVVCHAR24 (char(24)), HVDOUBLE (double), and HVBLOB50k (blob(51200)) in a Fortran program.

```
EXEC SQL BEGIN DECLARE SECTION
INTEGER*2      HVSMINT
CHARACTER*24   HVVCHAR24
REAL*8        HVDOUBLE
SQL TYPE IS BLOB(50K) HVBLOB50K
EXEC SQL END DECLARE SECTION
```

**Note:** In Fortran, if the expected value is greater than 254 characters, then a CLOB host variable should be used.

*Example 4:* Define the host variables HVSMINT (smallint), HVBLOB50K (blob(51200)), and HVCLOBLOC (a CLOB locator) in a REXX program.

```
DECLARE :HVCLOBLOC LANGUAGE TYPE CLOB LOCATOR
call sqlexec 'FETCH c1 INTO :HVSMINT, :HVBLOB50K'
```

Note that the variables HVSMINT and HVBLOB50K were implicitly defined by using them in the FETCH statement.

---

## CALL

Invokes a procedure stored at the location of a database. A stored procedure, for example, executes at the location of the database, and returns data to the client application.

Programs using the SQL CALL statement are designed to run in two parts, one on the client and the other on the server. The server procedure at the database runs within the same transaction as the client application. If the client application and stored procedure are on the same partition, the stored procedure is executed locally.

### Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. However, the procedure name may be specified via a host variable and this, coupled with the use of the USING DESCRIPTOR clause, allows both the procedure name and the parameter list to be provided at run time; thus achieving the same effect as a dynamically prepared statement.

### Authorization

The authorization rules vary according to the server at which the procedure is stored.

DB2 Universal Database:

The privileges held by the authorization ID of the CALL statement **at run time** statement must include at least one of the following:

- EXECUTE privilege for the package associated with the stored procedure
- CONTROL privilege for the package associated with the stored procedure
- SYSADM or DBADM authority

DB2 for OS/390:

The privileges held by the authorization ID of the CALL statement **at bind time** must include at least one of the following:

- EXECUTE privilege for the package associated with the stored procedure
- Ownership of the package associated with the stored procedure
- PACKADM authority for the package's collection
- SYSADM authority

DB2 for AS/400:

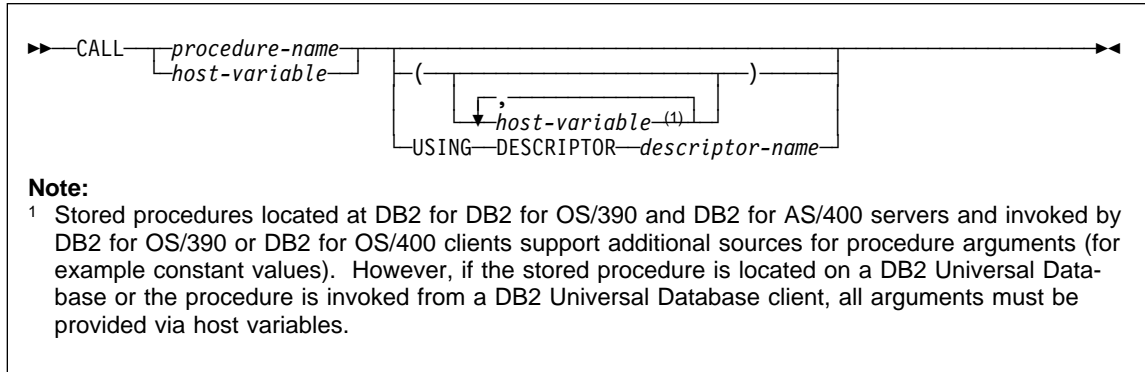
The privileges held by the authorization ID of the CALL statement **at bind time** must include at least one of the following:

- If the stored procedure is written in REXX:
  - The system authorities \*OBJOPR and \*READ on the source file associated with the procedure
  - The system authority \*EXECUTE on the library containing the source file and the system authority \*USE to the CL command
- If the stored procedure is not written in REXX:
  - The system authority \*EXECUTE on both the program associated with the procedure and on the library containing that program

# CALL

- Administrative authority

## Syntax



## Description

*procedure-name* or *host-variable*

Identifies the procedure to call. The procedure name may be specified either directly or within a host variable. The procedure identified must exist at the current server (SQLSTATE 42724).

If *procedure-name* is specified it must be an ordinary identifier not greater than 254 bytes. Since this can only be an ordinary identifier, it cannot contain blanks or special characters and the value is converted to upper case. Thus, if it is necessary to use lower case names, blanks or special characters, the name must be specified via a *host-variable*.

If *host-variable* is specified, it must be a character-string variable with a length attribute that is not greater than 254 bytes, and it must not include an indicator variable. Note that the value is **not** converted to upper case. *procedure-name* must be left-justified.

The procedure name can take one of several forms. The forms supported vary according to the server at which the procedure is stored.

DB2 Universal Database:

*procedure-name* The name (with no extension) of the procedure to execute.

The procedure invoked is determined as follows.

1. The *procedure-name* is used both as the name of the stored procedure library and the function name within that library. For example, if *procedure-name* is `proclib`, the DB2 server will load the stored procedure library named `proclib` and execute the function routine `proclib()` within that library.

In UNIX-based systems, the DB2 server finds the stored procedure library in the default directory `sqllib/function`.



## CALL

Unfenced stored procedures are in the `sqllib/function/unfenced` directory.

In OS/2, the location of the stored procedures is specified by the LIBPATH variable in the CONFIG.SYS file. Unfenced stored procedures are in the `sqllib\dll\unfenced` directory.

2. If the library or function could not be found, the *procedure-name* is used to search the defined procedures (in SYSCAT.PROCEDURES) for a matching procedure. A matching procedure is determined using the steps that follow.
  - a. Find the procedures from the catalog (SYSCAT.PROCEDURES) where the PROCNAME matches the *procedure-name* specified and the PROCSHEMA is a schema name in the SQL path (CURRENT PATH special register). If the schema name is explicitly specified, the SQL path is ignored and only procedures with the specified schema name are considered.
  - b. Next, eliminate any of these procedures that do not have the same number of parameters as the number of arguments specified in the CALL statement.
  - c. Choose the remaining procedure that is earliest in the SQL path.
  - d. If there are no remaining procedures after step 2, an error is returned (SQLSTATE 42884).

Once the procedure is selected, DB2 will invoke the procedure defined by the external name.

*procedure-library!function-name* The exclamation character (!) , acts as a delimiter between the library name and the function name of the stored procedure. For example, if `proclib!func` was specified, then `proclib` would be loaded into memory and the function `func` from that library would be executed. This allows multiple functions to be placed in the same stored procedure library.

The stored procedure library is located in the directories or specified in the LIBPATH variable, as described in *procedure-name*.

*absolute-path!function-name* The *absolute-path* specifies the complete path to the stored procedure library.

In a UNIX-based system, for example, if `/u/terry/proclib!func` was specified, then the stored procedure library `proclib` would be obtained from the directory `/u/terry` and the function `func` from that library would be executed.

## CALL

In OS/2, if `d:\terry\proclib!func` was specified, then it would cause the database manager to load the `func.dll` file from the `d:\terry\proclib` directory.

In all these cases, the total length of the procedure name including its implicit or explicit full path must not be longer than 254 bytes.

DB2 for OS/390 (V4.1 or later) server:

An implicit or explicit three part name. The parts are as follows.

high order: The location name of the server where the procedure is stored.

middle: SYSPROC

middle: Some value in the PROCEDURE column of the SYSIBM.SYSPROCEDURES catalog table.

DB2 for OS/400 (V3.1 or later) server:

The external program name is assumed to be the same as the *procedure-name*.

For portability, *procedure-name* should be specified as a single token no larger than 8 bytes.

*(host-variable,...)*

Each specification of *host-variable* is a parameter of the CALL. The nth parameter of the CALL corresponds to the nth parameter of the server's stored procedure.

Each *host-variable* is assumed to be used for exchanging data in both directions between client and server. In order to avoid sending unnecessary data between client and server, the client application should provide an indicator variable with each parameter and set the indicator to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the indicator variable to -128 for any parameter that is not used to return data to the client application.

If the server is DB2 Universal Database the parameters must have matching data types in both the client and server program.<sup>50</sup>

### **USING DESCRIPTOR** *descriptor-name*

Identifies an SQLDA that must contain a valid description of host variables. The nth SQLVAR element corresponds to the nth parameter of the server's stored procedure.

Before the CALL statement is processed, the application must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA

<sup>50</sup> DB2 for OS/390 and DB2 for OS/400 servers support conversions between compatible data types when invoking their stored procedures. For example, if the client program uses the INTEGER data type and the stored procedure expects FLOAT, the server will convert the INTEGER value to FLOAT before invoking the procedure.

- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables. The following fields of each Base SQLVAR element passed must be initialized:
  - SQLTYPE
  - SQLLEN
  - SQLDATA
  - SQLIND

The following fields of each Secondary SQLVAR element passed must be initialized:

- LEN.SQLLONGLEN
- SQLDATALEN
- SQLDATATYPE\_NAME

Each SQLDA is assumed to be used for exchanging data in both directions between client and server. In order to avoid sending unnecessary data between client and server, the client application should set the SQLIND field to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the SQLIND field -128 for any parameter that is not used to return data to the client application.

## Notes

- **Use of Large Object (LOB) data types:**

If the client and server application needs to specify LOB data from an SQLDA, allocate double the number of SQLVAR entries.

LOB data types are supported by stored procedures starting with DB2 Version 2. The LOB data types are not supported by all down level clients or servers.

- **Returning Result Sets from Stored Procedures:**

If the client application program is written using CLI, result sets can be returned directly to the client application. The stored procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on the result set, and leaving the cursor open when exiting the procedure.

At the end of a procedure that is invoked via CLI:

- For every cursor that has been left open, a result set is returned to the application.
- If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened.
- Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the stored procedure at the time the stored procedure is terminated, then rows 151 through 500 will be returned to the stored procedure.

## CALL

For additional information refer to the *Embedded SQL Programming Guide* and the *CLI Guide and Reference*.

- **Inter-operability between the CALL statement and the DARI API:**

In general, the CALL statement will not work with existing DARI procedures. See the *Embedded SQL Programming Guide* for details.

## Examples

### Example 1:

In C, invoke a procedure called TEAMWINS in the ACHIEVE library passing it a parameter stored in the host variable HV\_ARGUMENT.

```
strcpy(HV_PROCNAME, "ACHIEVE!TEAMWINS");  
CALL :HV_PROCNAME (:HV_ARGUMENT);
```

### Example 2:

In C, invoke a procedure called :SALARY\_PROC using the SQLDA named INOUT\_SQLDA.

```
struct sqlda *INOUT_SQLDA;  
  
/* Setup code for SQLDA variables goes here */  
  
CALL :SALARY_PROC  
USING DESCRIPTOR :*INOUT_SQLDA;
```

### Example 3:

A Java stored procedure is defined in the database using the following statement:

```
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,  
                                OUT COST     DECIMAL(7,2),  
                                OUT QUANTITY INTEGER)  
    EXTERNAL NAME 'parts!onhand'  
    LANGUAGE JAVA PARAMETER STYLE DB2GENERAL;
```

A Java application calls this stored procedure using the following code fragment:

## CALL

```
...
CallableStatement stpCall ;

String sql = "CALL PARTS_ON_HAND (?,?,?)" ;

stpCall = con.prepareCall( sql ) ; /* con is the connection */

stpCall.setInt( 1, variable1 ) ;
stpCall.setBigDecimal( 2, variable2 ) ;
stpCall.setInt( 3, variable3 ) ;

stpCall.registerOutParameter( 2, Types.DECIMAL, 2 ) ;
stpCall.registerOutParameter( 3, Types.INTEGER ) ;

stpCall.execute() ;

variable2 = stpCall.getBigDecimal(2) ;
variable3 = stpCall.getInt(3) ;
...
```

This application code fragment will invoke the Java method *onhand* in class *parts* since the procedure-name specified on the CALL statement is found in the database and has the external name 'parts!onhand'.

# CLOSE

---

## CLOSE

The CLOSE statement closes a cursor. If a result table was created when the cursor was opened, that table is destroyed.

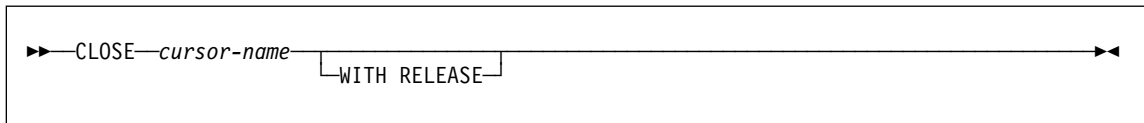
### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared.

### Authorization

None required. See “DECLARE CURSOR” on page 595 for the authorization required to use a cursor.

### Syntax



### Description

*cursor-name*

Identifies the cursor to be closed. The *cursor-name* must identify a declared cursor as explained in the DECLARE CURSOR statement. When the CLOSE statement is executed, the cursor must be in the open state.

#### **WITH RELEASE**

When specified then all read locks (if any) that have been held for the cursor are released.

### Notes

- At the end of a unit of work, all cursors that belong to an application process and that were declared without the WITH HOLD option are implicitly closed.
- CLOSE does not cause a commit or rollback operation.
- The WITH RELEASE clause has no effect for cursors that are operating under isolation levels CS or UR. When specified for cursors that are operating under isolation levels RS or RR, WITH RELEASE terminates some of the guarantees of those isolation levels. Specifically, if the cursor is opened again, an RS cursor may experience the 'nonrepeatable read' phenomenon and an RR cursor may experience either the 'nonrepeatable read' or 'phantom' phenomenon. Refer to Appendix H, “Comparison of Isolation Levels” on page 885 for more details.

If a cursor that was originally either RR or RS is reopened after being closed using the WITH RELEASE clause, then new read locks will be acquired.

## CLOSE

- Special rules apply to cursors within a stored procedure that have not been closed before returning to the calling program. See “Notes” on page 413 for more information.

### Example

A cursor is used to fetch one row at a time into the C 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 TDEPT
  WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
  .
  .
}

EXEC SQL CLOSE C1;
```

## COMMENT ON

---

### COMMENT ON

The COMMENT ON statement adds or replaces comments in the catalog descriptions of various objects.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges that must be held by the authorization ID of the COMMENT ON statement must include one of the following:

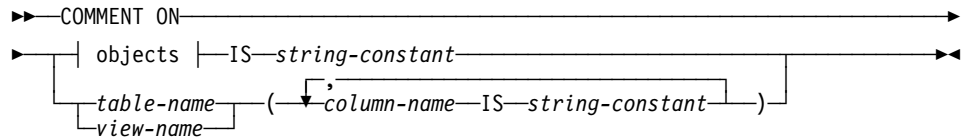
- SYSADM or DBADM
- definer of the object (underlying table for column or constraint) as recorded in the DEFINER column of the catalog view for the object (OWNER column for a schema)
- ALTERIN privilege on the schema (applicable only to objects allowing more than one-part names)
- CONTROL privilege on the object (applicable to index, package, table and view objects only)
- ALTER privilege on the object (applicable to table objects only)

Note that for table space or nodegroup the authorization ID must have SYSADM or SYSCTRL authority.

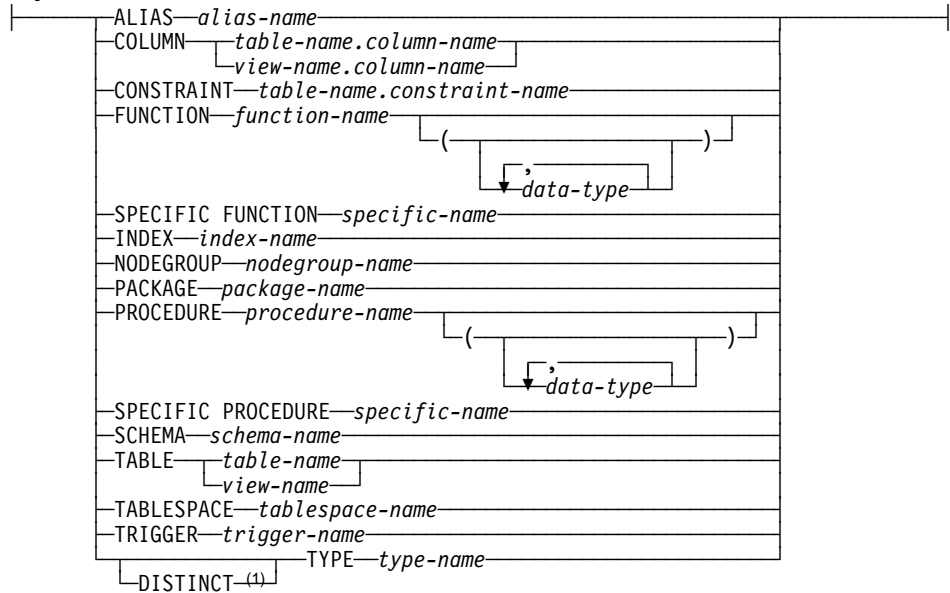
#### Syntax



## COMMENT ON



### objects:



### Note:

- 1 The keyword DATA can be used as a synonym for DISTINCT.

## Description

### ALIAS *alias-name*

Indicates a comment will be added or replaced for an alias. The *alias-name* must identify an alias that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the alias.

### COLUMN *table-name.column-name* or *view-name.column-name*

Indicates a comment will be added or replaced for a column. The *table-name.column-name* or *view-name.column-name* combination must identify a column and table combination that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.COLUMNS catalog view for the row that describes the column.

A comment cannot be made on a column of an inoperative view. (SQLSTATE 51024).

## COMMENT ON

### **CONSTRAINT** *table-name.constraint-name*

Indicates a comment will be added or replaced for a constraint. The *table-name.constraint-name* combination must identify a constraint and the table that it constrains; they must be described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.TABCONST catalog view for the row that describes the constraint.

### **FUNCTION**

Indicates a comment will be added or replaced for a function. The function instance specified must be a user-defined function described in the catalog.

There are several different ways available to identify the function instance:

#### **FUNCTION** *function-name*

Identifies the particular function, and is valid only if there is exactly one function with the *function-name*. The function thus identified may have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the function in the named or implied schema, an error (SQLSTATE 42854) is raised.

#### **FUNCTION** *function-name (data-type,...)*

Provides the function signature, which uniquely identifies the function to be commented upon. The function selection algorithm is *not* used.

##### *function-name*

Gives the function name of the function to be commented upon. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

##### *(data-type,...)*

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 is used to identify the specific function for which to add or replace the comment.

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(*n*) does not need to match the defined value for *n* since  $0 < n < 25$  means REAL and  $24 < n < 54$  means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

(Note that the FOR BIT DATA attribute is not considered part of the signature for matching purposes. So, for example, a CHAR FOR BIT DATA specified in the signature would match a function defined with CHAR only, and vice versa.)

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

#### **SPECIFIC FUNCTION** *specific-name*

Indicates that comments will be added or replaced for a function (see FUNCTION for other methods of identifying a function). Identifies the particular user-defined function that is to be commented upon, using the specific name either specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

It is not possible to comment on a function that is either in the SYSIBM schema or the SYSFUN schema (SQLSTATE 42832).

The comment replaces the value of the REMARKS column of the SYSCAT.FUNCTIONS catalog view for the row that describes the function.

#### **INDEX** *index-name*

Indicates a comment will be added or replaced for an index. The *index-name* must identify a distinct index that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.INDEXES catalog view for the row that describes the index.

#### **NODEGROUP** *nodegroup-name*

Indicates a comment will be added or replaced for a nodegroup. The *nodegroup-name* must identify a distinct nodegroup that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.NODEGROUPS catalog view for the row that describes the nodegroup.

#### **PACKAGE** *package-name*

Indicates a comment will be added or replaced for a package. The *package-name* must identify a distinct package that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.PACKAGES catalog view for the row that describes the package.

## COMMENT ON

### PROCEDURE

Indicates a comment will be added or replaced for a procedure. The procedure instance specified must be a stored procedure described in the catalog.

There are several different ways available to identify the procedure instance:

#### **PROCEDURE** *procedure-name*

Identifies the particular procedure, and is valid only if there is exactly one procedure with the *procedure-name* in the schema. The procedure thus identified may have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the procedure in the named or implied schema, an error (SQLSTATE 42854) is raised.

#### **PROCEDURE** *procedure-name (data-type,...)*

This is used to provide the procedure signature, which uniquely identifies the procedure to be commented upon.

##### *procedure-name*

Gives the procedure name of the procedure to be commented upon. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

##### *(data-type,...)*

Must match the data types that were specified on the CREATE PROCEDURE statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure for which to add or replace the comment.

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE PROCEDURE statement.

A type of FLOAT(n) does not need to match the defined value for n since 0<n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

**SPECIFIC PROCEDURE** *specific-name*

Indicates that comments will be added or replaced for a procedure (see PROCEDURE for other methods of identifying a procedure). Identifies the particular stored procedure that is to be commented upon, using the specific name either specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

The comment replaces the value of the REMARKS column of the SYSCAT.PROCEDURES catalog view for the row that describes the procedure.

**SCHEMA** *schema-name*

Indicates a comment will be added or replaced for a schema. The *schema-name* must identify a schema that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.SCHEMATA catalog view for the row that describes the schema.

**TABLE** *table-name or view-name*

Indicates a comment will be added or replaced for a table or view. The *table-name* or *view-name* must identify a distinct table or view (not an alias) that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the table or view.

**TABLESPACE** *tablespace-name*

Indicates a comment will be added or replaced for a table space. The *tablespace-name* must identify a distinct table space that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TABLESPACES catalog view for the row that describes the table space.

**TRIGGER** *trigger-name*

Indicates a comment will be added or replaced for a trigger. The *trigger-name* must identify a distinct trigger that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TRIGGERS catalog view for the row that describes the trigger.

**TYPE** *type-name*

Indicates a comment will be added or replaced for a user-defined type. The *type-name* must identify a user-defined type that is described in the catalog (SQLSTATE 42704). If DISTINCT is specified, *type-name* must identify a distinct type that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.DATATYPES catalog view for the row that describes the user-defined type.

## COMMENT ON

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

### **IS** *string-constant*

Specifies the comment to be added or replaced. The *string-constant* can be any character string constant of up to 254 bytes. (Carriage return and line feed each count as 1 byte.)

### *table-name|view-name ( { column-name IS string-constant } ... )*

This form of the COMMENT ON statement provides the ability to specify comments for multiple columns of a table or view. The column names must not be qualified, each name must identify a column of the specified base table or view, and the table or view must be described in the catalog.

A comment cannot be made on a column of an inoperative view (SQLSTATE 51024).

## Examples

*Example 1:* Add a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE
IS 'Reflects first quarter reorganization'
```

*Example 2:* Add a comment for the EMP\_VIEW1 view.

```
COMMENT ON TABLE EMP_VIEW1
IS 'View of the EMPLOYEE table without salary information'
```

*Example 3:* Add a comment for the EDLEVEL column of the EMPLOYEE table.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL
IS 'highest grade level passed in school'
```

*Example 4:* Add comments for two different columns of the EMPLOYEE table.

```
COMMENT ON EMPLOYEE
(WORKDEPT IS 'see DEPARTMENT table for names',
EDLEVEL IS 'highest grade level passed in school' )
```

*Example 5:* Pellow wants to comment on the CENTRE function, which he created in his PELLOW schema, using the signature to identify the specific function to be commented on.

```
COMMENT ON FUNCTION CENTRE (INT,FLOAT)
IS 'Frank''s CENTRE fctn, uses Chebychev method'
```

*Example 6:* McBride wants to comment on another CENTRE function, which she created in the PELLOW schema, using the specific name to identify the function instance to be commented on:

```
COMMENT ON SPECIFIC FUNCTION PELLOW.FOCUS92 IS
'Louise''s most triumphant CENTRE function, uses the
Brownian fuzzy-focus technique'
```

## COMMENT ON

*Example 7:* Comment on the function ATOMIC\_WEIGHT in the CHEM schema, where it is known that there is only one function with that name:

```
COMMENT ON FUNCTION CHEM.ATOMIC_WEIGHT  
IS 'takes atomic nbr, gives atomic weight'
```

*Example 8:* Eigler wants to comment on the SEARCH procedure, which he created in his EIGLER schema, using the signature to identify the specific procedure to be commented on.

```
COMMENT ON PROCEDURE SEARCH (CHAR,INT)  
IS 'Frank''s mass search and replace algorithm'
```

*Example 9:* Macdonald wants to comment on another SEARCH function, which he created in the EIGLER schema, using the specific name to identify the procedure instance to be commented on:

```
COMMENT ON SPECIFIC PROCEDURE EIGLER.DESTROY IS  
'Patrick''s mass search and destroy algorithm'
```

*Example 10:* Comment on the procedure OSMOSIS in the BIOLOGY schema, where it is known that there is only one procedure with that name:

```
COMMENT ON PROCEDURE BIOLOGY.OSMOSIS  
IS 'Calculations modelling osmosis'
```

# COMMIT

---

## COMMIT

The COMMIT statement terminates a unit of work and commits the database changes that were made by that unit of work.

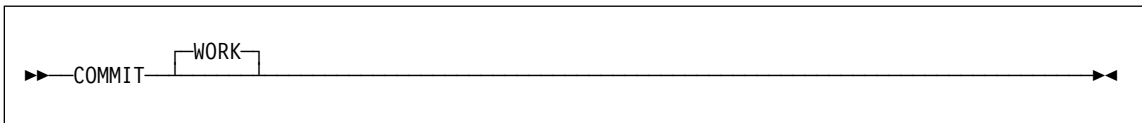
### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

None required.

### Syntax



### Description

The unit of work in which the COMMIT statement is executed is terminated and a new unit of work is initiated. All changes made by the following statements executed during the unit of work are committed: ALTER, COMMENT ON, CREATE, DELETE, DROP, GRANT, INSERT, LOCK TABLE, REVOKE, SET CONSTRAINTS, SET transition-variable, and UPDATE.

The following statements, however, are not under transaction control and changes made by them are independent of issuing the COMMIT statement:

- SET CONNECTION,
- SET CURRENT DEGREE,
- SET CURRENT EXPLAIN MODE,
- SET CURRENT EXPLAIN SNAPSHOT,
- SET CURRENT PACKAGESET,
- SET CURRENT QUERY OPTIMIZATION,
- SET CURRENT REFRESH AGE,
- SET EVENT MONITOR STATE,
- SET PATH,
- SET SCHEMA.

All locks acquired by the unit of work subsequent to its initiation are released, except necessary locks for open cursors that are declared WITH HOLD. All open cursors not defined WITH HOLD are closed. Open cursors defined WITH HOLD remain open, and



## COMMIT

the cursor is positioned before the next logical row of the result table.<sup>51</sup> All LOB locators are freed. Note that this is true even when the locators are associated with LOB values retrieved via a cursor that has the WITH HOLD property.

### Notes

It is strongly recommended that each application process explicitly ends its unit of work before terminating. If the application program ends normally without a COMMIT or ROLLBACK statement then the database manager attempts a commit or rollback depending on the application environment. Refer to *Embedded SQL Programming Guide* for implicitly ending a transaction in different application environments.

See the “Notes” on page 627 for impact of COMMIT on cached dynamic SQL statements.

### Example

Commit alterations to the database made since the last commit point.

**COMMIT WORK**

---

<sup>51</sup> A FETCH must be performed before a Positioned UPDATE or DELETE statement is issued.

## Compound SQL

---

### Compound SQL

Combines one or more other SQL statements (*sub-statements*) into an executable block.

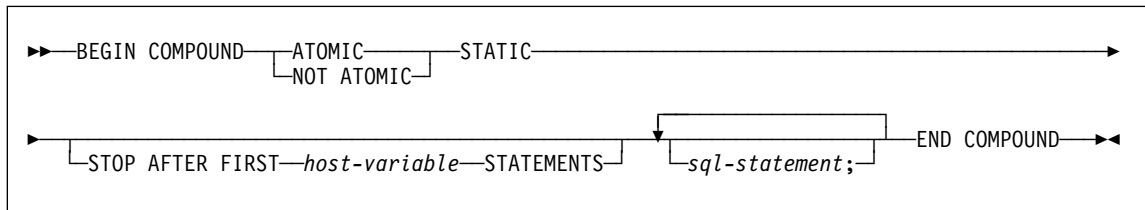
### Invocation

This statement can only be embedded in an application program. The entire Compound SQL statement construct is an executable statement that cannot be dynamically prepared. The statement is not supported in REXX.

### Authorization

None for the Compound SQL statement itself. The authorization ID of the Compound SQL statement must have the appropriate authorization on all the individual statements that are contained within the Compound SQL statement.

### Syntax



### Description

#### ATOMIC

Specifies that, if any of the sub-statements within the Compound SQL statement fails, then all changes made to the database by any of the sub-statements, including changes made by successful sub-statements, are undone.

#### NOT ATOMIC

Specifies that, regardless of the failure of any sub-statements, the Compound SQL statement will not undo any changes made to the database by the other sub-statements.

#### STATIC

Specifies that input variables for all sub-statements retain their original value. For example, if

```
SELECT ... INTO :abc ...
```

is followed by:

```
UPDATE T1 SET C1 = 5 WHERE C2 = :abc
```

the UPDATE statement will use the value that :abc had at the start of the execution of the Compound SQL statement, not the value that follows the SELECT INTO.

## Compound SQL

If the same variable is set by more than one sub-statement, the value of that variable following the Compound SQL statement is the value set by the last sub-statement.

**Note:** Non-static behavior is not supported. This means that the sub-statements should be viewed as executing non-sequentially and sub-statements should not have interdependencies.

### STOP AFTER FIRST

Specifies that only a certain number of sub-statements will be executed.

#### *host-variable*

A small integer that specifies the number of sub-statements to be executed.

### STATEMENTS

Completes the STOP AFTER FIRST *host-variable* clause.

#### *sql-statement*

All executable statements except the following can be contained within a Compound SQL statement:

CALL	FETCH
CLOSE	OPEN
CONNECT	PREPARE
Compound SQL	RELEASE
DESCRIBE	ROLLBACK
DISCONNECT	SET CONNECTION
EXECUTE IMMEDIATE	

If a COMMIT statement is included, it must be the last sub-statement. If COMMIT is in this position (for example the last of a hundred statements), it will be issued even if the STOP AFTER FIRST *host-variable* STATEMENT option indicates that only the first fifty statements are to be executed. In this case, it will be treated as the fifty-first sub-statement.

An error will be returned if COMMIT is included when using CONNECT TYPE 2 or running in an XA distributed transaction processing environment (SQLSTATE 25000).

## Rules

- No host language code is allowed within a Compound SQL statement; that is, no host language code is allowed between the sub-statements that make up the Compound SQL statement.
- Only NOT ATOMIC Compound SQL statements will be accepted by DDCS.
- Compound SQL statements cannot be nested.

## Notes

One SQLCA is returned for the entire Compound SQL statement. Most of the information in that SQLCA reflects the values set by the application server when it processed the last sub-statement. For instance:

## Compound SQL

- The SQLCODE and SQLSTATE are normally those for the last sub-statement (the exception is described in the next point).
- If a 'no data found' warning (SQLSTATE '02000') is returned, then that warning is given precedence over any other warning in order that a WHENEVER NOT FOUND exception can be acted upon.<sup>52</sup>
- The SQLWARN indicators are an accumulation of the indicators set for all sub-statements.

If one or more errors occurred during NOT ATOMIC Compound SQL execution and none of these are of a serious nature, the SQLERRMC will contain information on up to a maximum of seven of these errors. The first token of the SQLERRMC will indicate the total number of errors that occurred. The remaining tokens will each contain the ordinal position and the SQLSTATE of the failing sub-statement within the Compound SQL statement. The format is a character string of the form:

**nnnXssscccc**

with the substring starting with X repeating up to six more times and the string elements defined as follows.

- |             |                                                                                                                                                                                       |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>nnn</b>  | The total number of statements that produced errors. <sup>53</sup> This field is left-justified and padded with blanks.                                                               |
| <b>X</b>    | The token separator X'FF'.                                                                                                                                                            |
| <b>sss</b>  | The ordinal position of the statement that caused the error. <sup>53</sup> For example, if the first statement failed, this field would contain the number one left-justified ('1 '). |
| <b>cccc</b> | The SQLSTATE of the error.                                                                                                                                                            |

The second SQLERRD field contains the number of statements that failed (returned negative SQLCODES).

The third SQLERRD field in the SQLCA is an accumulation of the number of rows affected by all sub-statements.

The fourth SQLERRD field in the SQLCA is a count of the number of successful sub-statements. If, for example, the third sub-statement in a Compound SQL statement failed, the fourth SQLERRD field would be set to 2, indicating that 2 sub-statements were successfully processed before the error was encountered.

The fifth SQLERRD field in the SQLCA is an accumulation of the number of rows updated or deleted due to the enforcement of referential integrity constraints for all sub-statements that triggered such constraint activity.

---

<sup>52</sup> This means that the SQLCODE, SQLERRML, SQLERRMC, and SQLERRP fields in the SQLCA that is eventually returned to the application are those from the sub-statement that triggered the 'no data found'. If there is more than one 'no data found' warning within the Compound SQL statement, the fields for the last sub-statement will be the fields returned.

<sup>53</sup> If the number would exceed 999, counting restarts at zero.

## Examples

*Example 1:* In a C program, issue a Compound SQL statement that updates both the ACCOUNTS and TELLERS tables. If there is an error in any of the statements, undo the effect of all statements (ATOMIC). If there are no errors, commit the current unit of work.

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
  UPDATE ACCOUNTS SET ABALANCE = ABALANCE + :delta
    WHERE AID = :aid;
  UPDATE TELLERS SET TBALANCE = TBALANCE + :delta
    WHERE TID = :tid;
  INSERT INTO TELLERS (TID, BID, TBALANCE) VALUES (:i, :branch_id, 0);
  COMMIT;
END COMPOUND;
```

*Example 2:* In a C program, insert 10 rows of data into the database. Assume the host variable :nbr contains the value 10 and S1 is a prepared INSERT statement. Further, assume that all the inserts should be attempted regardless of errors (NOT ATOMIC).

```
EXEC SQL BEGIN COMPOUND NOT ATOMIC STATIC STOP AFTER FIRST
:nbr STATEMENTS
  EXECUTE S1 USING DESCRIPTOR :*sqlda0;
  EXECUTE S1 USING DESCRIPTOR :*sqlda1;
  EXECUTE S1 USING DESCRIPTOR :*sqlda2;
  EXECUTE S1 USING DESCRIPTOR :*sqlda3;
  EXECUTE S1 USING DESCRIPTOR :*sqlda4;
  EXECUTE S1 USING DESCRIPTOR :*sqlda5;
  EXECUTE S1 USING DESCRIPTOR :*sqlda6;
  EXECUTE S1 USING DESCRIPTOR :*sqlda7;
  EXECUTE S1 USING DESCRIPTOR :*sqlda8;
  EXECUTE S1 USING DESCRIPTOR :*sqlda9;
END COMPOUND;
```

## CONNECT (Type 1)

---

### CONNECT (Type 1)

The CONNECT (Type 1) statement connects an application process to the identified application server according to the rules for remote unit of work.

An application process can only be connected to one application server at a time. This is called the *current server*. A default application server may be established when the application requester is initialized. If implicit connect is available and an application process is started, it is implicitly connected to the default application server. The application process can explicitly connect to a different application server by issuing a CONNECT TO statement. A connection lasts until a CONNECT RESET statement or a DISCONNECT statement is issued or until another CONNECT TO statement changes the application server.

See “Remote Unit of Work Connection Management” on page 26 for concepts and additional details on connection states. See “Options that Govern Distributed Unit of Work Semantics” on page 33 for the precompiler options that determine the framework for CONNECT behavior.

### Invocation

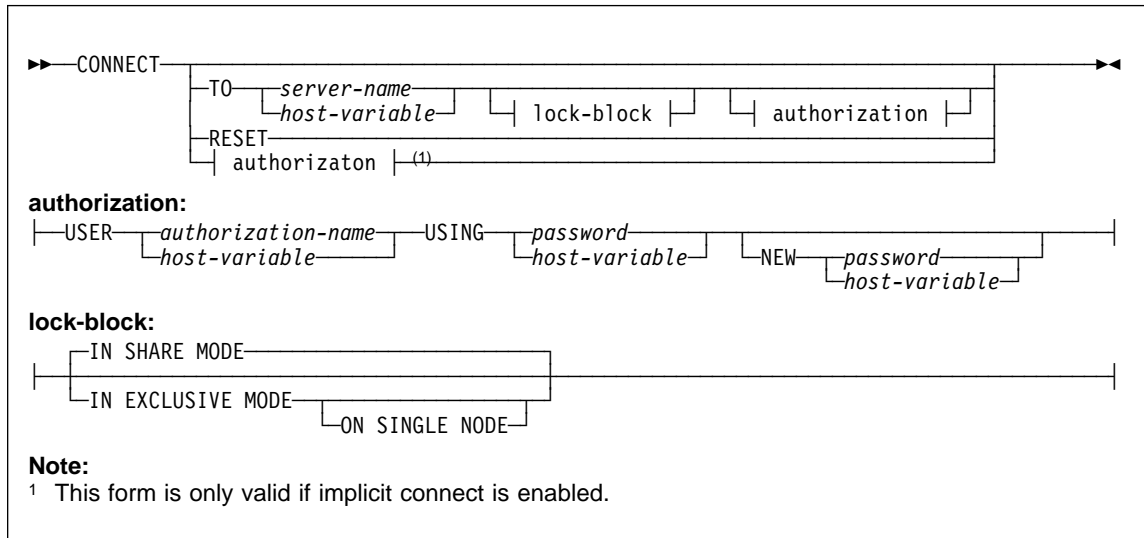
Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

The authorization ID of the statement must be authorized to connect to the identified application server. Depending on the authentication setting for the database, the authorization check may be performed by either the client or the server. For a partitioned database, the user and group definitions must be identical across partitions or nodes. Refer to the AUTHENTICATION database manager configuration parameter in the *Administration Guide* for information about the authentication setting.

### Syntax

## CONNECT (Type 1)



### Description

#### **CONNECT** (with no operand)

Returns information about the current server. The information is returned in the SQLERRP field of the SQLCA as described in "Successful Connection".

If a connection state exists, the authorization ID and database alias are placed in the SQLERRMC field of the SQLCA. If no connection exists and implicit connect is possible, then an attempt to make an implicit connection is made. If implicit connect is not available, this attempt results in an error (no existing connection). If no connection, then the SQLERRMC field is blank.

The country code and code page of the application server are placed in the SQLERRMC field (as they are with a successful CONNECT TO statement).

This form of CONNECT:

- Does not require the application process to be in the connectable state.
- If connected, does not change the connection state.
- If unconnected and implicit connect is available, a connection to the default application server is made. In this case, the country code and code page of the application server are placed in the SQLERRMC field, like a successful CONNECT TO statement.
- If unconnected and implicit connect is not available, the application process remains unconnected.
- Does not close cursors.

#### **TO** *server-name* or *host-variable*

Identifies the application server by the specified *server-name* or a *host-variable* which contains the server-name.

## CONNECT (Type 1)

If a host-variable is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The server-name that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

**Note:** DB2 for MVS supports a 16 byte location-name and both SQL/DS and DB2/400 support a 18 byte target database name. DB2 Version 5.2 only supports the use of 8 byte database-alias name on the SQL CONNECT statement. However, the database-alias name can be mapped to an 18 byte database name through the Database Connection Service Directory.

When the CONNECT TO statement is executed, the application process must be in the connectable state (see "Remote Unit of Work Connection Management" on page 26 for information about connection states with Type 1 CONNECT).

### **Successful Connection:**

If the CONNECT TO statement is successful:

- All open cursors are closed, all prepared statements are destroyed, and all locks are released from the previous application server.
- The application process is disconnected from its previous application server, if any, and connected to the identified application server.
- The actual name of the application server (not an alias) is placed in the CURRENT SERVER special register.
- Information about the application server 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* identifies the product as follows:
    - DSN for DB2 for MVS
    - ARI for SQL/DS
    - QSQ for DB2/400
    - SQL for DB2 Universal Database
  - *vv* is a two-digit version identifier such as '02'
  - *rr* is a two-digit release identifier such as '01'
  - *m* is a one-digit modification level identifier such as '0'.

For example, if the application server is Version 1 Release 1 of DB2 for OS/2, the value of SQLERRP is 'SQL01010'.<sup>54</sup>

- The SQLERRMC field of the SQLCA is set to contain the following values (separated by X'FF')

---

<sup>54</sup> This release of DB2 Universal Database Version 5.2 is 'SQL05000'.



## CONNECT (Type 1)

1. the country code of the application server (or blanks if using DDCS),
2. the code page of the application server (or CCSID if using DDCS),
3. the authorization ID,
4. the database alias,
5. the platform type of the application server. Currently identified values are:

<b>Token</b>	<b>Server</b>
QAS	DB2/400
QDB2	DB2 for MVS
QDB2/2	DB2 Universal Database for OS/2
QDB2/6000	DB2 Universal Database for AIX
QDB2/HPUX	DB2 Universal Database for HP-UX
QDB2/NT	DB2 Universal Database for NT
QDB2/SNI	DB2 Universal Database for Siemens Nixdorf
QDB2/SUN	DB2 Universal Database for Solaris
QOS/2 DBM	ES 1.0 DBM (note that there is a single space between QOS/2 and DBM)
QSQLDS/VM	SQL/DS for VM
QSQLDS/VSE	SQL/DS for VSE

6. The agent ID. It identifies the agent executing within the database manager on behalf of the application. This field is the same as the agent\_id element returned by the database monitor.
  7. The agent index. It identifies the index of the agent and is used for service.
  8. Partition number. For a non-partitioned database, this is always 0, if present.
  9. The code page of the application client.
  10. Number of partitions in a partitioned database. If the database cannot be partitioned, the value is 0 (zero). Token is present only with Version 5 or later.
- The SQLERRD(1) field of the SQLCA indicates the maximum expected difference in length of mixed character data (CHAR data types) when converted to the database code page from the application code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction.<sup>55</sup>
  - The SQLERRD(2) field of the SQLCA indicates the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1

---

<sup>55</sup> See the "Character Conversion Expansion Factor" section of the "Programming in Complex Environments" chapter in the *Embedded SQL Programming Guide* for details.

## CONNECT (Type 1)

indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. <sup>55</sup>

- The SQLERRD(3) field of the SQLCA indicates whether or not the database on the connection is updatable. A database is initially updatable, but is changed to read-only if a unit of work determines the authorization ID cannot perform updates. The value is one of:
  - 1 - updatable
  - 2 - read-only
- The SQLERRD(4) field of the SQLCA returns certain characteristics of the connection. The value is one of:
  - 0 - N/A (only possible if running from a down-level client which is one phase commit and is an updater).
  - 1 - one-phase commit.
  - 2 - one-phase commit; read-only (only applicable to connections to DRDA1 databases in TP Monitor environment).
  - 3 - two-phase commit.
- The SQLERRD(5) field of the SQLCA returns the authentication type of the connection. The value is one of:
  - 0 - Authenticated on the server.
  - 1 - Authenticated on the client.
  - 2 - Authenticated using DB2 Connect.
  - 3 - Authenticated using Distributed Computing Environment security services.
  - 255 - Authentication not specified.

See "Controlling Database Access" in the *Administration Guide* for details on authentication types.

- The SQLERRD(6) field of the SQLCA returns the partition number of the partition to which the connection was made if the database is partitioned. Otherwise, a value of 0 is returned.

### **Unsuccessful Connection:**

If the CONNECT TO statement is unsuccessful:

- The SQLERRP field of the SQLCA is set to the name of the module at the application requester that detected the error. Note that the first three characters of the module name identifies the product. For example, if the application requester is on the OS/2 database manager, the first three characters are 'SQL'.
- If the CONNECT TO statement is unsuccessful because the application process is not in the connectable state, the connection state of the application process is unchanged.
- If the CONNECT TO statement is unsuccessful because the *server-name* is not listed in the local directory, an error message (SQLSTATE 08001) is

## CONNECT (Type 1)

issued and the connection state of the application process remains unchanged:

- If the application requester was not connected to an application server then the application process remains unconnected.
- If the application requester was already connected to an application server, the application process remains connected to that application server. Any further statements are executed at that application server.
- If the CONNECT TO statement is unsuccessful for any other reason, the application process is placed into the unconnected state.

### IN SHARE MODE

Allows other concurrent connections to the database and prevents other users from connecting to the database in exclusive mode.

### IN EXCLUSIVE MODE <sup>56</sup>

Prevents concurrent application processes from executing any operations at the application server, unless they have the same authorization ID as the user holding the exclusive lock.

### ON SINGLE NODE

Specifies that the coordinator partition is connected in exclusive mode and all other partitions are connected in share mode. This option is only effective in a partitioned database .

### RESET

Disconnects the application process from the current server. A commit operation is performed. If implicit connect is available, the application process remains unconnected until an SQL statement is issued.

### USER *authorization-name/host-variable*

Identifies the userid trying to connect to the application server. If a host-variable is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The userid that is contained within the *host-variable* must be left justified and must not be delimited by quotation marks.

### USING *password/host-variable*

Identifies the password of the userid trying to connect to the application server. *Password* or *host-variable* may be up to 18 characters. If a host variable is specified, it must be a character string variable with a length attribute not greater than 18 and it must not include an indicator variable.

### NEW *password/host-variable*

Identifies the new password that should be assigned to the userid identified by the USER option. *Password* or *host-variable* may be up to 18 characters. If a host variable is specified, it must be a character string variable with a length attribute not

---

<sup>56</sup> This option is not supported by DDCS.

## CONNECT (Type 1)

greater than 18 and it must not include an indicator variable. The system on which the password will be changed depends on how user authentication is set up.

### Notes

- It is good practice for the first SQL statement executed by an application process to be the **CONNECT TO** statement.
- If a **CONNECT TO** statement is issued to the current application server with a different userid and password then the conversation is deallocated and reallocated. All cursors are closed by the database manager (with the loss of the cursor position if the **WITH HOLD** option was used).
- If a **CONNECT TO** statement is issued to the current application server with the same userid and password then the conversation is not deallocated and reallocated. Cursors, in this case, are not closed.
- To use DB2 Universal Database Extended Enterprise Edition, the user or application must connect to one of the partitions listed in the `db2nodes.cfg` file ( see “Data Partitioning Across Multiple Partitions” on page 41 for information about this file). You should try to ensure that not all users use the same partition as the coordinator partition.

### Examples

*Example 1:* In a C program, connect to the application server TOROLAB3, where TOROLAB3 is a database alias of the same name, with the userid FERMAT and the password THEOREM.

```
EXEC SQL CONNECT TO TOROLAB3 USER FERMAT USING THEOREM;
```

*Example 2:* In a C program, connect to an application server whose database alias is stored in the host variable APP\_SERVER (varchar(8)). Following a successful connection, copy the 3 character product identifier of the application server to the variable PRODUCT (char(3)).

```
EXEC SQL CONNECT TO :APP_SERVER;  
if (strncmp(SQLSTATE, '00000', 5))  
    strncpy(PRODUCT, sqlca.sqlerrp, 3);
```

---

### CONNECT (Type 2)

The CONNECT (Type 2) statement connects an application process to the identified application server and establishes the rules for application-directed distributed unit of work. This server is then the current server for the process.

See “Application-Directed Distributed Unit of Work” on page 30 for concepts and additional details.

Most aspects of a CONNECT (Type 1) statement also apply to a CONNECT (Type 2) statement. Rather than repeating that material here, this section describes only those elements of Type 2 that differ from Type 1.

#### Invocation

The invocation is the same as “Invocation” on page 432.

#### Authorization

The authorization is the same as “Authorization” on page 432.

#### Syntax

The syntax is the same as “Syntax” on page 432. The selection between Type 1 and Type 2 is determined by precompiler options. See “Options that Govern Distributed Unit of Work Semantics” on page 33 for an overview of these options. Further details are provided in the *Command Reference* and *API Reference* manuals.

#### Description

**TO** *server-name/host-variable*

The rules for coding the name of the server are the same as for Type 1.

If the SQLRULES(STD) option is in effect, the *server-name* must not identify an existing connection of the application process, otherwise an error (SQLSTATE 08002) is raised.

If the SQLRULES(DB2) option is in effect and the *server-name* identifies an existing connection of the application process, that connection is made current and the old connection is placed into the dormant state. That is, the effect of the CONNECT statement in this situation is the same as that of a SET CONNECTION statement.

See “Options that Govern Distributed Unit of Work Semantics” on page 33 for information about the specification of SQLRULES.

##### **Successful Connection**

If the CONNECT TO statement is successful:

- A connection to the application server is either created (or made non-dormant) and placed into the current and held states.
- If the CONNECT TO is directed to a different server than the current server, then the current connection is placed into the dormant state.

## CONNECT (Type 2)

- The CURRENT SERVER special register and the SQLCA are updated in the same way as for Type 1 CONNECT; see page on page 434.

### **Unsuccessful Connection**

If the CONNECT TO statement is unsuccessful:

- No matter what the reason for failure, the connection state of the application process and the states of its connections are unchanged.
- As with an unsuccessful Type 1 CONNECT, the SQLERP field of the SQLCA is set to the name of the module at the application requester or server that detected the error.

### **CONNECT (with no operand), IN SHARE/EXCLUSIVE MODE, USER, and USING**

If a connection exists, Type 2 behaves like a Type 1. The authorization ID and database alias are placed in the SQLERRMC field of the SQLCA. If a connection does not exist, no attempt to make an implicit connection is made and the SQLERP and SQLERRMC fields return a blank. (Applications can check if a current connection exists by checking these fields.)

A CONNECT with no operand that includes USER and USING can still connect an application process to a database using the DB2DBDFT environment variable. This method is equivalent to a Type 2 CONNECT RESET, but permits the use of a userid and password.

### **RESET**

Equivalent to an explicit connect to the default database if it is available. If a default database is not available, the connection state of the application process and the states of its connections are unchanged.

Availability of a default database is determined by installation options, environment variables, and authentication settings. See the *Quick Beginnings* for information on setting implicit connect on installation and environment variables, and the *Administration Guide* for information on authentication settings.

## Rules

- As outlined in “Options that Govern Distributed Unit of Work Semantics” on page 33 a set of connection options governs the semantics of connection management. Default values are assigned to every preprocessed source file. An application can consist of multiple source files precompiled with different connection options.

Unless a SET CLIENT command or API has been executed first, the connection options used when preprocessing the source file containing the first SQL statement executed at run-time become the effective connection options.

If a CONNECT statement from a source file preprocessed with different connection options is subsequently executed without the execution of any intervening SET CLIENT command or API, an error (SQLSTATE 08001) is raised. Note that once a SET CLIENT command or API has been executed, the connection options used when preprocessing all source files in the application are ignored.

Example 1 on page 443 illustrates these rules.

## CONNECT (Type 2)

- Although the CONNECT TO statement can be used to establish or switch connections, CONNECT TO with the USER/USING clause will only be accepted when there is no current or dormant connection to the named server. The connection must be released before issuing a connection to the same server with the USER/USING clause, otherwise it will be rejected (SQLSTATE 51022). Release the connection by issuing a DISCONNECT statement or a RELEASE statement followed by a COMMIT statement.

### Notes

- Implicit connect is supported for the first SQL statement in an application with Type 2 connections. In order to execute SQL statements on the default database, first the CONNECT RESET or the CONNECT USER/USING statement must be used to establish the connection. The CONNECT statement with no operands will display information about the current connection if there is one, but will not connect to the default database if there is no current connection.

### Comparing Type 1 and Type 2 CONNECT Statements:

The semantics of the CONNECT statement are determined by the CONNECT precompiler option or the SET CLIENT API (see “Options that Govern Distributed Unit of Work Semantics” on page 33). CONNECT Type 1 or CONNECT Type 2 can be specified and the CONNECT statements in those programs are known as Type 1 and Type 2 CONNECT statements respectively. Their semantics are described below:

#### Use of **CONNECT TO**:

Type 1	Type 2
Each unit of work can only establish connection to one application server.	Each unit of work can establish connection to multiple application servers.
The current unit of work must be committed or rolled back before allowing a connection to another application server.	The current unit of work need not be committed or rolled back before connecting to another application server.
The CONNECT statement establishes the current connection. Subsequent SQL requests are forwarded to this connection until changed by another CONNECT.	Same as Type 1 CONNECT if establishing the first connection. If switching to a dormant connection and SQLRULES is set to STD, then the SET CONNECTION statement must be used instead.
Connecting to the current connection is valid and does not change the current connection.	Same as Type 1 CONNECT if the SQLRULES precompiler option is set to DB2. If SQLRULES is set to STD, then the SET CONNECTION statement must be used instead.

## CONNECT (Type 2)

---

### Type 1

Connecting to another application server disconnects the current connection. The new connection becomes the current connection. Only one connection is maintained in a unit of work.

---

SET CONNECTION statement is supported for Type 1 connections, but the only valid target is the current connection.

### Use of **CONNECT...USER...USING**:

---

#### Type 1

Connecting with the USER...USING clauses disconnects the current connection and establishes a new connection with the given authorization name and password.

### Use of **Implicit CONNECT, CONNECT RESET, and Disconnecting**:

---

#### Type 1

CONNECT RESET can be used to disconnect the current connection.

---

### Type 2

Connecting to another application server puts the current connection into the *dormant state*. The new connection becomes the current connection. Multiple connections can be maintained in a unit of work.

If the CONNECT is for an application server on a dormant connection, it becomes the current connection.

Connecting to a dormant connection using CONNECT is only allowed if SQLRULES(DB2) was specified. If SQLRULES(STD) was specified, then the SET CONNECTION statement must be used instead.

---

SET CONNECTION statement is supported for Type 2 connections to change the state of a connection from dormant to current.

---

#### Type 2

Connecting with the USER/USING clause will only be accepted when there is no current or dormant connection to the same named server.

---

#### Type 2

CONNECT RESET is equivalent to connecting to the default application server explicitly if one has been defined in the system.

Connections can be disconnected by the application at a successful COMMIT. Prior to the commit, use the RELEASE statement to mark a connection as release-pending. All such connections will be disconnected at the next COMMIT.

An alternative is to use the precompiler options DISCONNECT(EXPLICIT), DISCONNECT(CONDITIONAL), DISCONNECT(AUTOMATIC), or the DISCONNECT statement instead of the RELEASE statement.



## CONNECT (Type 2)

Type 1	Type 2
After using CONNECT RESET to disconnect the current connection, if the next SQL statement is not a CONNECT statement, then it will perform an implicit connect to the default application server if one has been defined in the system.	CONNECT RESET is equivalent to an explicit connect to the default application server if one has been defined in the system.
It is an error to issue consecutive CONNECT RESETs.	It is an error to issue consecutive CONNECT RESETs ONLY if SQLRULES(STD) was specified because this option disallows the use of CONNECT to existing connection.
CONNECT RESET also implicitly commits the current unit of work.	CONNECT RESET does not commit the current unit of work.
If an existing connection is disconnected by the system for whatever reasons, then subsequent non-CONNECT SQL statements to this database will receive an SQLSTATE of 08003.	If an existing connection is disconnected by the system, COMMIT, ROLLBACK, and SET CONNECTION statements are still permitted.
The unit of work will be implicitly committed when the application process terminates successfully.	Same as Type 1.
All connections (only one) are disconnected when the application process terminates.	All connections (current, dormant, and those marked for release pending) are disconnected when the application process terminates.

### CONNECT Failures:

Type 1	Type 2
Regardless of whether there is a current connection when a CONNECT fails (with an error other than server-name not defined in the local directory), the application process is placed in the unconnected state. Subsequent non-CONNECT statements receive an SQLSTATE of 08003.	If there is a current connection when a CONNECT fails, the current connection is unaffected.  If there was no current connection when the CONNECT fails, then the program is then in an unconnected state. Subsequent non-CONNECT statements receive an SQLSTATE of 08003.

## Examples

*Example 1:* This example illustrates the use of multiple source programs (shown in the boxes), some preprocessed with different connection options (shown above the code) and one of which contains a SET CLIENT API call.

```
PGM1: CONNECT(2) SQLRULES(DB2) DISCONNECT(CONDITIONAL)
```

```
...  
exec sql CONNECT TO OTTAWA;  
exec sql SELECT col1 INTO :hv1  
FROM tb11;  
...
```

## CONNECT (Type 2)

PGM2: CONNECT(2) SQLRULES(STD) DISCONNECT(AUTOMATIC)

```
...
exec sql CONNECT TO QUEBEC;
exec sql SELECT col1 INTO :hv1
FROM tb12;
...
```

PGM3: CONNECT(2) SQLRULES(STD) DISCONNECT(EXPLICIT)

```
...
SET CLIENT CONNECT 2 SQLRULES DB2 DISCONNECT EXPLICIT 1
exec sql CONNECT TO LONDON;
exec sql SELECT col1 INTO
: hv1 FROM tb13;
...
```

1 Note: not the actual syntax of the SET CLIENT API

PGM4: CONNECT(2) SQLRULES(DB2) DISCONNECT(CONDITIONAL)

```
...
exec sql CONNECT TO REGINA;
exec sql SELECT col1 INTO
: hv1 FROM tb14;
...
```

If the application executes PGM1 then PGM2:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to QUEBEC fails with SQLSTATE 08001 because both SQLRULES and DISCONNECT are different.

If the application executes PGM1 then PGM3:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to LONDON runs: connect=2, sqlrules=DB2, disconnect=EXPLICIT

This is OK because the SET CLIENT API is run before the second CONNECT statement.

If the application executes PGM1 then PGM4:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to REGINA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL

This is OK because the preprocessor options for PGM1 are the same as those for PGM4.

*Example 2:*

This example shows the interrelationships of the CONNECT (Type 2), SET CONNECTION, RELEASE, and DISCONNECT statements. S0, S1, S2, and S3 represent four servers.

## CONNECT (Type 2)

Sequence	Statement	Current Server	Dormant Connections	Release Pending
0	No statement	None	None	None
1.	SELECT * FROM TBLA	S0 (default)	None	None
2	CONNECT TO S1 SELECT * FROM TBLB	S1 S1	S0 S0	None None
3	CONNECT TO S2 UPDATE TBLC SET ...	S2 S2	S0, S1 S0, S1	None None
4	CONNECT TO S3 SELECT * FROM TBLD	S3 S3	S0, S1, S2 S0, S1, S2	None None
5	SET CONNECTION S2	S2	S0, S1, S3	None
6	RELEASE S3	S2	S0, S1	S3
7	COMMIT	S2	S0, S1	None
8	SELECT * FROM TBLE	S2	S0, S1	None
9	DISCONNECT S1 SELECT * FROM TBLF	S2 S2	S0 S0	None None

## CREATE ALIAS

---

### CREATE ALIAS

The CREATE ALIAS statement defines an alias for a table, view, or another alias.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include as least one of the following:

- SYSADM or DBADM authority
- IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the alias does not exist
- CREATEIN privilege on the schema, if the schema name of the alias refers to an existing schema .

To use the referenced object via the alias, the same privileges are required on that object as would be necessary if the object itself were used.

#### Syntax

```
▶ CREATE ALIAS alias-name FOR table-name | view-name | alias-name2
      | SYNONYM (1)
```

**Note:**

<sup>1</sup> CREATE SYNONYM is accepted as an alternative for CREATE ALIAS for syntax toleration of existing CREATE SYNONYM statements of other SQL implementations.

#### Description

*alias-name*

Names the alias. The name must not identify a table, view, or alias that exists in the current database.

If a two-part name is specified, the schema name cannot begin with "SYS" (SQLSTATE 42939).

The rules for defining an alias name are the same as those used for defining a table name.

**FOR** *table-name*, *view-name*, or *alias-name2*

Identifies the table, view, or alias for which *alias-name* is defined. If another alias name is supplied (*alias-name2*), then it must not be the same as the new *alias-name* being defined (in its fully-qualified form).

### Notes

- The definition of the newly created alias is stored in SYSCAT.TABLES.
- An alias can be defined for an object that does not exist at the time of the definition. If it does not exist, a warning is issued (SQLSTATE 01522). However, the referenced object must exist when a SQL statement containing the alias is compiled, otherwise an error is issued (SQLSTATE 52004).
- An alias can be defined to refer to another alias as part of an alias chain but this chain is subject to the same restrictions as a single alias when used in an SQL statement. An alias chain is resolved in the same way as a single alias. If an alias used in a view definition, a statement in a package, or a trigger points to an alias chain, then a dependency is recorded for the view, package, or trigger on each alias in the chain. Repetitive cycles in an alias chain are not allowed and are detected at alias definition time.
- Creating an alias with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

### Examples

*Example 1:* HEDGES attempts to create an alias for a table T1 (both unqualified).

```
CREATE ALIAS A1 FOR T1
```

The alias HEDGES.A1 is created for HEDGES.T1.

*Example 2:* HEDGES attempts to create an alias for a table (both qualified).

```
CREATE ALIAS HEDGES.A1 FOR MCKNIGHT.T1
```

The alias HEDGES.A1 is created for MCKNIGHT.T1.

*Example 3:* HEDGES attempts to create an alias for a table (alias in a different schema; HEDGES is not a DBADM; HEDGES does not have CREATEIN on schema MCKNIGHT).

```
CREATE ALIAS MCKNIGHT.A1 FOR MCKNIGHT.T1
```

This example fails (SQLSTATE 42501).

*Example 4:* HEDGES attempts to create an alias for an undefined table (both qualified; FUZZY.WUZZY does not exist).

```
CREATE ALIAS HEDGES.A1 FOR FUZZY.WUZZY
```

This statement succeeds but with a warning (SQLSTATE 01522).

*Example 5:* HEDGES attempts to create an alias for an alias (both qualified).

```
CREATE ALIAS HEDGES.A1 FOR MCKNIGHT.T1  
CREATE ALIAS HEDGES.A2 FOR HEDGES.A1
```

## CREATE ALIAS

The first statement succeeds (as per example 2).

The second statement succeeds and an alias chain is created, consisting of HEDGES.A2 which refers to HEDGES.A1 which refers to MCKNIGHT.T1. Note that it does not matter whether or not HEDGES has any privileges on MCKNIGHT.T1. The alias is created regardless of the table privileges.

## CREATE BUFFERPOOL

### CREATE BUFFERPOOL

The CREATE BUFFERPOOL statement creates a new buffer pool to be used by the database manager. Although the buffer pool definition is transactional and the entries will be reflected in the catalog tables on commit, the buffer pool will not become active until the next time the database is started.

In a partitioned database, a default buffer pool definition is specified for each partition or node, with the capability to override the size on specific partitions or nodes. Also, in a partitioned database, the buffer pool is defined on all partitions unless nodegroups are specified. If nodegroups are specified, the buffer pool will only be created on partitions that are in those nodegroups.

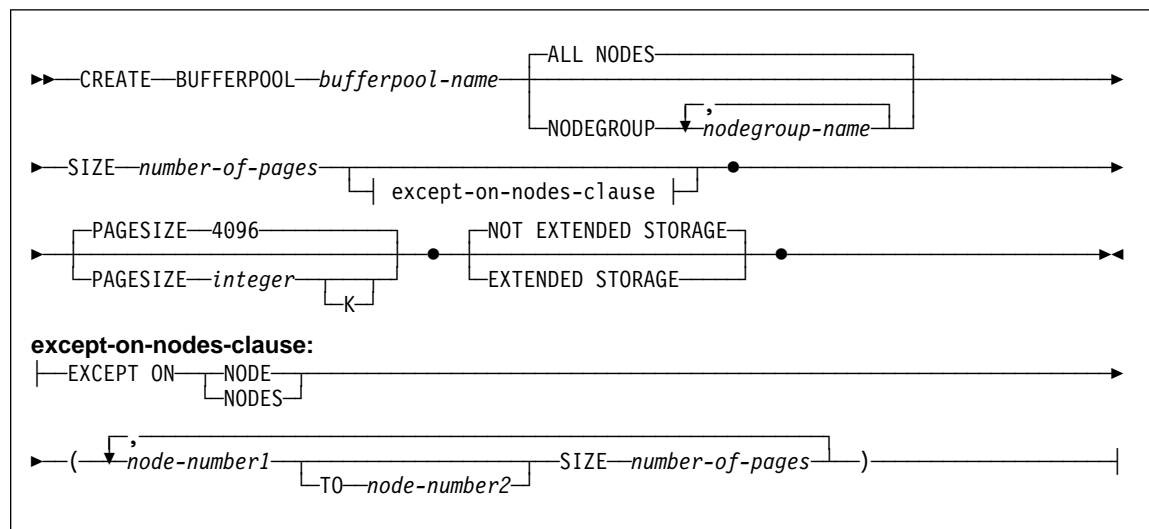
### Invocation

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 ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax



### Description

*bufferpool-name*

Names the buffer pool. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *bufferpool-name* must not identify a buffer pool that already exists in a catalog (SQLSTATE 42710). The *bufferpool-name* must not begin with the characters "SYS" or "IBM" (SQLSTATE 42939).

## CREATE BUFFERPOOL

### ALL NODES

This buffer pool will be created on all partitions in the database.

### NODEGROUP *nodegroup-name, ...*

Identifies the nodegroup or nodegroups to which the buffer pool definition is applicable. If this is specified, this buffer pool will only be created on partitions in these nodegroups. Each nodegroup must currently exist in the database (SQLSTATE 42704). If the NODEGROUP keyword is not specified, then this buffer pool will be created on all partitions (and any partitions subsequently added to the database).

### SIZE *number-of-pages*

The size of the buffer pool specified as the number of pages.<sup>57</sup> In a partitioned database, this will be the default size for all partitions where the buffer pool exists.

### *except-on-nodes-clause*

Specifies the partition or partitions for which the size of the buffer pool will be different than the default. If this clause is not specified, then all partitions will have the same size as specified for this buffer pool.

### EXCEPT ON NODES

Keywords that indicate that specific partitions are specified. NODE is a synonym for NODES.

#### *node-number1*

Specifies a specific partition number that is included in the partitions for which the buffer pool is created.

#### TO *node-number2*

Specify a range of partition numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers must be included in the partitions for which the buffer pool is created (SQLSTATE 42729).

### SIZE *number-of-pages*

The size of the buffer pool specified as the number of pages.

### PAGESIZE *integer [K]*

Defines the size of pages used for the bufferpool. The valid values for *integer* without the suffix K are 4096 or 8192. The valid values for *integer* with the suffix K are 4 or 8. An error occurs if the page size is not one of these values (SQLSTATE 428DE). The default is 4096 byte (4K) pages. Any number of spaces is allowed between *integer* and K, including no space.

### EXTENDED STORAGE

---

<sup>57</sup> The size can be specified with a value of (-1) which will indicate that the buffer pool size should be taken from the BUFFPAGE database configuration parameter.



## CREATE BUFFERPOOL

If the extended storage configuration is turned on,<sup>58</sup> pages that are being migrated out of this buffer pool will be cached in the extended storage.

### NOT EXTENDED STORAGE

Even if the database extended storage configuration is turned on, pages that are being migrated out of this buffer pool, will NOT be cached in the extended storage.

### Notes

- Until the next time the database is started, any table space that is created will use an already active buffer pool of the same page size. The database has to be restarted for the table space assignment to the new buffer pool to take effect.
- There should be enough real memory on the machine for the total of all the buffer pools, as well as for the rest of the database manager and application requirements. If DB2 is unable to obtain the total memory for all buffer pools, it will attempt to start up only the default buffer pool. If this is unsuccessful, it will start up a minimal default buffer pool. In either of these cases, a warning will be returned to the user (SQLSTATE 01626) and the pages from all table spaces will use the default buffer pool.

---

<sup>58</sup> Extended storage configuration is turned on by setting the database configuration parameters NUM\_ESTORE\_SEGS and ESTORE\_SEG\_SIZE to non-zero values. See *Administration Guide* for details.

## CREATE DISTINCT TYPE

---

### CREATE DISTINCT TYPE

The CREATE DISTINCT TYPE statement defines a distinct type. The distinct type is always sourced on one of the built-in data types. Successful execution of the statement also generates functions to cast between the distinct type and its source type and, optionally, generates support for the comparison operators (=, <>, <, <=, >, and >=) for use with the distinct type.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include as least one of the following:

- SYSADM or DBADM authority
- IMPLICIT\_SCHEMA authority on the database, if the schema name of the distinct type does not refer to an existing schema.
- CREATEIN privilege on the schema, if the schema name of the distinct type refers to an existing schema .

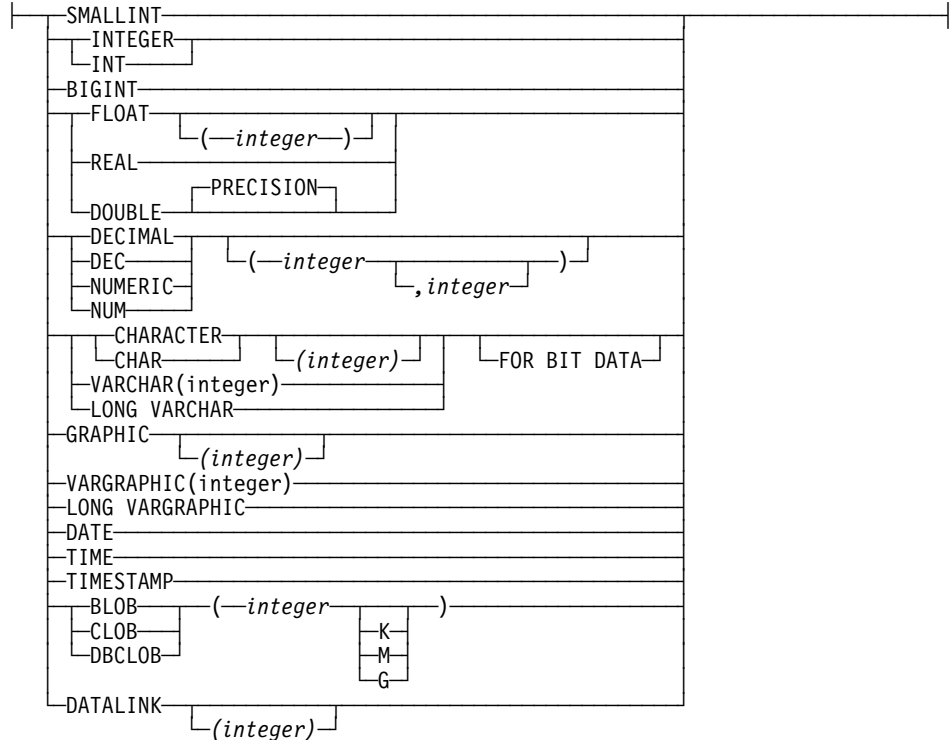
#### Syntax

## CREATE DISTINCT TYPE

► CREATE DISTINCT TYPE *distinct-type-name* AS | source-data-type | ►

► WITH COMPARISONS <sup>(1)</sup> ►

### source-data-type:



### Note:

<sup>1</sup> Required for all source-data-types except LOBs, LONG VARCHAR, LONG VARGRAPHIC and DATALINK which are not supported.

## Description

### *distinct-type-name*

Names the distinct type. The name, including the implicit or explicit qualifier must not identify a distinct type described in the catalog. The unqualified name must not be the same as the name of a source-data-type or BOOLEAN (SQLSTATE 42918).

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a schema-name followed by a period and an SQL identifier.

A number of names used as keywords in predicates are reserved for system use, and may not be used as a *distinct-type-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS,

## CREATE DISTINCT TYPE

SIMILAR, MATCH and the comparison operators as described in “Basic Predicate” on page 136 . Failure to observe this rule will lead to an error (SQLSTATE 42939).

If a two-part *distinct-type-name* is specified, the schema name cannot begin with "SYS"; otherwise, an error (SQLSTATE 42939) is raised.

### source-data-type

Specifies the data type used as the basis for the internal representation of the distinct type. For information about the association of distinct types with other data types, see “Distinct Types” on page 64. For information about data types, see “CREATE TABLE” on page 522.

### WITH COMPARISONS

Specifies that system-generated comparison operators are to be created for comparing two instances of a distinct type. These keywords should not be specified if the source-data-type is BLOB, CLOB, DBCLOB, LONG VARCHAR, LONG VARGRAPHIC, or DATALINK, otherwise a warning will be returned (SQLSTATE 01596) and the comparison operators will not be generated. For all other source-data-types, the WITH COMPARISONS keywords are required.

## Notes

- Creating a distinct type with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- The following functions are generated to cast to and from the source type:
  - One function to convert from the distinct type to the source type
  - One function to convert from the source type to the distinct type
  - One function to convert from INTEGER to the distinct type if the source type is SMALLINT
  - one function to convert from VARCHAR to the distinct type if the source type is CHAR
  - one function to convert from VARGRAPHIC to the distinct type if the source type is GRAPHIC.

In general these functions will have the following format:

```
CREATE FUNCTION source-type-name (distinct-type-name)  
  RETURNS source-type-name ...
```

```
CREATE FUNCTION distinct-type-name (source-type-name)  
  RETURNS distinct-type-name ...
```

In cases in which the source type is a parameterized type, the function to convert from the distinct type to the source type will have as function name the name of the source type without the parameters (see Table 17 on page 455 for details). The type of the return value of this function will include the parameters given on the CREATE DISTINCT TYPE statement. The function to convert from the source type to the distinct type will have an input parameter whose type is the source type including its parameters. For example,

## CREATE DISTINCT TYPE

```
CREATE DISTINCT TYPE T_SHOESIZE AS CHAR(2)
WITH COMPARISONS
```

```
CREATE DISTINCT TYPE T_MILES AS DOUBLE
WITH COMPARISONS
```

will generate the following functions:

```
FUNCTION CHAR (T_SHOESIZE) RETURNS CHAR (2)
```

```
FUNCTION T_SHOESIZE (CHAR (2))
RETURNS T_SHOESIZE
```

```
FUNCTION DOUBLE (T_MILES) RETURNS DOUBLE
```

```
FUNCTION T_MILES (DOUBLE) RETURNS T_MILES
```

The schema of the generated cast functions is the same as the schema of the distinct type. No other function with this name and with the same signature may already exist in the database (SQLSTATE 42710).

The following table gives the names of the functions to convert from the distinct type to the source type and from the source type to the distinct type for all predefined data types.

Table 17 (Page 1 of 3). CAST functions on distinct types

Source Type Name	Function Name	Parameter	Return-type
CHAR	<distinct>	CHAR (n)	<distinct>
	CHAR	<distinct>	CHAR (n)
	<distinct>	VARCHAR (n)	<distinct>
VARCHAR	<distinct>	VARCHAR (n)	<distinct>
	VARCHAR	<distinct>	VARCHAR (n)
LONG VARCHAR	<distinct>	LONG VARCHAR	<distinct>
	LONG_VARCHAR	<distinct>	LONG VARCHAR
CLOB	<distinct>	CLOB (n)	<distinct>
	CLOB	<distinct>	CLOB (n)
BLOB	<distinct>	BLOB (n)	<distinct>
	BLOB	<distinct>	BLOB (n)
GRAPHIC	<distinct>	GRAPHIC (n)	<distinct>
	GRAPHIC	<distinct>	GRAPHIC (n)
	<distinct>	VARGRAPHIC (n)	<distinct>
VARGRAPHIC	<distinct>	VARGRAPHIC (n)	<distinct>
	VARGRAPHIC	<distinct>	VARGRAPHIC (n)

## CREATE DISTINCT TYPE

Table 17 (Page 2 of 3). CAST functions on distinct types

Source Type Name	Function Name	Parameter	Return-type
LONG VARCHAR	<distinct>	LONG VARCHAR	<distinct>
	LONG_VARCHAR	<distinct>	LONG VARCHAR
DBCLOB	<distinct>	DBCLOB (n)	<distinct>
	DBCLOB	<distinct>	DBCLOB (n)
SMALLINT	<distinct>	SMALLINT	<distinct>
	<distinct>	INTEGER	<distinct>
	SMALLINT	<distinct>	SMALLINT
INTEGER	<distinct>	INTEGER	<distinct>
	INTEGER	<distinct>	INTEGER
BIGINT	<distinct>	BIGINT	<distinct>
	BIGINT	<distinct>	BIGINT
DECIMAL	<distinct>	DECIMAL (p,s)	<distinct>
	DECIMAL	<distinct>	DECIMAL (p,s)
NUMERIC	<distinct>	DECIMAL (p,s)	<distinct>
	DECIMAL	<distinct>	DECIMAL (p,s)
REAL	<distinct>	REAL	<distinct>
	<distinct>	DOUBLE	<distinct>
	REAL	<distinct>	REAL
FLOAT(n) where n<=24	<distinct>	REAL	<distinct>
	<distinct>	DOUBLE	<distinct>
	REAL	<distinct>	REAL
FLOAT(n) where n>24	<distinct>	DOUBLE	<distinct>
	DOUBLE	<distinct>	DOUBLE
FLOAT	<distinct>	DOUBLE	<distinct>
	DOUBLE	<distinct>	DOUBLE
DOUBLE	<distinct>	DOUBLE	<distinct>
	DOUBLE	<distinct>	DOUBLE
DOUBLE PRECISION	<distinct>	DOUBLE	<distinct>
	DOUBLE	<distinct>	DOUBLE
DATE	<distinct>	DATE	<distinct>
	DATE	<distinct>	DATE
TIME	<distinct>	TIME	<distinct>
	TIME	<distinct>	TIME
TIMESTAMP	<distinct>	TIMESTAMP	<distinct>
	TIMESTAMP	<distinct>	TIMESTAMP

## CREATE DISTINCT TYPE

Table 17 (Page 3 of 3). CAST functions on distinct types

Source Type Name	Function Name	Parameter	Return-type
DATALINK	<distinct>	DATALINK	<distinct>
	DATALINK	<distinct>	DATALINK

**Note:** NUMERIC and FLOAT are not recommended when creating a user-defined type for a portable application. DECIMAL and DOUBLE should be used instead.

The functions described in the above table are the only functions that are generated automatically when distinct types are defined. Consequently, none of the built-in functions (AVG, MAX, LENGTH, etc.) are supported on distinct types until the CREATE FUNCTION statement (see “CREATE FUNCTION” on page 467) is used to register user-defined functions for the distinct type, where those user-defined functions are sourced on the appropriate built-in functions. In particular, note that it is possible to register user-defined functions that are sourced on the built-in column functions.

When a distinct type is created using the WITH COMPARISONS clause, system-generated comparison operators are created. Creation of these comparison operators will generate entries in the SYSCAT.FUNCTIONS catalog view for the new functions.

The schema name of the distinct type must be included in the SQL path (see “SET PATH” on page 731 or the FUNCPATH BIND option as described in the *Embedded SQL Programming Guide*) for successful use of these operators and cast functions in SQL statements.

### Examples

*Example 1:* Create a distinct type named SHOESIZE that is based on an INTEGER data type.

```
CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS
```

This will also result in the creation of comparison operators (=, <>, <, <=, >, >=) and cast functions INTEGER(SHOESIZE) returning INTEGER and SHOESIZE(INTEGER) returning SHOESIZE.

*Example 2:* Create a distinct type named MILES that is based on a DOUBLE data type.

```
CREATE DISTINCT TYPE MILES AS DOUBLE WITH COMPARISONS
```

This will also result in the creation of comparison operators (=, <>, <, =, >, >=) and cast functions DOUBLE(MILES) returning DOUBLE and MILES(DOUBLE) returning MILES.

## CREATE EVENT MONITOR

---

### CREATE EVENT MONITOR

The CREATE EVENT MONITOR statement defines a monitor that will record certain events that occur when using the database. The definition of each event monitor also specifies where the database should record the events.

#### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

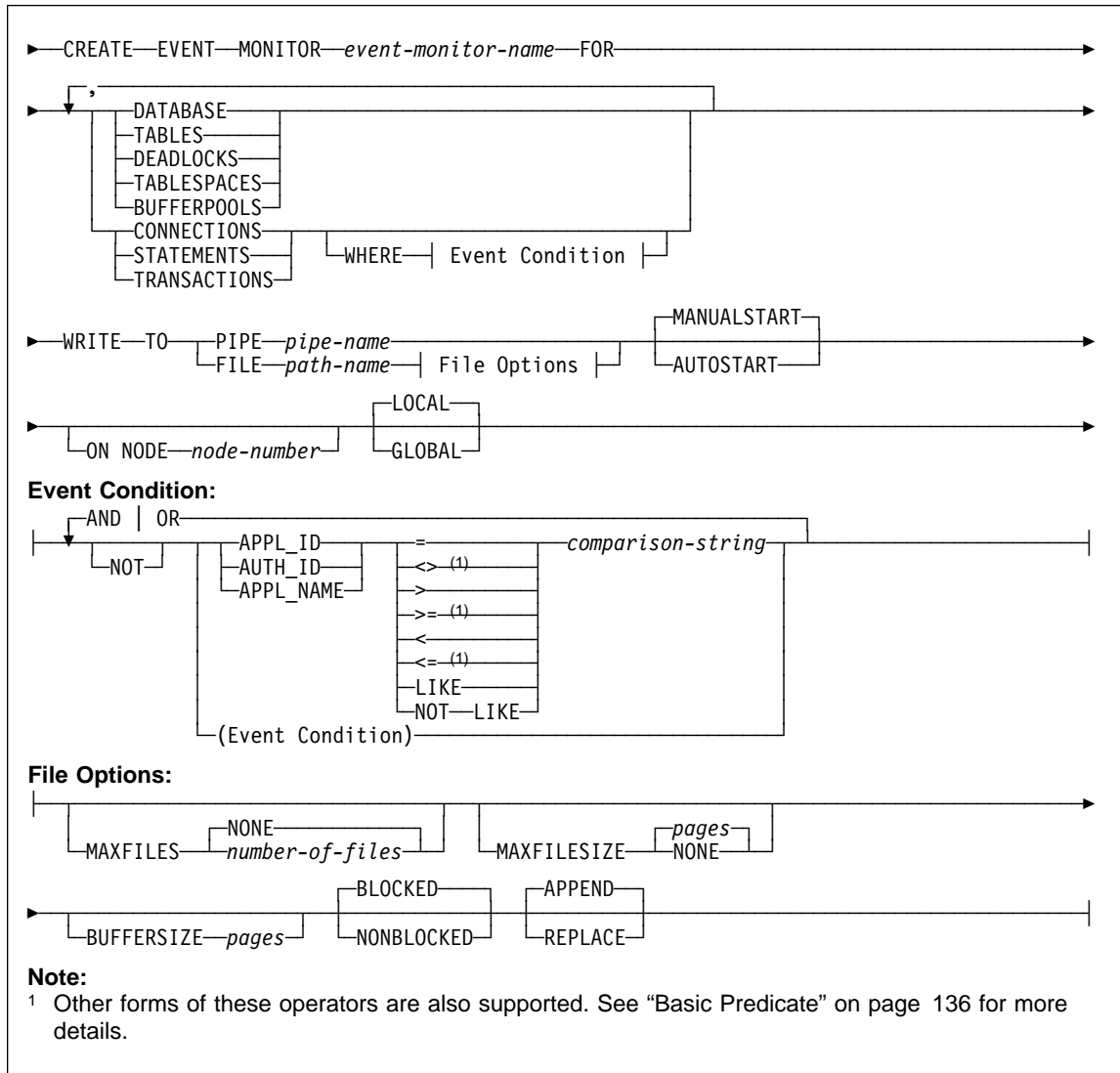
#### Authorization

The privileges held by the authorization ID must include either SYSADM or DBADM authority (SQLSTATE 42502).

#### Syntax



## CREATE EVENT MONITOR



### Description

*event-monitor-name*

Names the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

### FOR

Introduces the type of event to record.

## CREATE EVENT MONITOR

### DATABASE

Specifies that the event monitor records a database event when the last application disconnects from the database.

### TABLES

Specifies that the event monitor records a table event for each active table when the last application disconnects from the database. An active table is a table that has changed since the first connection to the database.

### DEADLOCKS

Specifies that the event monitor records a deadlock event whenever a deadlock occurs.

### TABLESPACES

Specifies that the event monitor records a table space event for each table space when the last application disconnects from the database.

### BUFFERPOOLS

Specifies that the event monitor records a buffer pool event when the last application disconnects from the database.

### CONNECTIONS

Specifies that the event monitor records a connection event when an application disconnects from the database.

### STATEMENTS

Specifies that the event monitor records a statement event whenever a SQL statement finishes executing.

### TRANSACTIONS

Specifies that the event monitor records a transaction event whenever a transaction completes (that is, whenever there is a commit or rollback operation).

### WHERE *event condition*

Defines a filter that determines which connections cause a CONNECTION, STATEMENT or TRANSACTION event to occur. If the result of the event condition is TRUE for a particular connection, then that connection will generate the requested events.

This clause is a special form of the WHERE clause that should not be confused with a standard search condition.

To determine if an application will generate events for a particular event monitor, the WHERE clause is evaluated:

1. For each active connection when an event monitor is first turned on.
2. Subsequently for each new connection to the database at connect time.

The WHERE clause is not evaluated for each event.

If no WHERE clause is specified then all events of the specified event type will be monitored.

## CREATE EVENT MONITOR

### APPL\_ID

Specifies that the application ID of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

### AUTH\_ID

Specifies that the authorization ID of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

### APPL\_NAME

Specifies that the application program name of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

The application program name is the first 20 bytes of the application program file name, after the last path separator.

### *comparison-string*

A string to be compared with the APPL\_ID, AUTH\_ID, or APPL\_NAME of each application that connects to the database. *comparison-string* must be a string constant (that is, host variables and other string expressions are not permitted).

### WRITE TO

Introduces the target for the data.

### PIPE

Specifies that the target for the event monitor data is a named pipe. The event monitor writes the data to the pipe in a single stream (that is, as if it were a single, infinitely long file). When writing the data to a pipe, an event monitor does not perform blocked writes. If there is no room in the pipe buffer, then the event monitor will discard the data. It is the monitoring application's responsibility to read the data promptly if it wishes to ensure no data loss.

### *pipe-name*

The name of the pipe (FIFO on AIX) to which the event monitor will write the data.

The naming rules for pipes are platform specific. On UNIX operating systems pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see *path-name* below). However, on OS/2, Windows 95 and Windows NT, there is a special syntax for a pipe name. As a result, on OS/2, Windows 95 and Windows NT absolute pipe names are required.

The existence of the pipe will not be checked at event monitor creation time. It is the responsibility of the monitoring application to have created and opened the pipe for reading at the time that the event monitor is acti-

## CREATE EVENT MONITOR

vated. If the pipe is not available at this time, then the event monitor will turn itself off, and will log an error. (That is, if the event monitor was activated at database start time as a result of the AUTOSTART option, then the event monitor will log an error in the system error log.) If the event monitor is activated via the SET EVENT MONITOR STATE SQL statement, then that statement will fail (SQLSTATE 58030).

### FILE

Indicates that the target for the event monitor data is a file (or set of files). The event monitor writes out the stream of data as a series of 8 character numbered files, with the extension "evt". (for example, 00000000.evt, 00000001.evt, 00000002.evt, etc). The data should be considered to be one logical file even though the data is broken up into smaller pieces (that is, the start of the data stream is the first byte in the file 00000000.evt; the end of the data stream is the last byte in the file nnnnnnnn.evt).

The maximum size of each file can be defined as well as the maximum number of files. An event monitor will never split a single event record across two files. However, an event monitor may write related records in two different files. It is the responsibility of the application that uses this data to keep track of such related information when processing the event files.

#### *path-name*

The name of the directory in which the event monitor should write the event files data. The path must be known at the server, however, the path itself could reside on another partition or node (for example, in a UNIX-based system, this might be an NFS mounted file). A string constant must be used when specifying the *path-name*.

The directory does not have to exist at CREATE EVENT MONITOR time. However, a check is made for the existence of the target path when the event monitor is activated. At that time, if the target path does not exist, an error (SQLSTATE 428A3) is raised.

If an absolute path (a path that starts with the root directory on AIX, or a disk identifier on OS/2, Windows 95 and Windows NT ) is specified, then the specified path will be the one used. If a relative path (a path that does not start with the root) is specified, then the path relative to the DB2EVENT directory in the database directory will be used.

When a relative path is specified, the DB2EVENT directory is used to convert it into an absolute path. Thereafter, no distinction is made between absolute and relative paths. The absolute path is stored in the SYSCAT.EVENTMONITORS catalog view.

It is possible to specify two or more event monitors that have the same target path. However, once one of the event monitors has been activated for the first time, and as long as the target directory is not empty, it will be impossible to activate any of the other event monitors.

## CREATE EVENT MONITOR

### File Options

Specifies the options for the file format.

#### **MAXFILES NONE**

Specifies that there is no limit to the number of event files that the event monitor will create. This is the default.

#### **MAXFILES** *number-of-files*

Specifies that there is a limit on the number of event monitor files that will exist for a particular event monitor at any time. Whenever an event monitor has to create another file, it will check to make sure that the number of .evt files in the directory is less than *number-of-files*. If this limit has already been reached, then the event monitor will turn itself off.

If an application removes the event files from the directory after they have been written, then the total number of files that an event monitor can produce can exceed *number-of-files*. This option has been provided to allow a user to guarantee that the event data will not consume more than a specified amount of disk space.

#### **MAXFILESIZE** *pages*

Specifies that there is a limit to the size of each event monitor file. Whenever an event monitor writes a new event record to a file, it checks that the file will not grow to be greater than *pages* (in units of 4K pages). If the resulting file would be too large, then the event monitor switches to the next file. The default for this option is:

- OS/2, Windows 95 and Windows NT - 200 4K pages
- UNIX - 1000 4K pages

The number of pages must be greater than at least the size of the event buffer in pages. If this requirement is not met, then an error (SQLSTATE 428A4) is raised.

#### **MAXFILESIZE NONE**

Specifies that there is no set limit on a file's size. If MAXFILESIZE NONE is specified, then MAXFILES 1 must also be specified. This option means that one file will contain all of the event data for a particular event monitor. In this case the only event file will be 00000000.evt.

#### **BUFFERSIZE** *pages*

Specifies the size of the event monitor buffers (in units of 4K pages). All event monitor file I/O is buffered to improve the performance of the event monitors. The larger the buffers, the less I/O will be performed by the event monitor. Highly active event monitors should have larger buffers than relatively inactive event monitors. When the monitor is started, two buffers of the speci-

## CREATE EVENT MONITOR

fied size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The minimum and default size of each buffer (if this option is not specified) is 1 page (that is, 2 buffers, each 4 K in size). The maximum size of the buffers is limited by the size of the database heap (DBHEAP) since the buffers are allocated from the heap. If using a lot of event monitors at the same time, increase the size of the DBHEAP database configuration parameter.

Event monitors that write their data to a pipe also have two internal (non-configurable) buffers that are each 1 page in size. These buffers are also allocated from the database heap (DBHEAP). For each active event monitor that has a pipe target, increase the size of the database heap by 2 pages.

### **BLOCKED**

Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. **BLOCKED** should be selected to guarantee no event data loss. This is the default option.

### **NONBLOCKED**

Specifies that each agent that generates an event should not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. **NONBLOCKED** event monitors do not slow down database operations to the extent of **BLOCKED** event monitors. However, **NONBLOCKED** event monitors are subject to data loss on highly active systems.

### **APPEND**

Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will append the new event data to the existing stream of data files. When the event monitor is re-activated, it will resume writing to the event files as if it had never been turned off. **APPEND** is the default option.

The **APPEND** option does not apply at **CREATE EVENT MONITOR** time, if there is existing event data in the directory where the newly created event monitor is to write its event data.

### **REPLACE**

Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will erase all of the event files and start writing data to file 00000000.evt.

### **MANUALSTART**

Specifies that the event monitor not be started automatically each time the database is started. Event monitors with the **MANUALSTART** option must be activated manually using the **SET EVENT MONITOR STATE** statement. This is the default option.

## CREATE EVENT MONITOR

### AUTOSTART

Specifies that the event monitor be started automatically each time the database is started.

### ON NODE

Keyword that indicates that specific partitions are specified.

#### *node-number*

Specifies a partition number where the event monitor runs and write the events. With the monitoring scope defined as GLOBAL, all partitions report to the specified partition number. The I/O component will physically run on the specified partition, writing its records to /tmp/dlocks directory on that partition.

### GLOBAL

Event monitor reports from all partitions. For a partitioned database in DB2 Universal Database Version 5.2, only deadlock event monitors can be defined as GLOBAL. The global event monitor will report deadlocks for all nodes in the system.

### LOCAL

Event monitor reports only on the partition that is running. It gives a partial trace of the database activity. This is the default.

## Rules

- Each of the event types (DATABASE, TABLES, DEADLOCKS,...) can only be specified once in a particular event monitor definition.

## Notes

- Event monitor definitions are recorded in the SYSCAT.EVENTMONITORS catalog view. The events themselves are recorded in the SYSCAT.EVENTS catalog view.
- For detailed information on using the database monitor and on interpreting data from pipes and files, see the *System Monitor Guide and Reference*.

## Examples

*Example 1:* The following example creates an event monitor called SMITHPAY. This event monitor, will collect event data for the database as well as for the SQL statements performed by the PAYROLL application owned by the JSMITH authorization ID. The data will be appended to the absolute path /home/jsmith/event/smithpay/. A maximum of 25 files will be created. Each file will be a maximum of 1024 4K pages long. The file I/O will be non-blocked.

```
CREATE EVENT MONITOR SMITHPAY
FOR DATABASE, STATEMENTS
WHERE APPL_NAME = 'PAYROLL' AND AUTH_ID = 'JSMITH'
WRITE TO FILE '/home/jsmith/event/smithpay'
MAXFILES 25
MAXFILESIZE 1024
NONBLOCKED
APPEND
```

## CREATE EVENT MONITOR

*Example 2:* The following example creates an event monitor called DEADLOCKS\_EVTs. This event monitor will collect deadlock events and will write them to the relative path DLOCKS. One file will be written, and there is no maximum file size. Each time the event monitor is activated, it will append the event data to the file 00000000.evt if it exists. The event monitor will be started each time the database is started. The I/O will be blocked by default.

```
CREATE EVENT MONITOR DEADLOCK_EVTs  
FOR DEADLOCKS  
WRITE TO FILE 'DLOCKS'  
MAXFILES 1  
MAXFILESIZE NONE  
AUTOSTART
```

*Example 3:* This example creates an event monitor called DB\_APPLs. This event monitor collects connection events, and writes the data to the named pipe /home/jsmith/applpipe.

```
CREATE EVENT MONITOR DB_APPLs  
FOR CONNECTIONS  
WRITE TO PIPE '/home/jsmith/applpipe'
```



---

### CREATE FUNCTION

This statement is used to register a user-defined function with an application server.

There are three different types of functions that can be created using this statement. Each of these is described separately.

- External Scalar

The function is written in a programming language and returns a scalar value. The external executable is registered in the database along with various attributes of the function. See “CREATE FUNCTION (External Scalar)” on page 468.

- External Table

The function is written in a programming language and returns a complete table. The external executable is registered in the database along with various attributes of the function. See “CREATE FUNCTION (External Table)” on page 484.

- Sourced

The function is implemented by invoking another function (either built-in, external or sourced) that is already registered in the database. See “CREATE FUNCTION (Sourced)” on page 497.

## CREATE FUNCTION (External Scalar)

---

### CREATE FUNCTION (External Scalar)

This statement is used to register a user-defined external scalar function with an application server.

A *scalar function* returns a single value each time it is invoked, and is in general valid wherever an SQL expression is valid.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- IMPLICIT\_SCHEMA authority on the database, if the schema name of the function does not refer to an existing schema.
- CREATEIN privilege on the schema, if the schema name of the function refers to an existing schema.

To create a not-fenced function, the privileges held by the authorization ID of the statement must also include at least one of the following:

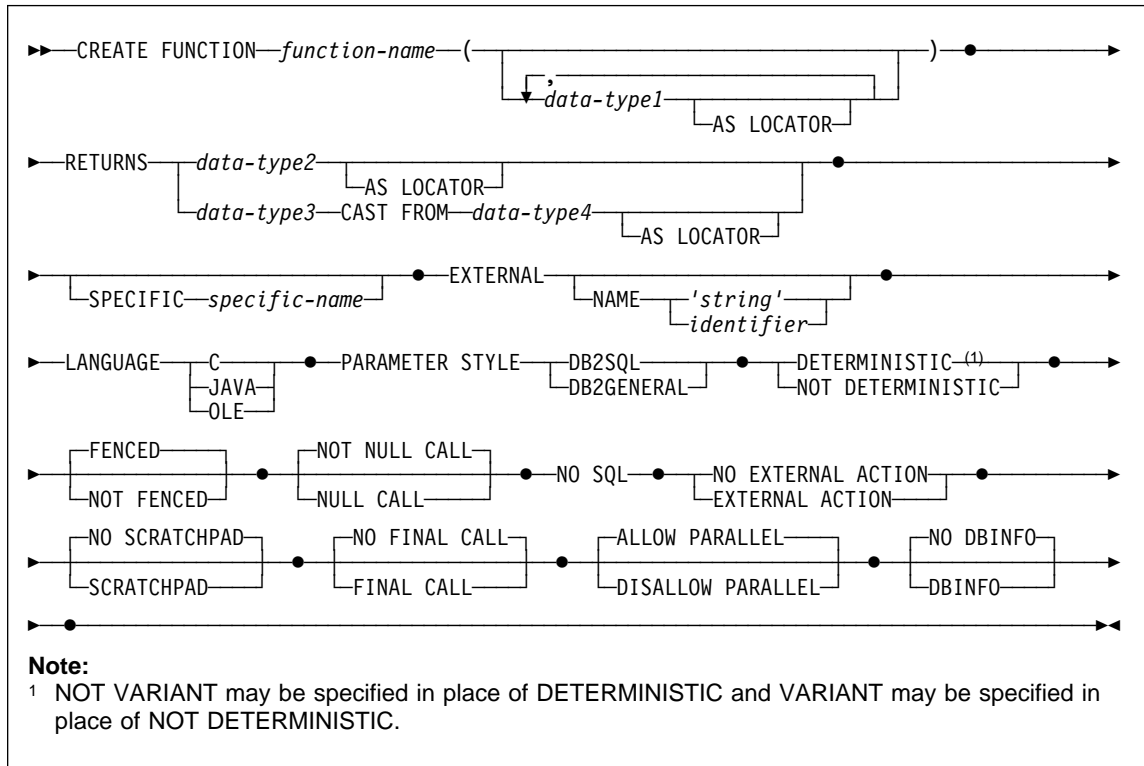
- CREATE\_NOT\_FENCED authority on the database
- SYSADM or DBADM authority.

To create a fenced function, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

#### Syntax

## CREATE FUNCTION (External Scalar)



### Description

#### *function-name*

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a schema-name followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the schema-name cannot begin with "SYS". Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for system use, and may not be used as a *function-name*. The names are SOME, ANY, ALL, NOT,

## CREATE FUNCTION (External Scalar)

AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in “Basic Predicate” on page 136. Failure to observe this rule will lead to an error (SQLSTATE 42939).

In general, the same name can be used for more than one function if there is some difference in the signature of the functions.

Although there is no prohibition against it, an external user-defined function should not be given the same name as a built-in function, unless it is an intentional override. To give a function having a different meaning the same name (for example, LENGTH, VALUE, MAX), with consistent arguments, as a built-in scalar or column function, is to invite trouble for dynamic SQL statements, or when static SQL applications are rebound; the application may fail, or perhaps worse, may appear to run successfully while providing a different result.

*(data-type1,...)*

Identifies the number of input parameters of the function, and specifies the data type of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions and scales are not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL(4,3). There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be duplicate functions.

*data-type1*

Specifies the data type of the parameter.

- SQL data type specifications and abbreviations which may be specified in the *data-type1* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the function may be specified. See the language-specific sections of the *Embedded SQL Programming Guide* for details on the mapping between the SQL data types and host language data types with respect to user-defined functions.

## CREATE FUNCTION (External Scalar)

- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815). For alternatives to using DECIMAL refer to *Embedded SQL Programming Guide*.
- REF(*type-name*) cannot be specified as the data type of a parameter (SQLSTATE 42997).

### AS LOCATOR

For the LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed to the UDF instead of the actual value. This saves greatly in the number of bytes passed to the UDF, and may save as well in performance, particularly in the case where only a few bytes of the value are actually of interest to the UDF. Use of LOB locators in UDFs are described in *Embedded SQL Programming Guide*.

Here is an example which illustrates the use of the AS LOCATOR clause in parameter definitions:

```
CREATE FUNCTION foo (CLOB(10M) AS LOCATOR, IMAGE AS LOCATOR)
    ...
```

which assumes that IMAGE is a distinct type based on one of the LOB types.

Note also that for argument promotion purposes, the AS LOCATOR clause has no effect. In the example the types are considered to be CLOB and IMAGE respectively, which would mean that a CHAR or VARCHAR argument could be passed to the function as the first argument. Likewise, the AS LOCATOR has no effect on the function signature, which is used in matching the function (a) when referenced in DML, by a process called "function resolution", and (b) when referenced in a DDL statement such as COMMENT ON or DROP. In fact the clause may or may not be used in COMMENT ON or DROP with no significance.

An error (SQLSTATE 42601) is raised if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the function is FENCED, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

### RETURNS

This mandatory clause identifies the output of the function.

*data-type2*

Specifies the data type of the output.

In this case, exactly the same considerations apply as for the parameters of external functions described above under *data-type1* for function parameters.

### AS LOCATOR

For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the UDF instead of the actual value.

## CREATE FUNCTION (External Scalar)

*data-type3* **CAST FROM** *data-type4*

Specifies the data type of the output.

This form of the RETURNS clause is used to return a different data type to the invoking statement from the data type that was returned by the function code. For example, in

```
CREATE FUNCTION GET_HIRE_DATE(CHAR(6))
    RETURNS DATE CAST FROM CHAR(10)
```

...

the function code returns a CHAR(10) value to the database manager, which, in turn, converts it to a DATE and passes that value to the invoking statement. The *data-type4* must be *castable* to the *data-type3* parameter. If it is not castable, an error (SQLSTATE 42880) is raised (for the definition of castable, see “Casting Between Data Types” on page 67).

Since the length, precision or scale for *data-type3* can be inferred from *data-type4*, it is not necessary (but still permitted) to specify the length, precision, or scale for parameterized types specified for *data-type3*. Instead empty parentheses may be used (for example VARCHAR() may be used). FLOAT() cannot be used (SQLSTATE 42601) since parameter value indicates different data types (REAL or DOUBLE).

A distinct type is not valid as the type specified in *data-type4* (SQLSTATE 42815).

The cast operation is also subject to run-time checks that might result in conversion errors being raised.

### AS LOCATOR

For *data-type4* specifications that are LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed back from the UDF instead of the actual value. Use of LOB locators in UDFs are described in *Embedded SQL Programming Guide*.

### SPECIFIC *specific-name*

Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a schema-name followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

## CREATE FUNCTION (External Scalar)

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmsshhn.

### EXTERNAL

This clause indicates that the CREATE FUNCTION statement is being used to register a new function based on code written in an external programming language and adhering to the documented linkage conventions and interface.

If NAME clause is not specified "NAME *function-name*" is assumed.

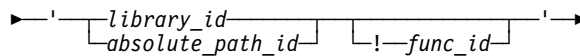
### NAME 'string'

This clause identifies the name of the user-written code which implements the function being defined.

The 'string' option is a string constant with a maximum of 254 characters. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

The *string* specified is the library name and function within library, which the database manager invokes to execute the user-defined function being CREATED. The library (and the function within the library) do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the library and function within the library must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42724) is raised .



The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

#### *library\_id*

Identifies the library name containing the function. The database manager will look for the library in the `.../sqllib/function` directory (UNIX-based systems), or `...\\sqllib\\function` directory (OS/2, Windows 95 and Windows NT ), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is `/u/$DB2INSTANCE/sqllib`.

If 'myfunc' were the *library\_id* in a UNIX-based system it would cause the database manager to look for the function in library `/u/production/sqllib/function/myfunc`, provided the database manager is being run from `/u/production`.

#### *absolute\_path\_id*

Identifies the full path name of the file containing the function.

In a UNIX-based system, for example, `'/u/jchui/mylib/myfunc'` would cause the database manager to look in `/u/jchui/mylib` for the myfunc shared library.

## CREATE FUNCTION (External Scalar)

In OS/2, Windows 95 and Windows NT , 'd:\mylib\myfunc' would cause the database manager to load dynamic link library, myfunc.dll file, from the d:\mylib directory.

### *! func\_id*

Identifies the entry point name of the function to be invoked. The ! serves as a delimiter between the library id and the function id. If ! *func\_id* is omitted, the database manager will use the default entry point established when the library was linked.

In a UNIX-based system, for example, 'mymod!func8' would direct the database manager to look for the library \$inst\_home\_dir/sqlib/function/mymod and to use entry point func8 within that library.

In OS/2, Windows 95 and Windows NT , 'mymod!func8' would direct the database manager to load the mymod.dll file and call the func8() function in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is raised.

The body of every external function should be in a directory which is mounted and available on every partition of the database.

- For LANGUAGE JAVA:

The *string* specified is the class identifier and method identifier, which the database manager invokes to execute the user-defined function being CREATED. The class identifier and method identifier do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42724) is raised .

► '*class\_id*—!*method\_id*' ◄

The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

### *class\_id*

Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, "myPacks.UserFuncs". The Java virtual machine will look in directory "../myPacks/UserFuncs/" for the classes. In OS/2 and Windows 95 and Windows NT, the Java virtual machine will look in directory "..\myPacks\UserFuncs\".

### *method\_id*

Identifies the method name of the Java object to be invoked.

- For LANGUAGE OLE:

The *string* specified is the OLE programmatic identifier (progid) or class identifier (clsid), and method identifier, which the database manager



## CREATE FUNCTION (External Scalar)

invokes to execute the user-defined function being CREATED. The programmatic identifier or class identifier, and method identifier do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42724) is raised .

► '—          —!          —' ►  
          └── *progid* ─┘  
          └── *clsid* ─┘

The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

### *progid*

Identifies the programmatic identifier of the OLE object.

*progid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time. The specified OLE object must be creatable and support late binding (also called IDispatch-based binding).

### *clsid*

Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a *progid* in the case that an OLE object is not registered with a *progid*. The

*clsid* has the form:

{nnnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnn}

where 'n' is an alphanumeric character. *clsid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time.

### *method\_id*

Identifies the method name of the OLE object to be invoked.

### **NAME** *identifier*

This *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

### **LANGUAGE**

This mandatory clause is used to specify the language interface convention to which the user-defined function body is written.

**C** This means the database manager will call the user-defined function as if it were a C function. The user-defined function must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

## CREATE FUNCTION (External Scalar)

**JAVA** This means the database manager will call the user-defined function as a method in a Java class.

**OLE** This means the database manager will call the user-defined function as if it were a method exposed by an OLE automation object. The user-defined function must conform with the OLE automation data types and invocation mechanism as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for user-defined functions stored in DB2 for Windows 95 and Windows NT.

### PARAMETER STYLE

This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

**DB2SQL** Used to specify the conventions for passing parameters to and returning the value from external functions that conform to C language calling and linkage conventions or methods exposed by OLE automation objects. This must be specified when LANGUAGE C or LANGUAGE OLE is used.

**DB2GENERAL** Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. This must be specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

Refer to *Embedded SQL Programming Guide* for details on passing parameters.

### DETERMINISTIC or NOT DETERMINISTIC

This mandatory clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a NOT DETERMINISTIC function would be a random-number generator. An example of a DETERMINISTIC function would be a function that determines the square root of the input.

### FENCED or NOT FENCED

This clause specifies whether or not the function is considered "safe" to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager insulates its internal resources (e.g. data buffers) from access by the function. Most functions will have the option of running as FENCED or NOT FENCED. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

**Warning:** Use of NOT FENCED for functions not adequately checked out can compromise the integrity of DB2. DB2 takes some precautions against

## CREATE FUNCTION (External Scalar)

many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user defined functions are used.

Most user-defined functions should be able to run either as FENCED or NOT FENCED. Only FENCED can be specified for a function with LANGUAGE OLE (SQLSTATE 42613) .

To change from FENCED to NOT FENCED, the function must be re-registered (by first dropping it and then re-creating it). Either SYSADM authority, DBADM authority or a special authority (CREATE\_NOT\_FENCED) is required to register a user-defined function as NOT FENCED.

### **NOT NULL CALL** or **NULL CALL**

This optional clause may be used to avoid a call to the external function if any of the arguments is null.

If NOT NULL CALL is specified and if at execution time any one of the function's arguments is null, the user-defined function is not called and the result is the null value.

If NULL CALL is specified, then regardless of whether any arguments are null, the user-defined function is called. It can return a null value or a normal (non-null) value. But responsibility for testing for null argument values lies with the UDF.

### **NO SQL**

This mandatory clauses indicates that the function cannot issue any SQL statements. If it does, an error (SQLSTATE 38502) is raised at run time.

### **NO EXTERNAL ACTION** or **EXTERNAL ACTION**

This mandatory clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Optimizations that assume functions have no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

### **NO SCRATCHPAD** or **SCRATCHPAD**

This optional clause may be used to specify whether a scratchpad is to be provided for an external function. (It is strongly recommended that user-defined functions be re-entrant, so a scratchpad provides a means for the function to "save state" from one call to the next.)

If SCRATCHPAD is specified, then at first invocation of the user-defined function, memory is allocated for a scratchpad to be used by the external function. This scratchpad has the following characteristics:

- It is 100 bytes in size.
- It is initialized to all X'00's.
- Its scope is the SQL statement. There is one scratchpad per reference to the external function in the SQL statement. So if the UDFX function in the following statement is defined with the SCRATCHPAD keyword, three scratchpads would be assigned.

## CREATE FUNCTION (External Scalar)

```
SELECT A, UDFX(A) FROM TABLEB
WHERE UDFX(A) > 103 OR UDFX(A) < 19
```

If ALLOW PARALLEL is specified or defaulted to, then the scope is different from the above. If the function is executed in multiple partitions, a scratchpad would be assigned in each partition where the function is processed, for each reference to the function in the SQL statement.

- It is persistent. Its content is preserved from one external function call to the next. Any changes made to the scratchpad by the external function on one call will be there on the next call. The database manager initializes scratchpads at the beginning of execution of each SQL statement. The database manager may reset scratchpads at the beginning of execution of each subquery. The system issues a final call before resetting a scratchpad if the FINAL CALL option is specified.
- It can be used as a central point for system resources (for example, memory) which the external function might acquire. The function could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

(In such a case where system resource is acquired, the FINAL CALL keyword should also be specified; this causes a special call to be made at end-of-statement to allow the external function to free any system resources acquired.)

If SCRATCHPAD is specified, then on each invocation of the user-defined function an additional argument is passed to the external function which addresses the scratchpad.

If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

### NO FINAL CALL or FINAL CALL

This optional clause specifies whether a final call is to be made to an external function. The purpose of such a final call is to enable the external function to free any system resources it has acquired. It can be useful in conjunction with the SCRATCHPAD keyword in situations where the external function acquires system resources such as memory and anchors them in the scratchpad. If FINAL CALL is specified, then at execution time:

An additional argument is passed to the external function which specifies the type of call. The types of calls are:

- Normal call: SQL arguments are passed and a result is expected to be returned.
- First call: the first call to the external function for this reference to the user-defined function in this SQL statement. The first call is a normal call.
- Final call: a final call to the external function to enable the function to free up resources. The final call is not a normal call. This final call occurs at the following times:

## CREATE FUNCTION (External Scalar)

- End-of-statement: This case occurs when the cursor is closed for cursor-oriented statements, or when the statement is through executing otherwise.
- End-of-transaction: This case occurs when the normal end-of-statement does not occur. For example, the logic of an application may for some reason bypass the close of the cursor.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made at the subsequent close of the cursor or at the end of the application.

If NO FINAL CALL is specified then no “call type” argument is passed to the external function, and no final call is made.

### **ALLOW PARALLEL or DISALLOW PARALLEL**

This optional clause specifies whether, for a single reference to the function, the invocation of the function can be parallelized. In general, the invocations of most scalar functions should be parallelizable, but there may be functions (such as those depending on a single copy of a scratchpad) that cannot. If either ALLOW PARALLEL or DISALLOW PARALLEL are specified for a scalar function, then DB2 will accept this specification. The following questions should be considered in determining which keyword is appropriate for the function.

- Are all the UDF invocations completely independent of each other? If YES, then specify ALLOW PARALLEL.
- Does each UDF invocation update the scratchpad, providing value(s) that are of interest to the next invocation? (For example, the incrementing of a counter.) If YES, then specify DISALLOW PARALLEL or accept the default.
- Is there some external action performed by the UDF which should happen only on one partition? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is the scratchpad used, but only so that some expensive initialization processing can be performed a minimal number of times? If YES, then specify ALLOW PARALLEL.

In any case, the body of every external function should be in a directory that is mounted and available on every partition of the database.

The syntax diagram indicates that the default value is ALLOW PARALLEL. However, the default is DISALLOW PARALLEL if one or more of the following options is specified in the statement:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- SCRATCHPAD
- FINAL CALL

## CREATE FUNCTION (External Scalar)

### NO DBINFO or DBINFO

This optional clause specifies whether certain specific information known by DB2 will be passed to the UDF as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613).

If DBINFO is specified, then a structure is passed to the UDF which contains the following information:

- Data base name - the name of the currently connected database.
- Application Authorization ID - the application run-time authorization ID, regardless of the nested UDFs in between this UDF and the application.
- Code page - identifies the database code page.
- Schema name - under the exact same conditions as for Table name, contains the name of the schema; otherwise blank.
- Table name - if and only if the UDF reference is either the right-hand side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement, contains the unqualified name of the table being updated or inserted; otherwise blank.
- Column name - under the exact same conditions as for Table name, contains the name of the column being updated or inserted; otherwise blank.
- Database version/release - identifies the version, release and modification level of the database server invoking the UDF.
- Platform - contains the server's platform type.
- Table function result column numbers - not applicable to external scalar functions.

Please see the *Embedded SQL Programming Guide* for detailed information on the structure and how it is passed to the user-defined function.

### Notes

- Determining whether one data type is castable to another data type does not consider length or precision and scale for parameterized data types such as CHAR and DECIMAL. Therefore, errors may occur when using a function as a result of attempting to cast a value of the source data type to a value of the target data type. For example, VARCHAR is castable to DATE but if the source type is actually defined as VARCHAR(5), an error will occur when using the function.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values (see "Promotion of Data Types" on page 66). For example, a constant which may be used as an input value could have a built-in data type different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - INTEGER instead of SMALLINT

## CREATE FUNCTION (External Scalar)

- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC
- For portability of UDFs across platforms the following data types should not be used:
  - FLOAT- use DOUBLE or REAL instead.
  - NUMERIC- use DECIMAL instead.
  - LONG VARCHAR- use CLOB (or BLOB) instead.
- For information on writing, compiling, and linking an external user-defined function, see the *Embedded SQL Programming Guide*.
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

### Examples

*Example 1:* Pellow is registering the CENTRE function in his PELLOW schema. Let those keywords that will default default, and let the system provide a function specific name:

```
CREATE FUNCTION CENTRE (INT,FLOAT)
  RETURNS FLOAT
  EXTERNAL NAME 'mod!middle'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
```

*Example 2:* Now, McBride (who has DBADM authority) is registering another CENTRE function in the PELLOW schema, giving it an explicit specific name for subsequent data definition language use, and explicitly providing all keyword values. Note also that this function uses a scratchpad and presumably is accumulating data there that affects subsequent results. Since DISALLOW PARALLEL is specified, any reference to the function is not parallelized and therefore a single scratchpad is used to perform some one-time only initialization and save the results.

```
CREATE FUNCTION PELLOW.CENTRE (FLOAT, FLOAT, FLOAT)
  RETURNS DECIMAL(8,4) CAST FROM FLOAT
  SPECIFIC FOCUS92
  EXTERNAL NAME 'effects!focalpt'
  LANGUAGE C PARAMETER STYLE DB2SQL
  DETERMINISTIC FENCED NOT NULL CALL NO SQL NO EXTERNAL ACTION
  SCRATCHPAD NO FINAL CALL
  DISALLOW PARALLEL
```

*Example 3:* The following is the C language user-defined function program written to implement the rule:

## CREATE FUNCTION (External Scalar)

```
output = 2 * input - 4
```

returning NULL if and only if the input is null. It could be written even more simply (that is, without the null checking), if the CREATE FUNCTION statement had used NOT NULL CALL. Further examples of user-defined function programs can be found in the *Embedded SQL Programming Guide*. The CREATE FUNCTION statement:

```
CREATE FUNCTION ntest1 (SMALLINT)
RETURNS SMALLINT
EXTERNAL NAME 'ntest1!nudft1'
LANGUAGE C PARAMETER STYLE DB2SQL
DETERMINISTIC NOT FENCED NULL CALL
NO SQL NO EXTERNAL ACTION
```

The program code:

```
#include "sqlsystem.h"

/* NUDFT1 IS A USER_DEFINED SCALAR FUNCTION */

/* udft1 accepts smallint input
and produces smallint output
implementing the rule:

    if (input is null)
        set output = null;
    else
        set output = 2 * input - 4;
*/
void SQL_API_FN nudft1
(short *input,      /* ptr to input arg */
 short *output,    /* ptr to where result goes */
 short *input_ind, /* ptr to input indicator var */
 short *output_ind, /* ptr to output indicator var */
 char sqlstate[6], /* sqlstate, allows for null-term */
 char fname[28],  /* fully qual func name, nul-term */
 char finst[19],  /* func specific name, null-term */
 char msgtext[71]) /* msg text buffer, null-term */
{
    /* first test for null input */
    if (*input_ind == -1)
    {
        /* input is null, likewise output */
        *output_ind = -1;
    }
    else
    {
        /* input is not null. set output to 2*input-4 */
        *output = 2 * (*input) - 4;

        /* and set out null indicator to zero */
        *output_ind = 0;
    }
}
```



## CREATE FUNCTION (External Scalar)

```
    /* signal successful completion by leaving sqlstate as is */
    /* and exit */
    return;
}
/* end of UDF: NUDFT1 */
```

*Example 4:* The following registers a Java UDF which returns the position of the first vowel in a string. The UDF is written in Java, is to be run fenced, and is the findvwl method of class javaUDFs.

```
CREATE FUNCTION findv ( CLOB(100K) )
RETURNS INTEGER
FENCED
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
EXTERNAL NAME 'javaUDFs!findvwl'
NO EXTERNAL ACTION
NULL CALL
DETERMINISTIC
NO SQL
```

## CREATE FUNCTION (External Table)

---

### CREATE FUNCTION (External Table)

This statement is used to register a user-defined external table function with an application server.

A *table function* may be used in the FROM clause of a SELECT, and returns a table to the SELECT by returning one row at a time.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
- CREATEIN privilege on the schema, if the schema name of the function exists.

To create a not-fenced function, the privileges held by the authorization ID of the statement must also include at least one of the following:

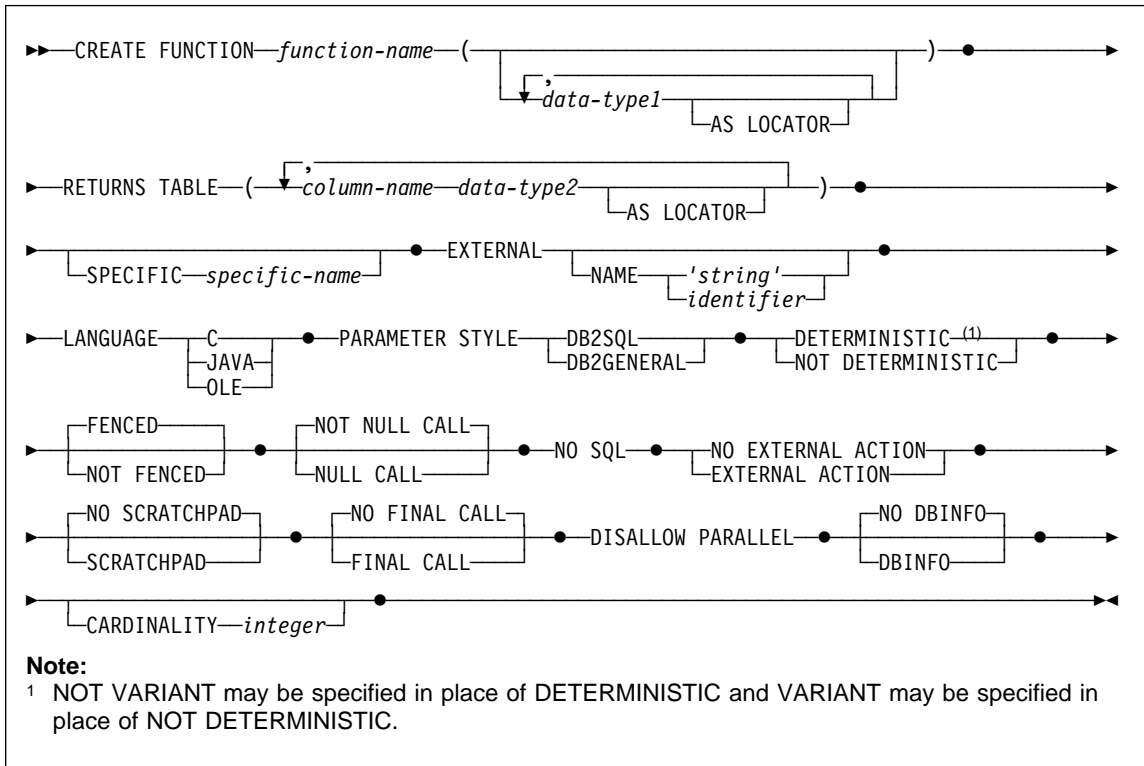
- CREATE\_NOT\_FENCED authority on the database
- SYSADM or DBADM authority.

To create a fenced function, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

#### Syntax

## CREATE FUNCTION (External Table)



### Description

#### *function-name*

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a schema-name followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the schema-name cannot begin with "SYS". Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for system use, and may not be used as a *function-name*. The names are SOME, ANY, ALL, NOT,

## CREATE FUNCTION (External Table)

AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in “Basic Predicate” on page 136 . Failure to observe this rule will lead to an error (SQLSTATE 42939).

The same name can be used for more than one function if there is some difference in the signature of the functions. Although there is no prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

*(data-type1,...)*

Identifies the number of input parameters of the function, and specifies the data type of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions and scales are not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3). There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...  
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...
```

```
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...  
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be a duplicate functions.

*data-type1*

Specifies the data type of the parameter.

- SQL data type specifications and abbreviations which may be specified in the *data-type* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the function may be specified. See the language-specific sections of the *Embedded SQL Programming Guide* for details on the mapping between the SQL data types and host language data types with respect to user-defined functions.
- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815). For alternatives to using DECIMAL refer to *Embedded SQL Programming Guide*.
- REF(*type-name*) cannot be specified as the data type of a parameter (SQLSTATE 42997).

## CREATE FUNCTION (External Table)

### AS LOCATOR

For the LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed to the UDF instead of the actual value. This saves greatly in the number of bytes passed to the UDF, and may save as well in performance, particularly in the case where only a few bytes of the value are actually of interest to the UDF. Use of LOB locators in UDFs are described in *Embedded SQL Programming Guide*.

Here is an example which illustrates the use of the AS LOCATOR clause in parameter definitions:

```
CREATE FUNCTION foo ( CLOB(10M) AS LOCATOR, IMAGE AS LOCATOR)
    ...
```

which assumes that IMAGE is a distinct type based on one of the LOB types.

Note also that for argument promotion purposes, the AS LOCATOR clause has no effect. In the example the types are considered to be CLOB and IMAGE respectively, which would mean that a CHAR or VARCHAR argument could be passed to the function as the first argument. Likewise, the AS LOCATOR has no effect on the function signature, which is used in matching the function (a) when referenced in DML, by a process called "function resolution", and (b) when referenced in a DDL statement such as COMMENT ON or DROP. In fact the clause may or may not be used in COMMENT ON or DROP with no significance.

An error (SQLSTATE 42601) is raised if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the function is FENCED, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

### RETURNS TABLE

Specifies that the output of the function is a table. The parentheses that follow this keyword delimit a list of the names and types of the columns of the table, resembling the style of a simple CREATE TABLE statement which has no additional specifications (constraints, for example).

#### *column-name*

Specifies the name of this column. This is a long-identifier, unique within the table function.

#### *data-type2*

Specifies the data type of the column, and can be any data type supported for a parameter of a UDF, i.e. any type, including a distinct type but excluding DECIMAL and NUMERIC.

### AS LOCATOR

When *data-type2* is a LOB type or distinct type based on a LOB type, the use of this option indicates that the function is returning a locator for the LOB value that is instantiated in the result table.

## CREATE FUNCTION (External Table)

The valid types for use with this clause are discussed on page 470.

### **SPECIFIC** *specific-name*

Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a schema-name followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmsshhn.

### **EXTERNAL**

This clause indicates that the CREATE FUNCTION statement is being used to register a new function based on code written in an external programming language and adhering to the documented linkage conventions and interface.

If NAME clause is not specified "NAME *function-name*" is assumed.

### **NAME** '*string*'

This clause identifies the user-written code which implements the function being defined.

The '*string*' option is a string constant with a maximum of 254 characters. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

The *string* specified is the library name and function within library, which the database manager invokes to execute the user-defined function being CREATED. The library (and the function within the library) do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the library and function within the library must exist and be accessible from the database server machine.

► ' *library\_id* *absolute\_path\_id* *func\_id* ' ►

The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

## CREATE FUNCTION (External Table)

### *library\_id*

Identifies the library name containing the function. The database manager will look for the library in the .../sqllib/function directory (UNIX-based systems), or ...\\sqllib\\function directory (OS/2, Windows 95 and Windows NT), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is /u/\$DB2INSTANCE/sqllib.

If 'myfunc' were the *library\_id* in a UNIX-based system it would cause the database manager to look for the function in library /u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.

### *absolute\_path\_id*

Identifies the full path name of the function.

In a UNIX-based system, for example, '/u/jchui/mylib/myfunc' would cause the database manager to look in /u/jchui/mylib for the myfunc function.

In OS/2, Windows 95 and Windows NT 'd:\mylib\myfunc' would cause the database manager to load the myfunc.dll file from the d:\mylib directory.

### *!func\_id*

Identifies the entry point name of the function to be invoked. The ! serves as a delimiter between the library id and the function id. If !*func\_id* is omitted, the database manager will use the default entry point established when the library was linked.

In a UNIX-based system, for example, 'mymod!func8' would direct the database manager to look for the library \$inst\_home\_dir/sqllib/function/mymod and to use entry point func8 within that library.

In OS/2, Windows 95 and Windows NT, 'mymod!func8' would direct the database manager to load the mymod.dll file and call the func8() function in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is raised.

In any case, the body of every external function should be in a directory which is mounted and available on every partition of the database.

- For LANGUAGE JAVA:

The *string* specified is the class identifier and method identifier, which the database manager invokes to execute the user-defined function being CREATED. The class identifier and method identifier do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine.

## CREATE FUNCTION (External Table)

► '—*class\_id*—!—*method\_id*—' ►

The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

*class\_id*

Identifies the class identifier of the Java object.

*method\_id*

Identifies the method name of the Java object to be invoked.

- For LANGUAGE OLE:

The *string* specified is the OLE programmatic identifier (*progid*) or class identifier (*clsid*), and method identifier, which the database manager invokes to execute the user-defined function being CREATED. The programmatic identifier or class identifier, and method identifier do not need to exist when the CREATE FUNCTION statement is performed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42724) is raised .

► '—*progid*—!—*method\_id*—' ►  
    └──┬──┘  
    └──┬──┘  
    *clsid*

The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

*progid*

Identifies the programmatic identifier of the OLE object.

*progid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time. The specified OLE object must be creatable and support late binding (also called IDispatch-based binding).

*clsid*

Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a *progid* in the case that an OLE object is not registered with a *progid*. The *clsid* has the form:

{nnnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnn}

where 'n' is an alphanumeric character. *clsid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time.

*method\_id*

Identifies the method name of the OLE object to be invoked.

### NAME *identifier*

This clause identifies the name of the user-written code which implements the function being defined. The *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier,



## CREATE FUNCTION (External Table)

the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

### LANGUAGE

This mandatory clause is used to specify the language interface convention to which the user-defined function body is written.

**C** This means the database manager will call the user-defined function as if it were a C function. The user-defined function must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

**JAVA** This means the database manager will call the user-defined function as a method in a Java class.

**OLE** This means the database manager will call the user-defined function as if it were a method exposed by an OLE automation object. The user-defined function must conform with the OLE automation data types and invocation mechanism as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for user-defined functions stored in DB2 for Windows 95 and Windows NT.

### PARAMETER STYLE

This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

**DB2SQL** Used to specify the conventions for passing parameters to and returning the value from external functions that conform to C language calling and linkage conventions. This must be specified when LANGUAGE C or LANGUAGE OLE is used.

**DB2GENERAL** Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. This must be specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

Refer to *Embedded SQL Programming Guide* for details on passing parameters.

### DETERMINISTIC or NOT DETERMINISTIC

This mandatory clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same table from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a NOT DETERMINISTIC table function would be a function that retrieves data from a data source such as a file.

## CREATE FUNCTION (External Table)

### **FENCED** or **NOT FENCED**

This clause specifies whether or not the function is considered “safe” to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager insulates its internal resources (e.g. data buffers) from access by the function. Most functions will have the option of running as FENCED or NOT FENCED. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

**Warning:** Use of NOT FENCED for functions not adequately checked out can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user defined functions are used.

Most user-defined functions should be able to run either as FENCED or NOT FENCED. Only FENCED can be specified for a function with LANGUAGE OLE (SQLSTATE 42613).

To change from FENCED to NOT FENCED, the function must be re-registered (by first dropping it and then re-creating it). Either SYSADM authority, DBADM authority or a special authority (CREATE\_NOT\_FENCED) is required to register a user-defined function as NOT FENCED.

### **NOT NULL CALL** or **NULL CALL**

This optional clause may be used to avoid a call to the external function if any of the arguments is null.

If NOT NULL CALL is specified and if at execution time any one of the function's arguments is null, the user-defined function is not called and the result is the empty table, i.e. a table with no rows.

If NULL CALL is specified, then at execution time regardless of whether any arguments are null, the user-defined function is called. It can return an empty table or not, depending on its logic. But responsibility for testing for null argument values lies with the UDF.

### **NO SQL**

This mandatory clause indicates that the function cannot issue any SQL statements. If it does, an error (SQLSTATE 38502) is raised at run time.

### **NO EXTERNAL ACTION** or **EXTERNAL ACTION**

This mandatory clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Optimizations that assume functions have no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

### **NO SCRATCHPAD** or **SCRATCHPAD**

This optional clause may be used to specify whether a scratchpad is to be provided for an external function. (It is strongly recommended that user-defined func-

## CREATE FUNCTION (External Table)

tions be re-entrant, so a scratchpad provides a means for the function to “save state” from one call to the next.)

If **SCRATCHPAD** is specified, then at first invocation of the user-defined function, memory is allocated for a scratchpad to be used by the external function. This scratchpad has the following characteristics:

- It is 100 bytes in size.
- It is initialized to all X'00's.
- Its scope is the SQL statement. There is one scratchpad per reference to the external function in the SQL statement. So if the **UDFX** function in the following statement is defined with the **SCRATCHPAD** keyword, two scratchpads would be assigned.

```
SELECT A.C1, B.C2 FROM
           TABLE (UDFX(:hv1))AS A,
           TABLE (UDFX(:hv1))AS B
WHERE ...
```

- It is persistent. It is initialized at the beginning of the execution of the statement, and can be used by the external table function to preserve the state of the scratchpad from one call to the next. If the **FINAL CALL** keyword is also specified for the UDF, then the scratchpad is **NEVER** altered by DB2, and any resources anchored in the scratchpad should be released when the special **FINAL** call is made.

If **NO FINAL CALL** is specified or defaulted, then the external table function should clean up any such resources on the **CLOSE** call, as DB2 will re-initialize the scratchpad on each **OPEN** call. This determination of **FINAL CALL** or **NO FINAL CALL** and the associated behavior of the scratchpad could be an important consideration, particularly if the table function will be used in a subquery or join, since that is when multiple **OPEN** calls can occur during the execution of a statement.

For external table functions, the call-type argument is **ALWAYS** present, regardless of which option is chosen. See *Embedded SQL Programming Guide* for more information about this argument and its values.

- It can be used as a central point for system resources (for example, memory) which the external function might acquire. The function could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

(As outlined above, the **FINAL CALL/NO FINAL CALL** keyword is used to control the re-initialization of the scratchpad, and also dictates when the external table function should release resources anchored in the scratchpad.)

If **SCRATCHPAD** is specified, then on each invocation of the user-defined function an additional argument is passed to the external function which addresses the scratchpad.

## CREATE FUNCTION (External Table)

If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

### NO FINAL CALL or FINAL CALL

This optional clause specifies whether a final call (and a separate first call) is to be made to an external function. It also controls when the scratchpad is re-initialized. If NO FINAL CALL is specified, then DB2 can only make three types of calls to the table function: open, fetch and close. However, if FINAL CALL is specified, then in addition to open, fetch and close, a first call and a final call can be made to the table function.

### DISALLOW PARALLEL

This clause specifies that, for a single reference to the function, the invocation of the function can not be parallelized. Table functions are always run on a single partition.

### NO DBINFO or DBINFO

This optional clause specifies whether certain specific information known by DB2 will be passed to the UDF as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613).

If DBINFO is specified, then a structure is passed to the UDF which contains the following information:

- Data base name - the name of the currently connected database.
- Application Authorization ID - the application run-time authorization ID, regardless of the nested UDFs in between this UDF and the application.
- Code page - identifies the database code page.
- Schema name - not applicable to external table functions.
- Table name - not applicable to external table functions.
- Column name - not applicable to external table functions.
- Database version/release- identifies the version, release and modification level of the database server invoking the UDF.
- Platform - contains the server's platform type.
- Table function result column numbers - an array of the numbers of the table function result columns actually needed for the particular statement referencing the function. Only provided for table functions, it enables the UDF to optimize by only returning the required column values instead of all column values.

Please see the *Embedded SQL Programming Guide* for detailed information on the structure and how it is passed to the UDF.

### CARDINALITY *integer*

This optional clause provides an estimate of the expected number of rows to be returned by the function for optimization purposes. Valid values for *integer* range from 0 to 2147483647 inclusive.

## CREATE FUNCTION (External Table)

If the `CARDINALITY` clause is not specified for a table function, DB2 will assume a finite value as a default- the same value assumed for tables for which the `RUNSTATS` utility has not gathered statistics.

Warning: if a function does in fact have infinite cardinality, i.e. it returns a row every time it is called to do so, never returning the "end-of-table" condition, then queries which require the "end-of-table" condition to correctly function will be infinite, and will have to be interrupted. Examples of such queries are those involving `GROUP BY` and `ORDER BY`. The user is advised to not write such UDFs.

### Notes

- Determining whether one data type is castable to another data type does not consider length or precision and scale for parameterized data types such as `CHAR` and `DECIMAL`. Therefore, errors may occur when using a function as a result of attempting to cast a value of the source data type to a value of the target data type. For example, `VARCHAR` is castable to `DATE` but if the source type is actually defined as `VARCHAR(5)`, an error will occur when using the function.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values (see "Promotion of Data Types" on page 66). For example, a constant which may be used as an input value could have a built-in data type different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use 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 UDFs across platforms the following data types should not be used:
  - `FLOAT`- use `DOUBLE` or `REAL` instead.
  - `NUMERIC`- use `DECIMAL` instead.
  - `LONG VARCHAR`- use `CLOB` (or `BLOB`) instead.
- For information on writing, compiling, and linking an external user-defined function, see the *Embedded SQL Programming Guide*.
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has `IMPLICIT_SCHEMA` authority. The schema owner is `SYSIBM`. The `CREATEIN` privilege on the schema is granted to `PUBLIC`.

### Examples

*Example 1:* 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.

## CREATE FUNCTION (External Table)

Within the context of a single session, the UDF will always return the same table, and therefore it is defined as DETERMINISTIC. Note the RETURNS clause which defines the output from DOCMATCH. FINAL CALL must be specified for each table function. In addition, the DISALLOW PARALLEL keyword is added as 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 the DB2 optimizer.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16))
  EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20
```

*Example 2:* The following registers an OLE table function that is used to retrieve message header information and the partial message text of messages in Microsoft Exchange. For an example of the code that implements this table function, see the *Embedded SQL Programming Guide*.

```
CREATE FUNCTION MAIL()
  RETURNS TABLE (TIMERECEIVED DATE,
                 SUBJECT VARCHAR(15),
                 SIZE INTEGER,
                 TEXT VARCHAR(30))
  EXTERNAL NAME 'tfmail.header!list'
  LANGUAGE OLE
  PARAMETER STYLE DB2SQL
  NOT DETERMINISTIC
  FENCED
  NULL CALL
  SCRATCHPAD
  FINAL CALL
  NO SQL
  EXTERNAL ACTION
  DISALLOW PARALLEL
```

### CREATE FUNCTION (Sourced)

This statement is used to register a user-defined function, based on another existing scalar or column function, with an application server.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

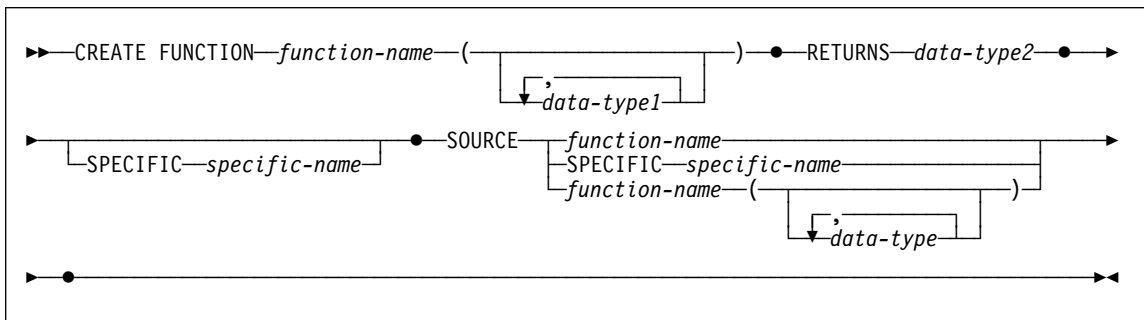
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
- CREATEIN privilege on the schema, if the schema name of the function exists.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

No authority is required on a function referenced in the SOURCE clause.

#### Syntax



#### Description

*function-name*

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a schema-name followed by a period and an SQL identifier.

## CREATE FUNCTION (Sourced)

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the schema-name cannot begin with "SYS". Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for system use, and may not be used as a *function-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in "Basic Predicate" on page 136. Failure to observe this rule will lead to an error (SQLSTATE 42939).

When naming a user-defined function that is sourced on an existing function with the purpose of supporting the same function with a user-defined distinct type, the same name as the sourced function may be used. This allows users to use the same function with a user-defined distinct type without realizing that an additional definition was required. In general, the same name can be used for more than one function if there is some difference in the signature of the functions.

*(data-type,...)*

Identifies the number of input parameters of the function, and specifies the data type of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions and scales are not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL(4,3). There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...  
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...
```

```
CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...  
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be a duplicate functions.



## CREATE FUNCTION (Sourced)

### *data-type1*

Specifies the data type of the parameter.

With a sourced scalar function any valid SQL data type may be used provided it is castable to the type of the corresponding parameter of the function identified in the SOURCE clause (for the definition of castable, see “Casting Between Data Types” on page 67). A REF(*type-name*) data type cannot be specified as the data type of a parameter (SQLSTATE 42997).

Since the function is sourced, it is not necessary (but still permitted) to specify length, precision, or scale for the parameterized data types. Instead, empty parentheses may be used (for example CHAR() may be used). A *parameterized data type* is any one of the data types that can be defined with a specific length, scale, or precision. The parameterized data types are the string data types and the decimal data types.

### RETURNS

This mandatory clause identifies the output of the function.

### *data-type2*

Specifies the data type of the output.

Any valid SQL data type is valid, as is a distinct type, provided it is castable from the result type of the source function (for the definition of castable, see “Casting Between Data Types” on page 67).

The parameter of a parameterized type need not be specified, as above for parameters of a sourced function. Instead, empty parentheses may be used, for example, VARCHAR().

Also see page on page 501 for additional considerations and rules that apply to the specification of the data type in the RETURNS clause when the function is sourced on another.

### SPECIFIC *specific-name*

Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a schema-name followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmsshhn.

## CREATE FUNCTION (Sourced)

### SOURCE

Specifies that the function being created is to be implemented by another function (the source function) already known to the database manager. The source function can be either a built-in function<sup>59</sup> or a previously created user-defined scalar function (including a sourced function).

The SOURCE clause may be specified only for scalar or column functions; it may not be specified for table functions.

The SOURCE clause provides the identity of the other function.

#### *function-name*

Identifies the particular function that is to be used as the source and is valid only if there is exactly one specific function in the schema with this *function-name*. This syntax variant is not valid for a source function that is a built-in function.

If an unqualified name is provided, then the authorization ID's current SQL path (the value of the CURRENT PATH special register) is used to locate the function. The first schema in the function path that has a function with this name is selected.

If no function by this name exists in the named schema or if the name is not qualified and there is no function with this name in the function path, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the function in the named or located schema, an error (SQLSTATE 42725) is raised.

#### **SPECIFIC** *specific-name*

Identifies the particular user-defined function that is to be used as the source, by the *specific-name* either specified or defaulted to at function creation time. This syntax variant is not valid for a source function that is a built-in function.

If an unqualified name is provided, then the authorization ID's current SQL path is used to locate the function. The first schema in the function path that has a function with this specific name is selected.

If no function by this *specific-name* exists in the named schema or if the name is not qualified and there is no function with this *specific-name* in the SQL path, an error (SQLSTATE 42704) is raised.

#### *function-name (data-type,...)*

Provides the function signature, which uniquely identifies the source function. This is the only valid syntax variant for a source function that is a built-in function.

The rules for function resolution (as described in "Function Resolution" on page 112) are applied to select one function from the functions with the same function name, given the data types specified in the SOURCE clause.

However, the data type of each parameter in the function selected must have

<sup>59</sup> With the exception of COALESCE, NODENUMBER, NULLIF, PARTITION, TYPE\_ID, TYPE\_NAME, TYPE\_SCHEMA and VALUE.

## CREATE FUNCTION (Sourced)

the exact same type as the corresponding data type specified in the source function.

### *function-name*

Gives the function name of the source function. If an unqualified name is provided, then the schemas of the user's SQL path are considered.

### *data-type*

Must match the data type that was specified on the CREATE FUNCTION statement in the corresponding position (comma separated).

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match. For example, DECIMAL() will match a parameter whose data type was defined as DECIMAL(7,2)).

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement. This can be useful in assuring that the exact intended function will be used. Also note that synonyms for data types will be considered a match (for example DEC and NUMERIC will match).

A type of FLOAT(n) does not need to match the defined value for n since  $0 < n < 25$  means REAL and  $24 < n < 54$  means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

## Rules

- **Rules for Functions Created with the SOURCE Clause:**

For convenience, in this section we will call the function being created CF and the function identified in the SOURCE clause SF, no matter which of the three allowable syntaxes was used to identify SF.

- The unqualified name of CF and the unqualified name of SF can be different.
- A function named as the source of another function can, itself, use another function as its source. Extreme care should be exercised when exploiting this facility because it could be very difficult to debug an application if an indirectly invoked function raises an error.
- The following clauses are invalid if specified in conjunction with the SOURCE clause (because CF will inherit these attributes from SF):

- CAST FROM ...,
- EXTERNAL ...,
- LANGUAGE ...,
- PARAMETER STYLE ...,

## CREATE FUNCTION (Sourced)

DETERMINISTIC / NOT DETERMINISTIC,  
FENCED / NOT FENCED,  
NULL CALL / NOT NULL CALL.  
EXTERNAL ACTION / NO EXTERNAL ACTION  
NO SQL  
SCRATCHPAD / NO SCRATCHPAD  
FINAL CALL / NO FINAL CALL  
RETURNS TABLE (...)  
CARDINALITY ...  
ALLOW PARALLEL / DISALLOW PARALLEL  
DBINFO / NO DBINFO

An error (SQLSTATE 42613) will result from violation of these rules.

- The number of input parameters in CF must be the same as those in SF; otherwise an error (SQLSTATE 42624) is raised.
- It is not necessary for CF to specify length, precision, or scale for a parameterized data type in the case of:
  - The function's input parameters,
  - Its RETURNS parameter

Instead, empty parentheses may be specified as part of the data type (for example: VARCHAR()) in order to indicate that the length/precision/scale will be the same as those of the source function, or determined by the casting.

However, if length, precision, or scale is specified then the value in CF is checked against the corresponding value in SF as outlined below for input parameters and returns value.

- The specification of the input parameters of CF are checked against those of SF. The data type of each parameter of CF must either be the same as or be *castable* to the data type of the corresponding parameter of SF. For the definition of *castable*, see “Casting Between Data Types” on page 67. If any parameter is not the same type or *castable*, an error (SQLSTATE 42879) is raised.

Note that this rule provides no guarantee against an error occurring when CF is used. An argument that matches the data type and length or precision attributes of a CF parameter may not be assignable if the corresponding SF parameter has a shorter length or less precision. In general, parameters of CF should not have length or precision attributes that are greater than the attributes of the corresponding SF parameters.

- The specifications for the RETURNS data type of CF are checked against that of SF. The final RETURNS data type of SF, after any casting, must either be the same as or *castable* to the RETURNS data type of CF. Otherwise an error (SQLSTATE 42866) is raised.

Note that this rule provides no guarantee against an error occurring when CF is used. A result value that matches the data type and length or precision attributes of the SF RETURNS data type may not be assignable if the CF RETURNS data type has a shorter length or less precision. Caution should be used when choosing to

## CREATE FUNCTION (Sourced)

specify the RETURNS data type of CF as having length or precision attributes that are less than the attributes of the SF RETURNS data type.

### Notes

- Determining whether one data type is castable to another data type does not consider length or precision and scale for parameterized data types such as CHAR and DECIMAL. Therefore, errors may occur when using a function as a result of attempting to cast a value of the source data type to a value of the target data type. For example, VARCHAR is castable to DATE but if the source type is actually defined as VARCHAR(5), an error will occur when using the function.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values (see “Promotion of Data Types” on page 66). For example, a constant which may be used as an input value could have a built-in data type different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

### Examples

*Example 1:* Some time after the creation of Pellow's original CENTRE external scalar function, another user wants to create a function based on it, except this function is intended to accept only integer arguments.

```
CREATE FUNCTION MYCENTRE (INTEGER, INTEGER)
  RETURNS FLOAT
  SOURCE PELLOW.CENTRE (INTEGER, FLOAT)
```

*Example 2:* You have created a distinct type HATSIZE which is based on the built-in INTEGER data type, and now would find it useful to have an AVG function to compute the average hat size of different departments. This is easily done as follows:

```
CREATE FUNCTION AVG (HATSIZE) RETURNS (HATSIZE)
  SOURCE SYSIBM.AVG (INTEGER)
```

The creation of the distinct type has generated the required cast function, allowing the cast from HATSIZE to INTEGER for the argument and from INTEGER to HATSIZE for the result of the function.

# CREATE INDEX

---

## CREATE INDEX

The CREATE INDEX statement creates an index on a table.

### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

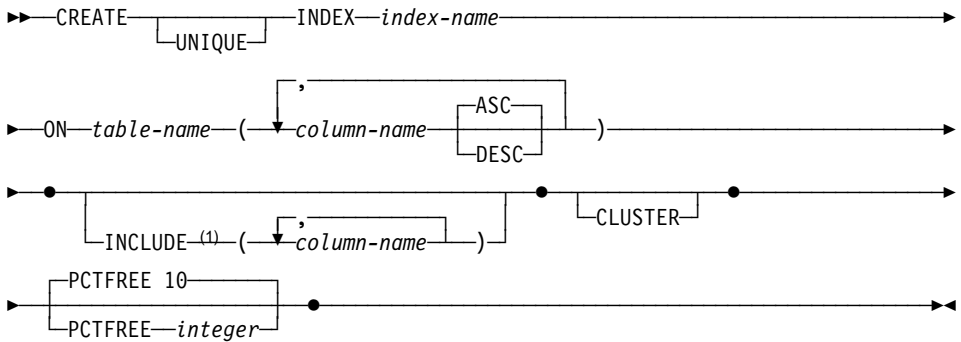
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority.
- One of:
  - CONTROL privilege on the table
  - INDEX privilege on the table

and one of:

- IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the index does not exist
- CREATEIN privilege on the schema, if the schema name of the index refers to an existing schema .

### Syntax



**Note:**

<sup>1</sup> The INCLUDE clause may only be specified if UNIQUE is specified.

### Description

#### UNIQUE

Prevents the table from containing two or more rows with the same value of the index key. The uniqueness is enforced at the end of the SQL statement that updates rows or inserts new rows. For details refer to Appendix I, “Interaction of Triggers and Constraints” on page 887.

## CREATE INDEX

The uniqueness 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.

When UNIQUE is used, null values are treated as any other values. For example, if the key is a single column that may contain null values, that column may contain no more than one null value.

If the UNIQUE option is specified and the table has a partitioning key, the columns in the index key must be a superset of the partitioning key. That is, the columns specified for a unique index key must include all the columns of the partitioning key (SQLSTATE 42997).

### **INDEX** *index-name*

Names the index. The name, including the implicit or explicit qualifier, must not identify an index described in the catalog. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939)

### **ON** *table-name*

Names the table on which the index is to be created. The *table-name* must name a base table (not a view) or a summary table described in the catalog. It must *not* name a catalog table (SQLSTATE 42832). If UNIQUE is specified and *table-name* is a typed table, it must not be a subtable (SQLSTATE 429B3). If UNIQUE is specified, the *table-name* cannot be a summary table (SQLSTATE 42809).

### *column-name*

Identifies a column that is to be part of the index key. Each *column-name* must be an unqualified name that identifies a column of the table. 16 columns or less may be specified. If *table-name* is a typed table, 15 columns or less may be specified. If *table-name* is a subtable, at least one *column-name* must be introduced in the subtable (not inherited from a supertable) (SQLSTATE 428DS). No *column-name* may be repeated (SQLSTATE 42711).

The sum of the length attributes of the specified columns must not be greater than 255. If *table-name* is a typed table, the sum of the length attributes of the specified columns must not be greater than 251. Note that this figure can be reduced by system overhead which varies according to the data type of the column and whether it is nullable. See Byte Counts on page 553 for more information on overhead affecting this limit. No LONG VARCHAR, LONG VARGRAPHIC, LOB or DATALINK column may be used as part of an index (even if the length attribute of the column is small enough to fit within the 255 byte limit (SQLSTATE 42962)).

### **ASC**

Puts the index entries in ascending order by the column. This is the default.

### **DESC**

Puts the index entries in descending order by the column.

### **INCLUDE**

This keyword introduces a clause that specifies additional columns to be appended to the set of index key columns. Any columns included with this clause are not used to enforce uniqueness. These included columns may improve the perform-

## CREATE INDEX

ance of some queries through index only access. The columns must be distinct from the columns used to enforce uniqueness (SQLSTATE 42711). The limits for the number of columns and sum of the length attributes apply to all of the columns in the unique key and in the index.

### *column-name*

Identifies a column that is included in the index but not part of the unique index key. The same rules apply as defined for columns of the unique index key. The keywords ASC or DESC may be specified following the column-name but have no effect on the order.

### **CLUSTER**

Specifies that the index is the clustering index of the table. The cluster factor of a clustering index is maintained or improved dynamically as data is inserted into the associated table, by attempting to insert new rows physically close to the rows for which the key values of this index are in the same range. Only one clustering index may exist for a table so CLUSTER may not be specified if it was used in the definition of any existing index on the table (SQLSTATE 55012). A clustering index may not be created on a table that is defined to use append mode (SQLSTATE 428D8).

### **PCTFREE** *integer*

Specifies what percentage of each index page to leave as free space when building the index. The first entry in a page is added without restriction. When additional entries are placed in an index page at least *integer* percent of free space is left on each page. The value of *integer* can range from 0 to 99. However, if a value greater than 10 is specified, only 10 percent free space will be left in non-leaf pages. The default is 10.

## Rules

- The CREATE INDEX statement will fail (SQLSTATE 01550) if attempting to create an index that matches an existing index. Two index descriptions are considered duplicates if:
  - the set of columns (both key and include columns) and their order in the index is the same as that of an existing index AND
  - the ordering attributes are the same AND
  - both the previously existing index and the one being created are non-unique OR the previously existing index is unique AND
  - if both the previously existing index and the one being created are unique, the key columns of the index being created are the same or a superset of key columns of the previously existing index.
- A unique index must exist on the underlying table of a replicated summary table where the index key columns are included in the select list of the query that defines the replicated summary table. One such index is selected when the replicated summary table is created as a system required index. This index cannot be dropped unless the replicated summary table is dropped. (SQLSTATE 42917).



## CREATE INDEX

### Notes

- If the named table already contains data, 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.
- Once the index is created and data is loaded into the table, it is advisable to issue the RUNSTATS command. (See *Command Reference* for information about RUNSTATS.) The RUNSTATS command updates statistics collected on the database tables, columns, and indexes. These statistics are used to determine the optimal access path to the tables. By issuing the RUNSTATS command, the database manager can determine the characteristics of the new index.
- Creating an index with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

### Examples

*Example 1:* Create an index named UNIQUE\_NAM on the PROJECT table. The purpose of the index is to ensure that there are not two entries in the table with the same value for project name (PROJNAME). The index entries are to be in ascending order.

```
CREATE UNIQUE INDEX UNIQUE_NAM  
ON PROJECT(PROJNAME)
```

*Example 2:* Create an index named JOB\_BY\_DPT on the EMPLOYEE table. Arrange the index entries in ascending order by job title (JOB) within each department (WORKDEPT).

```
CREATE INDEX JOB_BY_DPT  
ON EMPLOYEE (WORKDEPT, JOB)
```

## CREATE NODEGROUP

---

### CREATE NODEGROUP

The CREATE NODEGROUP statement creates a new nodegroup within the database and assigns partitions or nodes to the nodegroup, and records the nodegroup definition in the catalog.

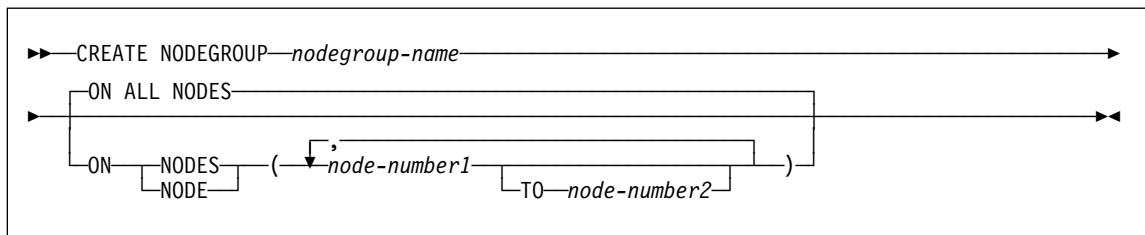
#### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be prepared dynamically.

#### Authorization

The authorization ID of the statement must have SYSCTRL or SYSADM or authority.

#### Syntax



#### Description

##### *nodegroup-name*

Names the nodegroup. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *nodegroup-name* must not identify a nodegroup that already exists in the catalog (SQLSTATE 42710). The *nodegroup-name* must not begin with the characters "SYS" or "IBM" (SQLSTATE 42939).

##### **ON ALL NODES**

Specifies that the nodegroup is defined over all partitions defined to the database (db2nodes.cfg file) at the time the nodegroup is created.

If a partition is added to the database system, the ALTER NODEGROUP statement should be issued to include this new partition in a nodegroup (including IBMDEFAULTGROUP). Furthermore, the REDISTRIBUTE NODEGROUP command must be issued to move data to the partition. Refer to the *API Reference* or the *Command Reference* for more information.

##### **ON NODES**

Specifies the specific partitions that are in the nodegroup. NODE is a synonym for NODES.

##### *node-number1*

## CREATE NODEGROUP

Specify a specific partition number.<sup>60</sup>

### TO *node-number2*

Specify a range of partition numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers are included in the nodegroup.

## Rules

- Each partition or node specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729).
- Each *node-number* listed in the ON NODES clause must appear at most once (SQLSTATE 42728).
- A valid *node-number* is between 0 and 999 inclusive (SQLSTATE 42729).

## Notes

- This statement creates a partitioning map for the nodegroup (Refer to “Data Partitioning Across Multiple Partitions” on page 41 for more information) . A partitioning map identifier (PMAP\_ID) is generated for each partitioning map. This information is recorded in the catalog and can be retrieved from SYSCAT.NODEGROUPS and SYSCAT.PARTITIONMAPS. Each entry in the partitioning map specifies the target partition on which all rows that are hashed reside. For a single-partition nodegroup, the corresponding partitioning map has only one entry. For a multiple partition nodegroup, the corresponding partitioning map has 4096 entries, where the partition numbers are assigned to the map entries in a round-robin fashion, by default.

## Example

Assume that you have a partitioned database with six partitions defined as: 0, 1, 2, 5, 7, and 8.

- Assume that you want to create a nodegroup call MAXGROUP on all six partitions . The statement is as follows:

```
CREATE NODEGROUP MAXGROUP  
ON ALL NODES
```

- Assume that you want to create a nodegroup MEDGROUP on partitions 0, 1, 2, 5, 8. The statement is as follows:

```
CREATE NODEGROUP MEDGROUP  
ON NODES (0 TO 2, 5, 8)
```

- Assume that you want to create a single-partition nodegroup MINGROUP on partition (or node) 7. The statement is as follows:

```
CREATE NODEGROUP MINGROUP  
ON NODE (7)
```

---

<sup>60</sup> node-name of the form 'NODEnnnn' may be specified for compatibility with the previous version.

## CREATE NODEGROUP

**Note:** The singular form of the keyword NODES is also accepted.

---

### CREATE PROCEDURE

This statement is used to register a stored procedure with an application server.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the procedure does not exist
- CREATEIN privilege on the schema, if the schema name of the procedure refers to an existing schema .

To create a not-fenced stored procedure, the privileges held by the authorization ID of the statement must also include at least one of the following:

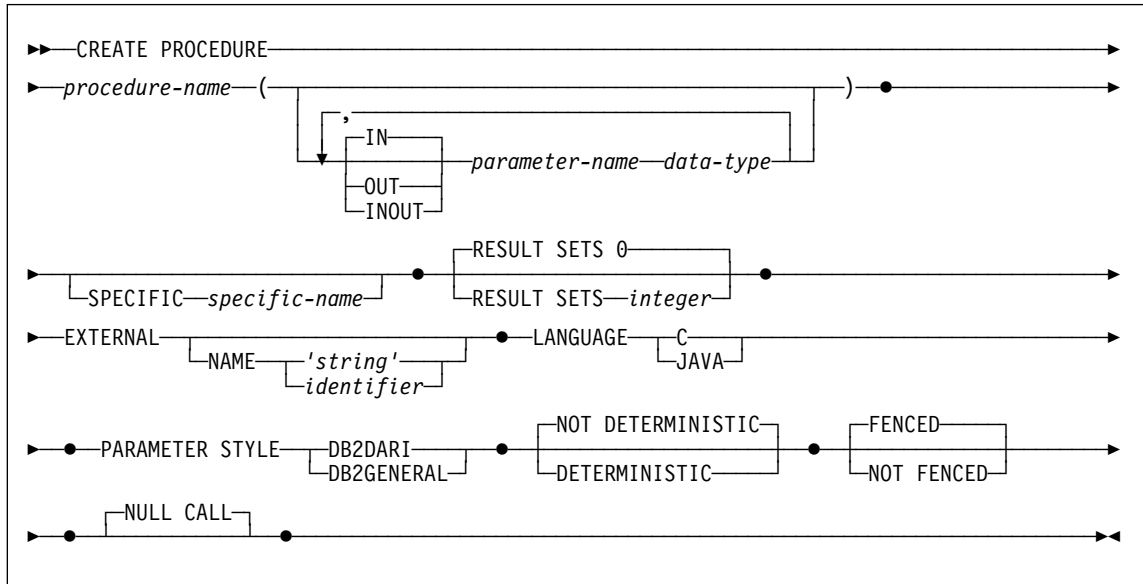
- CREATE\_NOT\_FENCED authority on the database
- SYSADM or DBADM authority.

To create a fenced stored procedure, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

#### Syntax

## CREATE PROCEDURE



### Description

#### *procedure-name*

Names the procedure being defined. It is a qualified or unqualified name that designates a procedure. The unqualified form of *procedure-name* is an SQL identifier (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a schema-name followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters must not identify a procedure described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of the parameters, while of course unique within its schema, need not be unique across schemas.

The result can name is specified, the schema-name cannot begin with "SYS". Otherwise, an error (SQLSTATE 42939) is raised.

#### ( IN | OUT | INOUT *parameter-name data-type*,...)

Identifies the parameters of the procedure, and specifies the mode, name and data type of each parameter. One entry in the list must be specified for each parameter that the procedure will expect.

It is possible to register a procedure that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

```
CREATE PROCEDURE SUBWOOFER() ...
```

No two identically-named procedures within a schema are permitted to have exactly the same number of parameters. Lengths, precisions and scales are not

## CREATE PROCEDURE

considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3). There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE PROCEDURE PART (IN NUMBER INT, OUT PART_NAME CHAR(35)) ...
CREATE PROCEDURE PART (IN COST DECIMAL(5,3), OUT COUNT INT) ...
```

the second statement will fail because the number of parameters of the procedure are the same even if the data types are not.

### IN | OUT | INOUT

Specifies the mode of the parameter.

- IN - parameter is input only
- OUT - parameter is output only
- INOUT - parameter is both input and output

### *parameter-name*

Specifies the name of the parameter.

### *data-type*

Specifies the data type of the parameter.

- SQL data type specifications and abbreviations which may be specified in the *data-type* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the procedure may be specified. See the language-specific sections of the *Embedded SQL Programming Guide* for details on the mapping between the SQL data types and host language data types with respect to store procedures.
- User-defined data types are not supported (SQLSTATE 42601).

### **SPECIFIC** *specific-name*

Provides a unique name for the instance of the procedure that is being defined.

This specific name can be used when dropping the procedure or commenting on the procedure. It can never be used to invoke the procedure. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another procedure instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *procedure-name*.

If no qualifier is specified, the qualifier that was used for *procedure-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *procedure-name* or an error (SQLSTATE 42882) is raised.

## CREATE PROCEDURE

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmsshhn.

### RESULT SETS *integer*

Indicates the estimated upper bound of returned result sets for the stored procedure. Refer to Returning Result Sets from Stored Procedures on page 413 for more information.

### EXTERNAL

This clause indicates that the CREATE PROCEDURE statement is being used to register a new procedure based on code written in an external programming language and adhering to the documented linkage conventions and interface.

If NAME clause is not specified "NAME *procedure-name*" is assumed.

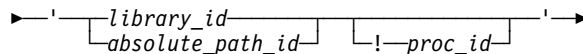
### NAME '*string*'

This clause identifies the name of the user-written code which implements the procedure being defined.

The '*string*' option is a string constant with a maximum of 254 characters. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

The *string* specified is the library name and procedure within the library, which the database manager invokes to execute the stored procedure being CREATED. The library (and the procedure within the library) do not need to exist when the CREATE PROCEDURE statement is performed. However, when the procedure is called, the library and procedure within the library must exist and be accessible from the database server machine.



The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

#### *library\_id*

Identifies the library name containing the procedure. The database manager will look for the library in the .../sqllib/function/unfenced directory and the .../sqllib/function directory (UNIX-based systems), or ...\sqllib\function\unfenced directory and the...\sqllib\function directory (OS/2, Windows 95 and Windows NT), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is /u/\$DB2INSTANCE/sqllib.

If 'myproc' were the *library\_id* in a UNIX-based system it would cause the database manager to look for the procedure in library /u/production/sqllib/function/unfenced/myfunc and



## CREATE PROCEDURE

/u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.

Stored procedures located in any of these directories do not use any of the registered attributes.

### *absolute\_path\_id*

Identifies the full path name of the procedure.

In a UNIX-based system, for example, '/u/jchui/mylib/myproc' would cause the database manager to look in /u/jchui/mylib for the myproc procedure.

In OS/2, Windows 95 and Windows NT 'd:\mylib\myproc' would cause the database manager to load the myproc.dll file from the d:\mylib directory.

If absolute path is specified, the procedure will run as fenced, ignoring the FENCED or NOT FENCED attribute.

### *!proc\_id*

Identifies the entry point name of the procedure to be invoked. The ! serves as a delimiter between the library id and the procedure id. If !*proc\_id* is omitted, the database manager will use the default entry point established when the library was linked.

In a UNIX-based system, for example, 'mymod!proc8' would direct the database manager to look for the library \$inst\_home\_dir/sqllib/function/mymod and to use entry point proc8 within that library.

In OS/2, Windows 95 and Windows NT 'mymod!proc8' would direct the database manager to load the mymod.dll file and call the proc8() procedure in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is raised.

The body of every stored procedure should be in a directory which is mounted and available on every partition of the database.

- For LANGUAGE JAVA:

The *string* specified is the class identifier and method identifier, which the database manager invokes to execute the stored procedure being CREATED. The class identifier and method identifier do not need to exist when the CREATE PROCEDURE statement is performed. However, when the procedure is called, the class identifier and the method identifier must exist and be accessible from the database server machine, otherwise an error (SQLSTATE 42884) is raised .

►—*class\_id*—!*method\_id*—►

The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

## CREATE PROCEDURE

### *class\_id*

Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, "myPacks.StoredProcs". The Java virtual machine will look in directory "../myPacks/StoredProcs/" for the classes. In OS/2 and Windows 95 and Windows NT, the Java virtual machine will look in directory "..\myPacks\StoredProcs\".

### *method\_id*

Identifies the method name with the Java class to be invoked.

### **NAME** *identifier*

This *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

### **LANGUAGE**

This mandatory clause is used to specify the language interface convention to which the stored procedure body is written.

**C** This means the database manager will call the stored procedure as if it were a C procedure. The stored procedure must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

**JAVA** This means the database manager will call the stored procedure as a method in a Java class.

### **PARAMETER STYLE**

This clause is used to specify the conventions used for passing parameters to and returning the value from stored procedures.

**DB2DARI** This means that the stored procedure will use a parameter passing convention that conforms to C language calling and linkage conventions. This must be specified when LANGUAGE C is used.

**DB2GENERAL** This means that the stored procedure will use a parameter passing convention that is defined for use with Java methods. This must be specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

Refer to *Embedded SQL Programming Guide* for details on passing parameters.

### **DETERMINISTIC** or **NOT DETERMINISTIC**

This clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same result from successive invocations with identical inputs.

## CREATE PROCEDURE

### FENCED or NOT FENCED

This clause specifies whether or not the stored procedure is considered “safe” to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a stored procedure is registered as FENCED, the database manager insulates its internal resources (e.g. data buffers) from access by the procedure. All procedures have the option of running as FENCED or NOT FENCED. In general, a procedure running as FENCED will not perform as well as a similar one running as NOT FENCED.

If the stored procedure is located in `...\sqllib\function\unfenced` directory and the `...\sqllib\function` directory (UNIX-based systems), or `...\sqllib\function\unfenced` directory and the `...\sqllib\function` directory (OS/2, Windows 95 and Windows NT), then the FENCED or NOT FENCED registered attribute (and every other registered attribute) will be ignored.

**Warning:** Use of NOT FENCED for procedures not adequately checked out can compromise the integrity of DB2. DB2 takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED stored procedures are used.

To change from FENCED to NOT FENCED, the procedure must be re-registered (by first dropping it and then re-creating it). Either SYSADM authority, DBADM authority or a special authority (CREATE\_NOT\_FENCED) is required to register a stored procedures as NOT FENCED.

### NULL CALL

NULL CALL always applies to stored procedures. This means that regardless if any arguments are null, the stored procedure is called. It can return a null value or a normal (non-null) value. Responsibility for testing for null argument values lies with the stored procedure.

## Notes

- For information on creating the programs for a stored procedure, see the *Embedded SQL Programming Guide*.
- Creating a procedure with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

## Examples

*Example 1:* Create the procedure definition for a stored procedure, written in Java, that is passed a part number and returns the cost of the part and the quantity that are currently available.

## CREATE PROCEDURE

```
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,  
                                OUT COST    DECIMAL(7,2),  
                                OUT QUANTITY INTEGER)  
  EXTERNAL NAME 'parts!onhand'  
  LANGUAGE JAVA PARAMETER STYLE DB2GENERAL
```

*Example 2:* Create the procedure definition for a stored procedure, written in C, that is passed an assembly number and returns the number of parts that make up the assembly, total part cost and a result set that lists the part numbers, quantity and unit cost of each part.

```
CREATE PROCEDURE ASSEMBLY_PARTS (IN ASSEMBLY_NUM INTEGER,  
                                 OUT NUM_PARTS   INTEGER,  
                                 OUT COST       DOUBLE)  
  EXTERNAL NAME 'parts!assembly'  
  RESULT SETS 1 NOT FENCED  
  LANGUAGE C PARAMETER STYLE DB2DARI
```

## CREATE SCHEMA

The CREATE SCHEMA statement defines a schema. It is also possible to create some objects and grant privileges on objects within the statement.

### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

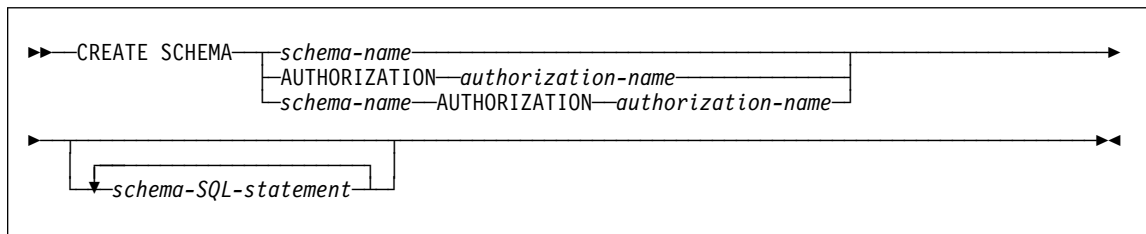
An authorization ID that holds SYSADM or DBADM authority can create a schema with any valid *schema-name* or *authorization-name*.

An authorization ID that does not hold SYSADM or DBADM authority can only create a schema with a *schema-name* or *authorization-name* that matches the authorization ID of the statement.

If the statement includes any *schema-SQL-statements* the privileges held by the *authorization-name* (if not specified, it defaults to the authorization ID of the statement) must include at least one of the following:

- The privileges required to perform each of the *schema-SQL-statements*
- SYSADM or DBADM authority.

### Syntax



### Description

#### *schema-name*

Names the schema. The name must not identify a schema already described in the catalog (SQLSTATE 42710). The name cannot begin with "SYS" (SQLSTATE 42939). The owner of the schema is the authorization ID that issued the statement.

#### **AUTHORIZATION** *authorization-name*

Identifies the user that is the owner of the schema. The value *authorization-name* is also used to name the schema. The *authorization-name* must not identify a schema already described in the catalog (SQLSTATE 42710). The *authorization-name* cannot begin with "SYS" (SQLSTATE 42602) .

## CREATE SCHEMA

*schema-name* **AUTHORIZATION** *authorization-name*

Identifies a schema called *schema-name* with the user called *authorization-name* as the schema owner. The *schema-name* must not identify a schema-name for a schema already described in the catalog (SQLSTATE 42710). The *schema-name* cannot begin with "SYS" (SQLSTATE 42939).

*schema-SQL-statements*

SQL statements that can be included as part of the CREATE SCHEMA statement are:

- CREATE TABLE statement excluding typed tables and summary tables (see "CREATE TABLE" on page 522)
- CREATE VIEW statement (see "CREATE VIEW" on page 582)
- CREATE INDEX statement (see "CREATE INDEX" on page 504)
- COMMENT ON statement (see "COMMENT ON" on page 418)
- GRANT statement (see "GRANT (Table or View Privileges)" on page 652).

## Notes

- The owner of the schema is determined as follows:
  - If an AUTHORIZATION clause is specified, the specified *authorization-name* is the schema owner
  - If an AUTHORIZATION clause is not specified, the authorization ID that issued the CREATE SCHEMA statement is the schema owner.
- The schema owner is assumed to be a user (not a group).
- When the schema is explicitly created with the CREATE SCHEMA statement, the schema owner is granted CREATEIN, DROPIN, and ALTERIN privileges on the schema with the ability to grant these privileges to other users.
- The definer of any object created as part of the CREATE SCHEMA statement is the schema owner. The schema owner is also the grantor for any privileges granted as part of the CREATE SCHEMA statement.
- Unqualified object names in any SQL statement within the CREATE SCHEMA statement are implicitly qualified by the name of the created schema.
- If the CREATE statement contains a qualified name for the object being created, the schema name specified in the qualified name must be the same as the name of the schema being created (SQLSTATE 42875). Any other objects referenced within the statements may be qualified with any valid schema name.
- If the AUTHORIZATION clause is specified and DCE authentication is used, the group membership of the *authorization-name* specified will not be considered in evaluating the authorizations required to perform the statements that follow the clause. If the *authorization-name* specified is different than the authorization id creating the schema, an authorization failure may result during the execution of the CREATE SCHEMA statement.

## CREATE SCHEMA

### Examples

*Example 1:* As a user with DBADM authority, create a schema called RICK with the user RICK as the owner.

```
CREATE SCHEMA RICK AUTHORIZATION RICK
```

*Example 2:* Create a schema that has an inventory part table and an index over the part number. Give authority on the table to user JONES.

```
CREATE SCHEMA INVENTORY
```

```
CREATE TABLE PART (PARTNO SMALLINT NOT NULL,  
DESCR VARCHAR(24),  
QUANTITY INTEGER)
```

```
CREATE INDEX PARTIND ON PART (PARTNO)
```

```
GRANT ALL ON PART TO JONES
```

*Example 3:* Create a schema called PERS with two tables that each have a foreign key that references the other table. This is an example of a feature of the CREATE SCHEMA statement that allows such a pair of tables to be created without the use of the ALTER TABLE statement.

```
CREATE SCHEMA PERS
```

```
CREATE TABLE ORG (DEPTNUMB SMALLINT NOT NULL,  
DEPTNAME VARCHAR(14),  
MANAGER SMALLINT,  
DIVISION VARCHAR(10),  
LOCATION VARCHAR(13),  
CONSTRAINT PKEYDNO  
PRIMARY KEY (DEPTNUMB),  
CONSTRAINT FKEYMGR  
FOREIGN KEY (MANAGER)  
REFERENCES STAFF (ID) )
```

```
CREATE TABLE STAFF (ID SMALLINT NOT NULL,  
NAME VARCHAR(9),  
DEPT SMALLINT,  
JOB VARCHAR(5),  
YEARS SMALLINT,  
SALARY DECIMAL(7,2),  
COMM DECIMAL(7,2),  
CONSTRAINT PKEYID  
PRIMARY KEY (ID),  
CONSTRAINT FKEYDNO  
FOREIGN KEY (DEPT)  
REFERENCES ORG (DEPTNUMB) )
```

## CREATE TABLE

---

### CREATE TABLE

The CREATE TABLE statement defines a table. The definition must include its name and the names and attributes of its columns. The definition may include other attributes of the table, such as its primary key or check constraints.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CREATETAB authority on the database and one of:
  - IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the table does not exist
  - CREATEIN privilege on the schema, if the schema name of the table refers to an existing schema.

If a subtable is being defined, the authorization ID must be the same as the definer of the root table of the table hierarchy.

To define a foreign key, the privileges held by the authorization ID of the statement must include one of the following on the parent table:

- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SYSADM or DBADM authority.

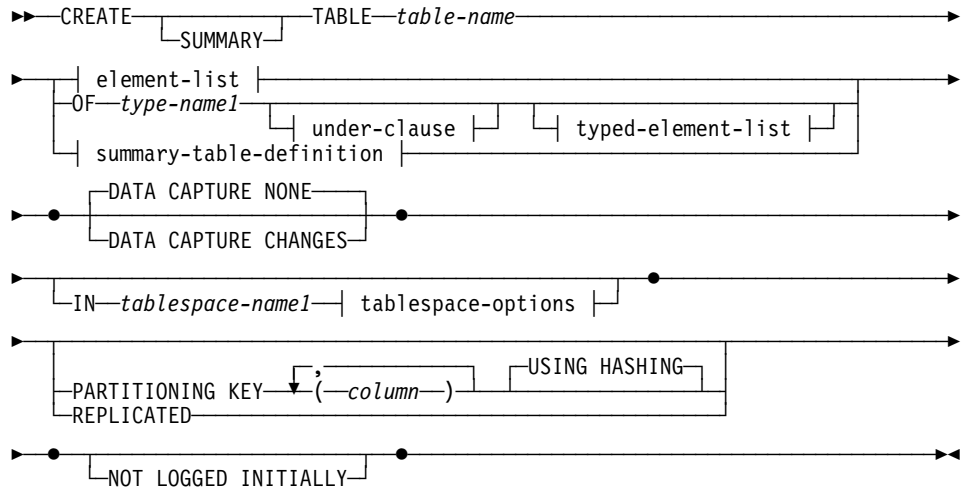
To define a summary table (using a fullselect) the privileges held by the authorization ID of the statement must include at least one of the following on each table or view identified in the fullselect:

- SELECT privilege on the table or view and ALTER privilege if REFRESH DEFERRED or REFRESH IMMEDIATE is specified
- CONTROL privilege on the table or view
- SYSADM or DBADM authority.

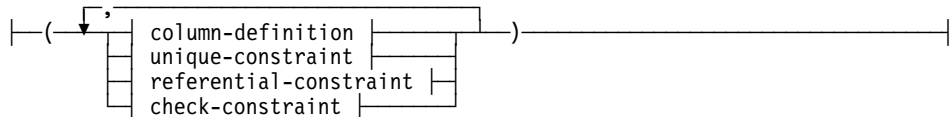
#### Syntax



## CREATE TABLE



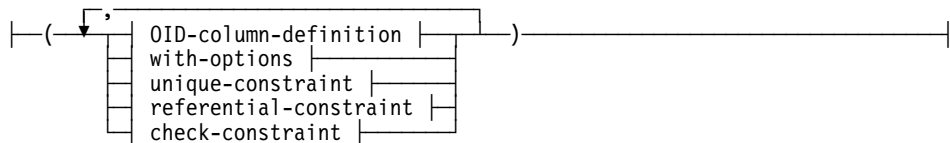
### element-list:



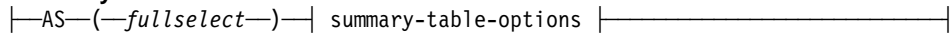
### under-clause:



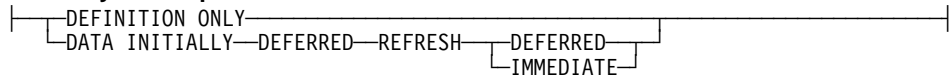
### typed-element-list:



### summary-table-definition:

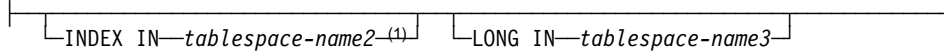


### summary-table-options:

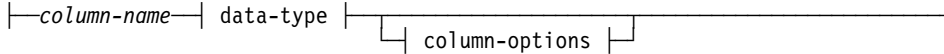


## CREATE TABLE

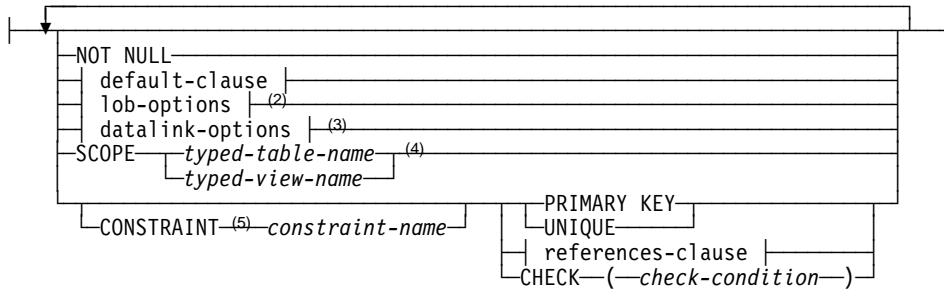
### tablespace-options:



### column-definition:



### column-options:

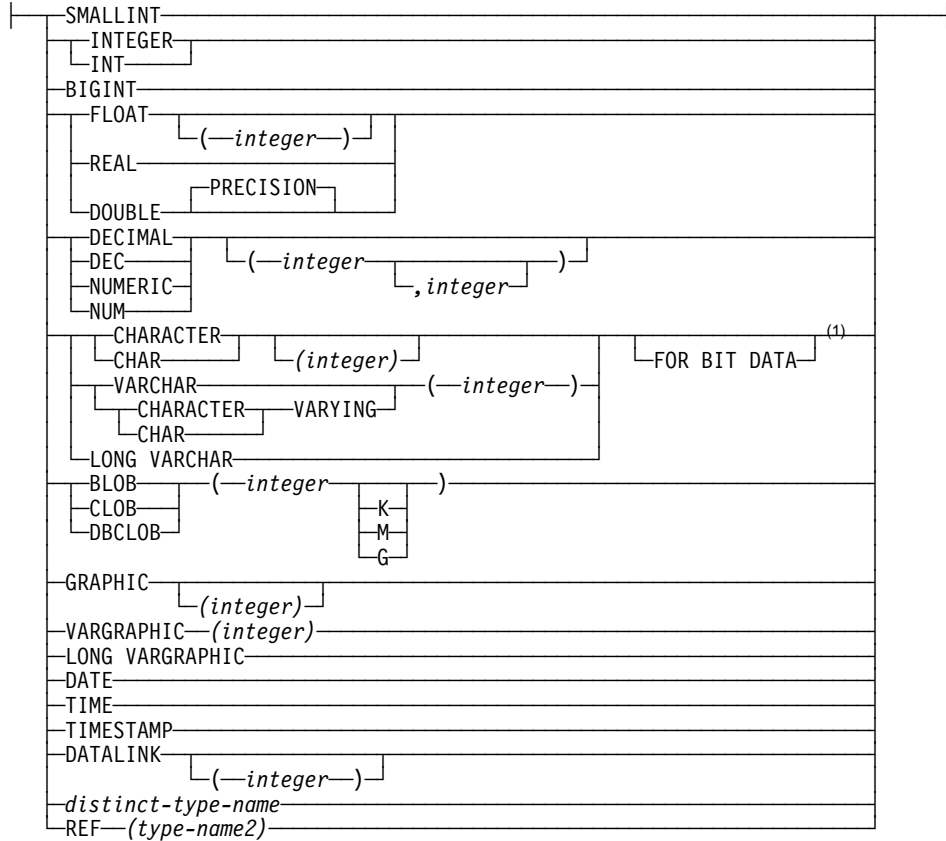


### Notes:

- 1 Specifying which table space will contain a table's index can only be done when the table is created.
- 2 The lob-options clause only applies to large object types (BLOB, CLOB and DBCLOB) and distinct types based on large object types.
- 3 The datalink-options clause only applies to the DATALINK type and distinct types based on the DATALINK type. The LINKTYPE URL clause is required for these types.
- 4 The SCOPE clause only applies to the REF type.
- 5 For compatibility with Version 1, the CONSTRAINT keyword may be omitted in a *column-definition* defining a references-clause.

## CREATE TABLE

### data-type:

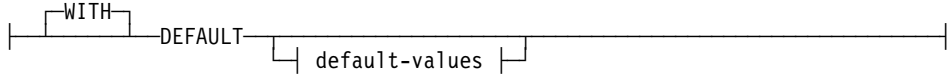


### Note:

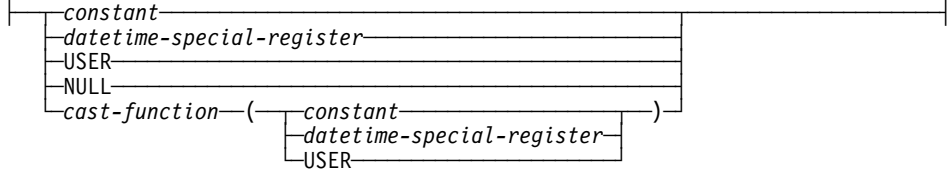
- <sup>1</sup> The FOR BIT DATA clause may be specified in random order with the other column constraints that follow.

# CREATE TABLE

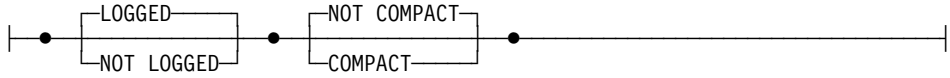
## default-clause:



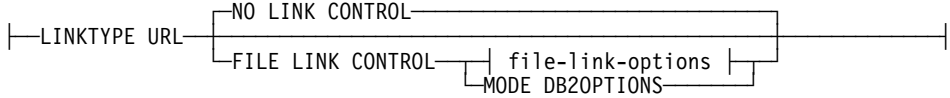
## default-values:



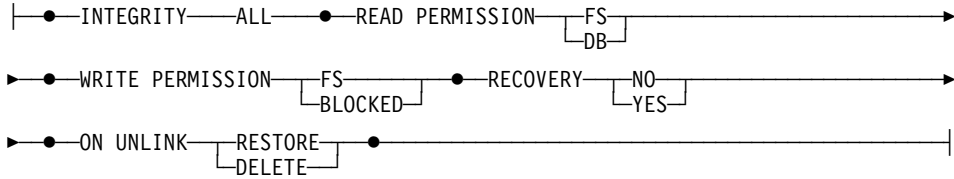
## lob-options:



## datalink-options:

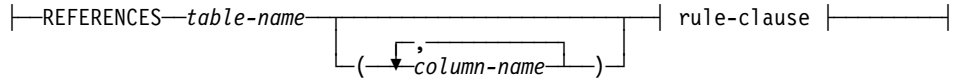


## file-link-options:

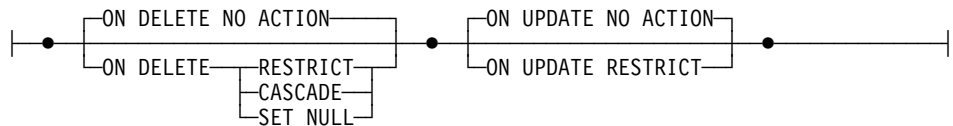


## CREATE TABLE

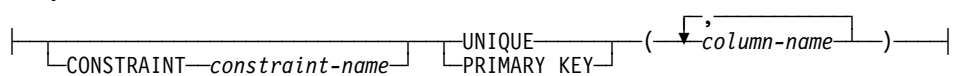
### references-clause:



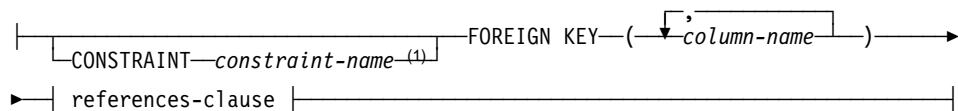
### rule-clause:



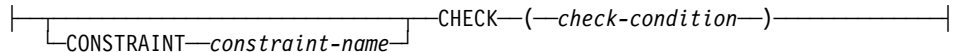
### unique-constraint:



### referential-constraint:



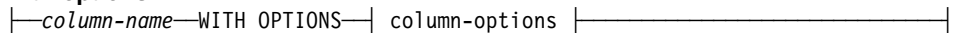
### check-constraint:



### OID-column-definition:



### with-options:



### Note:

<sup>1</sup> For compatibility with Version 1, `constraint-name` may be specified following `FOREIGN KEY` (without the `CONSTRAINT` keyword).

## Description

### SUMMARY

Indicates that a summary table is being defined. The keyword is optional, but when specified, the statement must include a *summary-table-definition*.

### table-name

Names the table. The name, including the implicit or explicit qualifier, must not identify a table, view, or alias described in the catalog. The schema name must not be `SYSIBM`, `SYSCAT`, `SYSFUN`, or `SYSSTAT` (SQLSTATE 42939).

### OF type-name1

Specifies that the columns of the table are based on the attributes of the structured type identified by *type-name1*. If *type-name1* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the `FUNCPATH` preprocessing option for static SQL and by the `CURRENT PATH` register for dynamic SQL). The type name must be the name of an existing user-

## CREATE TABLE

defined type (SQLSTATE 42704) and it must be a structured type (SQLSTATE 428DP).

If UNDER is not specified, an object identifier column must be specified (refer to the *OID-column-definition*) as the first column of the table. The object ID column is followed by columns based on the attributes of *type-name1*.

### **UNDER** *supertable-name*

Indicates that the table is a subtable of *supertable-name*. The supertable must be an existing table (SQLSTATE 42704) and the table must be defined using a structured type that is the immediate supertype of *type-name1* (SQLSTATE 428DB). The schema name of *table-name* and *supertable-name* must be the same (SQLSTATE 428DQ). The table identified by *supertable-name* must not have any existing subtable already defined using *type-name1* (SQLSTATE 42742).

The columns of the table include the object identifier column of the supertable with its type modified to be REF(*type-name1*), followed by columns based on the attributes of *type-name1* (remember that the type includes the attributes of its supertype). The attribute names cannot be the same as the OID column name (SQLSTATE 42711).

Other table options including table space, data capture, not logged initially and partitioning key options cannot be specified. These options are inherited from the supertable (SQLSTATE 42613).

### **INHERIT SELECT PRIVILEGES**

Any user or group holding a SELECT privilege on the supertable will be granted an equivalent privilege on the newly created subtable. The subtable definer is considered to be the grantor of this privilege.

#### *element-list*

Defines the elements of a table. This includes the definition of columns and constraints on the table.

#### *typed-element-list*

Defines the additional elements of a typed table. This includes the additional options for the columns, the addition of an object identifier column (root table only), and constraints on the table.

#### *summary-table-definition*

If the table definition is based on the result of a query, then the table is a summary table based on the query.

**AS** Introduces the query that is used for the definition of the table or to determine the data included in the table.

#### *fullselect*

Defines the query in which the table is based. The *summary-table-options* specified define attributes of the summary table. The option chosen also defines the contents of the fullselect as follows:

## CREATE TABLE

Defines the query in which the table is based. The *summary-table-options* define attributes of the summary table. The option chosen also defines the contents of the fullselect as follows.

When REFRESH DEFERRED or REFRESH IMMEDIATE is specified, the fullselect cannot include:

- references to a view, summary table, or typed table in any FROM clause
- expressions that are a reference type or DATALINK type (or distinct type based on these types)
- functions that have external action
- functions that depend on physical characteristics (for example NODENUMBER, PARTITION)
- table or view references to system objects (explain tables also should not be specified).

When REFRESH IMMEDIATE is specified, the fullselect must be a subselect and cannot include:

- functions that are not deterministic
- scalar fullselects
- predicates with fullselects
- special registers
- DISTINCT in the SELECT clause
- *expression* in the select list that are other than column names
- FROM clause references other than to a single base table
- GROUP BY clause and HAVING clause.

Furthermore, for REFRESH IMMEDIATE, the base table must have at least one unique index defined and the SELECT clause must include all of the columns of this unique index.

### *summary-table-options*

Define the attributes of the summary table.

#### **DEFINITION ONLY**

Data is used to define the elements of the table. The table is not populated using the results of query and the REFRESH TABLE statement cannot be used. When the CREATE TABLE statement is completed, the table is no longer considered a summary table.

#### **DATA INITIALLY DEFERRED**

Data is not inserted into the table as part of the CREATE TABLE statement. A REFRESH TABLE statement specifying the *table-name* is used to insert data into the table.

## CREATE TABLE

### REFRESH

Indicates how the data in the table is maintained.

### DEFERRED

The data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time the REFRESH TABLE statement is processed. Summary tables defined with this attribute do not allow INSERT, UPDATE or DELETE statements (SQLSTATE 42807).

### IMMEDIATE

The changes made to the underlying tables as part of a DELETE, INSERT, or UPDATE are cascaded to the summary table. In this case, the content of the table, at any point-in-time, is the same as if the specified *subselect* is processed. Summary tables defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807).

### *column-definition*

Defines the attributes of a column.

### *column-name*

Names a column of the table. The name cannot be qualified and the same name cannot be used for more than one column of the table.

A table may have the following:

- a 4K page size with maximum of 500 columns where the byte counts of the columns must not be greater than 4005 in a 4K page size, or
- an 8K page size with maximum of 1012 columns where the byte counts of the columns must not be greater than 8101.

### *data-type*

Is one of the types in the following list. Use:

### **SMALLINT**

For a small integer.

### **INTEGER** or **INT**

For a large integer.

### **BIGINT**

For a big integer.

### **FLOAT**(*integer*)

For a single or double precision floating-point number, depending on the value of the *integer*. The value of the integer must be in the range 1 through 53. The values 1 through 24 indicate single precision and the values 25 through 53 indicate double precision.

You can also specify:

### **REAL**

For single precision floating-point.

### **DOUBLE**

For double precision floating-point.



## CREATE TABLE

**DOUBLE PRECISION** For double precision floating-point.  
**FLOAT** For double precision floating-point.

**DECIMAL**(*precision-integer*, *scale-integer*) or **DEC**(*precision-integer*, *scale-integer*)

For a decimal number. The first integer is the precision of the number; that is, the total number of digits; it may 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; it may range from 0 to the precision of the number.

If precision and scale are not specified, the default values of 5,0 are used. The words **NUMERIC** and **NUM** can be used as synonyms for **DECIMAL** and **DEC**.

**CHARACTER**(*integer*) or **CHAR**(*integer*) or **CHARACTER** or **CHAR**

For a fixed-length character string of length *integer*, which may range from 1 to 254. If the length specification is omitted, a length of 1 character is assumed.

**VARCHAR**(*integer*), or **CHARACTER VARYING**(*integer*), or **CHAR VARYING**(*integer*)

For a varying-length character string of maximum length *integer*, which may range from 1 to 4000.

**LONG VARCHAR**

For a varying-length character string with a maximum length of 32700.

**FOR BIT DATA**

Specifies that the contents of the column are to be treated as bit (binary) data. During data exchange with other systems, code page conversions are not performed. Comparisons are done in binary, irrespective of the database collating sequence.

**BLOB**(*integer* [*K* | *M* | *G*])

For a binary large object string of the specified maximum length in bytes.

The length may be in the range of 1 byte to 2 147 483 647 bytes.

If *integer* by itself is specified, that is the maximum length.

If *integer K* (in either upper or lower case) is specified, the maximum length is 1 024 times *integer*. The maximum value for *integer* is 2 097 152.

If *integer M* is specified, the maximum length is 1 048 576 times *integer*. The maximum value for *integer* is 2 048.

If *integer G* is specified, the maximum length is 1 073 741 824 times *integer*. The maximum value for *integer* is 2.

To create BLOB strings greater than 1 gigabyte, you must specify the **NOT LOGGED** option.

## CREATE TABLE

Any number of spaces is allowed between the integer and K, M, or G. Also, no space is required. For example, all the following are valid.

BLOB(50K)    BLOB(50 K)    BLOB (50 K)

### **CLOB**(*integer* [*K* | *M* | *G*])<sup>61</sup>

For a character large object string of the specified maximum length in bytes.

The meaning of the *integer* *K* | *M* | *G* is the same as for BLOB.

To create CLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.

### **DBCLOB**(*integer* [*K* | *M* | *G*])

For a double-byte character large object string of the specified maximum length in double-byte characters.

The meaning of the *integer* *K* | *M* | *G* is similar to that for BLOB. The differences are that the number specified is the number of double-byte characters and that the maximum size is 1 073 741 823 double-byte characters.

To create DBCLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.

### **GRAPHIC**(*integer*)

For a fixed-length graphic string of length *integer* which may range from 1 to 127. If the length specification is omitted, a length of 1 is assumed.

### **VARGRAPHIC**(*integer*)

For a varying-length graphic string of maximum length *integer*, which may range from 1 to 2000.

### **LONG VARGRAPHIC**

For a varying-length graphic string with a maximum length of 16350.

### **DATE**

For a date.

### **TIME**

For a time.

### **TIMESTAMP**

For a timestamp.

### **DATALINK** or **DATALINK**(*integer*)

For a link to data stored outside the database.

The column in the table consists of "anchor values" that contain the refer-

---

<sup>61</sup> Observe that it is not possible to specify the FOR BIT DATA clause for CLOB columns. However, a CHAR FOR BIT DATA string can be assigned to a CLOB column and a CHAR FOR BIT DATA string can be concatenated with a CLOB string.

## CREATE TABLE

ence information that is required to establish and maintain the link to the external data as well as an optional comment.

The length of the column is 200. If the length specification is omitted, a length of 200 bytes is assumed.

A DATALINK value is an encapsulated value with a set of built-in scalar functions. There is a function called DLVALUE to create a DATALINK value. The following functions can be used to extract attributes from a DATALINK value.

- DLCOMMENT
- DLLINKTYPE
- DLURLCOMPLETE
- DLURLPATH
- DLURLPATHONLY
- DLURLSCHEME
- DLURLSERVER

A DATALINK column has the following restrictions:

- The column cannot be part of any index. Therefore, it cannot be included as a column of a primary key or unique constraint (SQLSTATE 42962).
- The column cannot be a foreign key of a referential constraint (SQLSTATE 42830).
- A default value (WITH DEFAULT) cannot be specified for the column. If the column is nullable, the default for the column is NULL (SQLSTATE 42894).

### *distinct-type-name*

For a user-defined type that is a distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL).

If a column is defined using a distinct type, then the data type of the column is the distinct type. The length and the scale of the column are respectively the length and the scale of the source type of the distinct type.

If a column defined using a distinct type is a foreign key of a referential constraint, then the data type of the corresponding column of the primary key must have the same distinct type.

### **REF** (*type-name2*)

For a reference to a typed table. If *type-name2* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The under-

## CREATE TABLE

lying data type of the column is based on the data type VARCHAR(16) FOR BIT DATA.

### *column-options*

Defines additional options related to columns of the table.

#### **NOT NULL**

Prevents the column from containing null values.

If NOT NULL is not specified, the column can contain null values, and its default value is either the null value or the value provided by the WITH DEFAULT clause.

### *default-clause*

Specifies a default value for the column.

#### **WITH**

An optional keyword.

#### **DEFAULT**

Provides a default value in the event a value is not supplied on INSERT or is specified as DEFAULT on INSERT or UPDATE. If a default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in Table 16 on page 385.

If a column is defined as a DATALINK, then a default value cannot be specified. The only possible default is NULL.

If the column is based on an attribute of a structured type, a specific default value must be specified when defining a default.

If a column is defined using a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column.

### *default-values*

Specific types of default values that can be specified are as follows.

#### *constant*

Specifies the constant as the default value for the column. The specified constant must:

- represent a value that could be assigned to the column in accordance with the rules of assignment as described in Chapter 3
- not be a floating-point constant unless the column is defined with a floating-point data type
- not have non-zero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column)

## CREATE TABLE

- be expressed with no more than 254 characters including the quote characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*.

### *datetime-special-register*

Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT or UPDATE as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified).

### **USER**

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default for the column. If USER is specified, the data type of the column must be a character string with a length not less than the length attribute of USER.

### **NULL**

Specifies NULL as the default for the column. If NOT NULL was specified, DEFAULT NULL may be specified within the same column definition but will result in an error on any attempt to set the column to the default value.

### *cast-function*

This form of a default value can only be used with columns defined as a distinct type, BLOB or datetime (DATE, TIME or TIMESTAMP) data type. For distinct type, with the exception of distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function may also be used. For a BLOB or a distinct type based on on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM. An example using the *cast-function* is given in Example 8 on page 397.

### *constant*

Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the

## CREATE TABLE

cast-function is BLOB, the constant must be a string constant.

### *datetime-special-register*

Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct type of the column must be the data type that corresponds to the specified special register.

### **USER**

Specifies the USER special register. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

If the value specified is not valid, an error (SQLSTATE 42894) is raised.

### *lob-options*

Specifies options for LOB data types.

### **LOGGED**

Specifies that changes made to the column are to be written to the log. The data in such columns is then recoverable with database utilities (such as RESTORE DATABASE). LOGGED is the default.

LOBs greater than 1 gigabyte cannot be logged (SQLSTATE 42993) and LOBs greater than 10 megabytes should probably not be logged.

### **NOT LOGGED**

Specifies that changes made to the column are not to be logged.

NOT LOGGED has no effect on a commit or rollback operation; that is, the database's consistency is maintained even if a transaction is rolled back, regardless of whether or not the LOB value is logged. The implication of not logging is that during a roll forward operation, after a backup or load operation, the LOB data will be replaced by zeros for those LOB values that would have had log records replayed during the roll forward. During crash recovery, all committed changes and changes rolled back will reflect the expected results. See the *Administration Guide* for the implications of not logging LOB columns.

### **COMPACT**

Specifies that the values in the LOB column should take up minimal disk space (free any extra disk pages in the last group used by the LOB value), rather than leave any left-over space at the end of the LOB storage area that might facilitate subsequent append operations. Note that storing data in this way may cause

## CREATE TABLE

a performance penalty in any append (length-increasing) operations on the column.

### **NOT COMPACT**

Specifies some space for insertions to assist in future changes to the LOB values in the column. This is the default.

### *datalink-options*

Specifies the options associated with a DATALINK data type.

### **LINKTYPE URL**

This defines the type of link as a Uniform Resource Locator (URL).

### **NO LINK CONTROL**

Specifies that there will not be any check made to determine that the file exists. Only the syntax of the URL will be checked. There is no database manager control over the file.

### **FILE LINK CONTROL**

Specifies that a check should be made for the existence of the file. Additional options may be used to give the database manager further control over the file.

### **file-link-options**

Additional options to define the level of database manager control of the file link.

### **INTEGRITY**

Specifies the level of integrity of the link between a DATALINK value and the actual file.

### **ALL**

Any file specified as a DATALINK value is under the control of the database manager and may NOT be deleted or renamed using standard file system programming interfaces.

### **READ PERMISSION**

Specifies how permission to read the file specified in a DATALINK value is determined.

**FS** The read access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.

**DB** The read access permission is determined by the database. Access to the file will only be allowed by passing a valid file access token, returned on retrieval of the DATALINK value from the table, in the open operation.

## CREATE TABLE

### **WRITE PERMISSION**

Specifies how permission to write to the file specified in a DATALINK value is determined.

**FS** The write access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.

### **BLOCKED**

Write access is blocked. The file cannot be directly updated through any interface. An alternative mechanism must be used to cause updates to the information. For example, the file is copied, the copy updated, and then the DATALINK value updated to point to the new copy of the file.

### **RECOVERY**

Specifies whether or not DB2 will support point in time recovery of files referenced by values in this column.

### **YES**

DB2 will support point in time recovery of files referenced by values in this column. This value can only be specified when INTEGRITY ALL and WRITE PERMISSION BLOCKED are also specified.

### **NO**

Specifies that point in time recovery will not be supported.

### **ON UNLINK**

Specifies the action taken on a file when a DATALINK value is changed or deleted (unlinked). Note that this is not applicable when WRITE PERMISSION FS is used.

### **RESTORE**

Specifies that when a file is unlinked, the DataLink File Manager will attempt to return the file to the owner with the permissions that existed at the time the file was linked. In the case where the user is no longer registered with the file server, the result is product-specific.<sup>62</sup> This can only be specified when INTEGRITY ALL and WRITE PERMISSION BLOCKED are also specified.

### **DELETE**

Specifies that the file will be deleted when it is unlinked. This can only be specified when READ PERMISSION

---

<sup>62</sup> With DB2 Universal Database, the file is assigned to a special predefined "dfmunknown" user id.



## CREATE TABLE

DB and WRITE PERMISSION BLOCKED are also specified.

### MODE DB2OPTIONS

This mode defines a set of default file link options. The defaults defined by DB2OPTIONS are:

- INTEGRITY ALL
- READ PERMISSION FS
- WRITE PERMISSION FS
- RECOVERY NO

ON UNLINK is not applicable since WRITE PERMISSION FS is used.

### SCOPE

Identifies the scope of the reference type column.

A scope must be specified for any column that is intended to be used as the left operand of a dereference operator or as the argument of the Deref function. Specifying the scope for a reference type column may be deferred to a subsequent ALTER TABLE statement to allow the target table to be defined, usually in the case of mutually referencing tables.

#### *typed-table-name*

The name of a typed table. The table must already exist or be the same as the name of the table being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of values assigned to *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

#### *typed-view-name*

The name of a typed view. The view must already exist or be the same as the name of the view being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of values assigned to *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

### CONSTRAINT *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that was already specified within the same CREATE TABLE statement. (SQLSTATE 42710).

## CREATE TABLE

If this clause is omitted, an 18-character identifier unique within the identifiers of the existing constraints defined on the table, is generated<sup>63</sup> by the system.

When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* may be used as the name of an index that is created to support the constraint.

### PRIMARY KEY

This 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.

A primary key cannot be specified if the table is a subtable (SQLSTATE 429B3) since the primary key is inherited from the supertable.

See PRIMARY KEY within the description of the *unique-constraint* below.

### UNIQUE

This 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.

A unique constraint cannot be specified if the table is a subtable (SQLSTATE 429B3) since unique constraints are inherited from the supertable.

See UNIQUE within the description of the *unique-constraint* below.

### *references-clause*

This 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.

See *references-clause* under *referential-constraint* below.

### CHECK (*check-condition*)

This provides a shorthand method of defining a check constraint that applies to a single column. See CHECK (*check-condition*) below.

### *unique-constraint*

Defines a unique or primary key constraint. If the table has a partitioning key, then any unique or primary key must be a superset of the partitioning key. A unique or primary key constraint cannot be specified for a table

<sup>63</sup> The identifier is formed of "SQL" followed by a sequence of 15 numeric characters generated by a timestamp-based function.

that is a subtable (SQLSTATE 429B3). If the table is a root table, the constraint applies to the table and all its subtables.

**CONSTRAINT** *constraint-name*

Names the primary key or unique constraint. See page 539.

**UNIQUE** (*column-name,...*)

Defines a unique key composed of the identified columns. The identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the table and the same column must not be identified more than once. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255. No LOB, LONG VARCHAR, LONG VARGRAPHIC, or DATALINK column may be used as part of a unique key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962). The set of columns in the unique key cannot be the same as the set of columns of the primary key or another unique key (SQLSTATE 01543).<sup>64</sup>

A unique constraint cannot be specified if the table is a subtable (SQLSTATE 429B3) since unique constraints are inherited from the supertable.

The description of the table as recorded in the catalog includes the unique key and its unique index. A unique index will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index will be the same as the *constraint-name* if this does not conflict with an existing index in the schema where the table is created. If the index name conflicts, the name will be SQL, followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

**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 identify a column of the table and the same column must not be identified more than once. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255. No LOB, LONG VARCHAR, LONG VARGRAPHIC, or DATALINK column may be used as part of a primary key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962). The set of columns in the primary key cannot be the same as the set of columns of a unique key (SQLSTATE 01543).<sup>64</sup>

Only one primary key can be defined on a table.

<sup>64</sup> If LANGLEVEL is SQL92E or MIA then an error is returned, SQLSTATE 42891.

## CREATE TABLE

A primary key cannot be specified if the table is a subtable (SQLSTATE 429B3) since the primary key is inherited from the super-table.

The description of the table as recorded in the catalog includes the primary key and its primary index. A unique index will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index will be the same as the *constraint-name* if this does not conflict with an existing index in the schema where the table is created. If the index name conflicts, the name will be SQL, followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

If the table has a partitioning key, the columns of a *unique-constraint* must be a superset of the partitioning key columns; column order is unimportant.

### *referential-constraint*

Defines a referential constraint. A referential constraint cannot be defined from a typed table or to a parent table that is a typed table (SQLSTATE 42997).

### **CONSTRAINT** *constraint-name*

Names the referential constraint. See page 539.

### **FOREIGN KEY** (*column-name,...*)

Defines a referential constraint with the specified *constraint-name*.

Let T1 denote the object table of the statement. The foreign key of the referential constraint is composed of the identified columns. Each name in the list of column names must identify a column of T1 and the same column must not be identified more than once. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255 minus the number of columns that allow null values. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may be used as part of a foreign key (SQLSTATE 42962). There must be the same number of foreign key columns as there are in the parent key and the data types of the corresponding columns must be compatible (SQLSTATE 42830). Two column descriptions are compatible if they have compatible data types (both columns are numeric, character strings, graphic, date/time, or have the same distinct type).

### *references-clause*

Specifies the parent table and parent key for the referential constraint.

### **REFERENCES** *table-name*

The table specified in a REFERENCES clause must identify a base table that is described in the catalog, but must not identify a catalog table.

## CREATE TABLE

A referential constraint is a duplicate if its foreign key, parent key, and parent table are the same as the foreign key, parent key and parent table of a previously specified referential constraint. Duplicate referential constraints are ignored and a warning is issued (SQLSTATE 01543).

In the following discussion, let T2 denote the identified parent table and let T1 denote the table being created<sup>65</sup> (T1 and T2 may be the same table).

The specified foreign key must have the same number of columns as the parent key of T2 and the description of the *n*th column of the foreign key must be comparable to the description of the *n*th column of that parent key. Datetime columns are not considered to be comparable to string columns for the purposes of this rule.

*(column-name,...)*

The parent key of a referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than once.

The list of column names must match the set of columns (in any order) of the primary key or a unique constraint that exists on T2 (SQLSTATE 42890). If a column name list is not specified, then T2 must have a primary key (SQLSTATE 42888). Omission of the column name list is an implicit specification of the columns of that primary key in the sequence originally specified.

The referential constraint specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

*rule-clause*

Specifies what action to take on dependent tables.

### ON DELETE

Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are four possible actions:

- NO ACTION (default)
- RESTRICT
- CASCADE
- SET NULL

---

<sup>65</sup> or altered, in the case where this clause is referenced from the description of the ALTER TABLE statement.

## CREATE TABLE

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.

SET NULL must not be specified unless some column of the foreign key allows null values. Omission of the clause is an implicit specification of ON DELETE NO ACTION.

A cycle involving two or more tables must not cause a table to be delete-connected to itself unless all of the delete rules in the cycle are CASCADE. Thus, if the new relationship would form a cycle and T2 is already delete connected to T1, then the constraint can only be defined if it has a delete rule of CASCADE and all other delete rules of the cycle are CASCADE.

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. The NO ACTION and RESTRICT actions are treated identically. Thus, if T1 is a dependent of T3 in a relationship with a delete rule of  $r$ , the referential constraint cannot be defined when  $r$  is SET NULL if any of these conditions exist:

- T2 and T3 are the same table
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3
- T2 and T3 are both descendants of the same table and the deletion of rows from that table cascades to both T2 and T3.

If  $r$  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  $r$ .

### ON UPDATE

Specifies what action is to take place on the dependent tables when a row of the parent table is updated. The clause

## CREATE TABLE

is optional. ON UPDATE NO ACTION is the default and ON UPDATE RESTRICT is the only alternative.

The difference between NO ACTION and RESTRICT is described under CREATE TABLE in "Notes" on page 550.

### *check-constraint*

Defines a check constraint. A *check-constraint* is a *search-condition* that must evaluate to not false. A check constraint cannot be defined on a typed table (SQLSTATE 42997).

### **CONSTRAINT** *constraint-name*

Names the check constraint. See page 539.

### **CHECK** (*check-condition*)

Defines a check constraint. A *check-condition* is a *search-condition* except as follows:

- A column reference must be to a column of the table being created
- It cannot contain any of the following:
  - subqueries
  - dereference operations or Deref functions where the reference value is other than the object identifier column
  - column functions
  - variant user-defined functions
  - user-defined functions using the EXTERNAL ACTION option
  - user-defined functions using the SCRATCHPAD option
  - host variables
  - parameter markers
  - special registers
  - an alias

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

Check constraints are not checked for inconsistencies, duplicate conditions or equivalent conditions. Therefore, contradictory or redundant check constraints can be defined resulting in possible errors at execution time.

The check-condition "IS NOT NULL" can be specified, however it is recommended that nullability be enforced directly using the NOT NULL attribute of a column. For example, CHECK (salary + bonus > 30000) is accepted if salary is set to NULL, because CHECK constraints must be either satisfied or unknown and in this case salary is unknown. However, CHECK (salary IS NOT NULL) would be considered false and a violation of the constraint if salary is set to NULL.

## CREATE TABLE

Check constraints are enforced when rows in the table are inserted or updated. A check constraint defined on a table automatically applies to all subtables of that table.

### *OID-column-definition*

Defines the object identifier column for the typed table.

### **REF IS *OID-column-name* USER GENERATED**

Specifies that an object identifier (OID) column is defined in the table as the first column. An OID is required for the root table of a table hierarchy (SQLSTATE 428DX). The table must be a typed table (the OF clause must be present) that is not a subtable (SQLSTATE 42613). The name for the column is defined as *OID-column-name* and cannot be the same as the name of any attribute of the structured type *type-name1* (SQLSTATE 42711). The column is defined with type REF(*type-name1*), NOT NULL and a system required unique index (with a default index name) is generated. This column is referred to as the *object identifier column* or *OID column*. The keywords USER GENERATED indicate that the initial value for the OID column must be provided by the user when inserting a row. Once a row is inserted, the OID column cannot be updated (SQLSTATE 42808).

### *with-options*

Defines additional options that apply to columns of a typed table.

### *column-name*

Specifies the name of the column for which additional options are specified. The *column-name* must correspond to the name of a column of the table that is not also a column of a supertable (SQLSTATE 428DJ). A column name can only appear in one WITH OPTIONS clause in the statement (SQLSTATE 42613).

If an option is already specified as part of the type definition (in CREATE TYPE), the options specified here override the options in CREATE TYPE.

### **WITH OPTIONS *column-options***

Defines options for the specified column. See *column-options* described earlier. If the table is a subtable, primary key or unique constraints cannot be specified (SQLSTATE 429B3).

### **DATA CAPTURE**

Indicates whether extra information for inter-database data replication is to be written to the log. This clause cannot be specified when creating a subtable (SQLSTATE 42613).

### **NONE**

Indicates that no extra information will be logged.



## CREATE TABLE

### CHANGES

Indicates that extra information regarding SQL changes to this table will be written to the log. This option is required if this table will be replicated and the Capture program is used to capture changes for this table from the log.

If the table is defined to allow data on a partition other than the catalog partition (multiple partition nodegroup or nodegroup with a partition other than the catalog partition), then this option is not supported (SQLSTATE 42997).

Further information about using replication can be found in the *Administration Guide* and the *DB2 Replication Guide and Reference*.

### IN *tablespace-name1*

Identifies the table space in which the table will be created. The table space must exist, and be a REGULAR table space. If no other table space is specified, then all table parts will be stored in this table space. This clause cannot be specified when creating a subtable (SQLSTATE 42613), since the table space is inherited from the root table of the table hierarchy. If this clause is not specified, a table space for the table is determined as follows:

```
IF table space IBMDEFAULTGROUP exists with sufficient page size
  THEN use it
ELSE IF user created table space exists with sufficient page size
  THEN use it
ELSE IF table space USERSPACE1 exists with sufficient page size
  THEN use it
ELSE IF USERSPACE8K exists with sufficient page size
  THEN use it
ELSE issue an error (SQLSTATE 42727).
```

The sufficient page size of a table is determined by either the byte count of the row or the number of columns. Refer to Row Size on page 553 for more information. If the table includes one or more LOB columns, then only a 4K page size table space can be used. If the number of columns or the row size would require an 8K page size table space, then a default cannot be select and an error is returned (SQLSTATE 42997). In this situation an 8K page size DMS table space should be specified with a 4K page size DMS table space in the LONG IN clause.

### *tablespace-options:*

Specifies the table space in which indexes and/or long column values will be stored. See “CREATE TABLESPACE” on page 559 for details on types of table spaces.

### INDEX IN *tablespace-name2*

Identifies the table space in which any indexes on the table will be created. This option is allowed only when the primary table space specified in the IN clause is a DMS table space. The specified table

## CREATE TABLE

space must exist, be a REGULAR DMS table space and must be in the same nodegroup as *tablespace-name1* (SQLSTATE 42838) .

Note that specifying which table space will contain a table's index can only be done when the table is created.

### **LONG IN** *tablespace-name3*

Identifies the table space in which the values of any long columns (LONG VARCHAR, LONG VARGRAPHIC, LOB data types, or distinct types with any of these as source types) will be stored. This option is allowed only when the primary table space specified in the IN clause is a DMS table space. The table space must exist, be a LONG DMS table space and must be in the same nodegroup of *tablespace-name1* (SQLSTATE 42838) .

### **PARTITIONING KEY** (*column-name,...*)

Specifies the partitioning key used when data in the table is partitioned. Each *column-name* must identify a column of the table and the same column must not be identified more than once. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may be used as part of a partitioning key (SQLSTATE 42962). A partitioning key cannot be specified for a table that is a subtable (SQLSTATE 42613), since the partitioning key is inherited from the root table in the table hierarchy.

If this clause is not specified, and this table resides in a multiple partition nodegroup, then the partitioning key is defined as follows:

- if the table is a typed table, the object identifier column
- if a primary key is specified, the first column of the primary key is the partitioning key
- otherwise, the first non-long column (LOB or long column type) is the partitioning key.

If none of the columns satisfy the requirement of the default partitioning key, the table is created without one. Such tables are allowed only in table spaces defined on single-partition nodegroups.

For tables in table spaces defined on single-partition nodegroups, any collection of non-long type columns can be used to define the partitioning key. If you do not specify this parameter, no partitioning key is created.

For restrictions related to the partitioning key, see "Rules" on page 549.

### **USING HASHING**

Specifies the use of the hashing function as the partitioning method for data distribution. This is the only partitioning method supported.

### **REPLICATED**

Specifies that the data stored in the table is physically replicated on each database partition of the nodegroup of the table space in which the table is defined. This means that a copy of all the data in the table exists on each of these database partitions. This option can only be specified for a

## CREATE TABLE

summary table defined with REFRESH IMMEDIATE (SQLSTATE 42997). A unique index must exist on the underlying table where the index key columns are included in the select list of the fullselect that defines the summary table.

### NOT LOGGED INITIALLY

Any changes made to the table by an Insert, Delete, Update, Create Index, Drop Index, or Alter Table operation in the same unit of work in which the table is created are not logged. See “Notes” on page 550 for other considerations when using this option.

All catalog changes and storage related information are logged, as are all operations that are done on the table in subsequent units of work.

A foreign key constraint cannot be defined on a table that references a parent with the NOT LOGGED INITIALLY attribute. This clause cannot be specified when creating a subtable (SQLSTATE 42613).

**Note:** An error in any operation in the unit of work in which the table is created will result in the entire unit of work being rolled back.

## Rules

- The sum of the byte counts of the columns must not be greater than the row size limit that is based on the page size of the table space (SQLSTATE 54010). Refer to Byte Counts on page 553 and Table 27 on page 755 for more information. For typed tables, the byte count is applied to the columns of the root table of the table hierarchy and every additional column introduced by every subtable in the table hierarchy (additional subtable columns must be considered nullable for byte count purposes, even when defined as not nullable). There is also an additional 4 bytes of overhead to identify the subtable to which each row belongs.
- The number of columns in a table cannot exceed 1012 (SQLSTATE 54011). For typed tables, the total number of attributes of the types of all of the subtables in the table hierarchy cannot exceed 1010.
- An object identifier column of a typed table cannot be updated (SQLSTATE 42808).
- A partitioning key column of a table cannot be updated (SQLSTATE 42997).
- Any unique or primary key constraint defined on the table must be a superset of the partitioning key (SQLSTATE 42997).
- A nullable column of a partitioning key cannot be included as a foreign key column when the relationship is defined with ON DELETE SET NULL (SQLSTATE 42997).
- The following table provides the supported combinations of DATALINK options in the *file-link-options* (SQLSTATE 42613).

## CREATE TABLE

Table 18. Valid DATALINK File Control Option Combinations

INTEGRITY	READ PER- MISSION	WRITE PER- MISSION	RECOVERY	ON UNLINK
ALL	FS	FS	NO	Not applicable
ALL	FS	BLOCKED	NO	RESTORE
ALL	FS	BLOCKED	YES	RESTORE
ALL	DB	BLOCKED	NO	RESTORE
ALL	DB	BLOCKED	NO	DELETE
ALL	DB	BLOCKED	YES	RESTORE
ALL	DB	BLOCKED	YES	DELETE

The following rules only apply to partitioned databases.

- Tables composed of only long columns can only be created in table spaces defined on single-partition nodegroups.
- The partitioning key definition of a table in a table space defined on a multiple partition nodegroup cannot be altered.
- The partitioning key column of a typed table must be the OID column.

### Notes

- Creating a table with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- If a foreign key is specified:
  - All packages with a delete usage on the parent table are invalidated.
  - All packages with an update usage on at least one column in the parent key are invalidated.
- Creating a subtable causes invalidation of all packages that depend on any table in table hierarchy.
- The use of NO ACTION or RESTRICT as delete or update rules for referential constraints determines when the constraint is enforced. A delete or update rule of RESTRICT is enforced before all other constraints including those referential constraints with modifying rules such as CASCADE or SET NULL. A delete or update rule of NO ACTION is enforced after other referential constraints. There are very few cases where this can make a difference during a delete or update. One example where different behavior is evident involves a DELETE of rows in a view that is defined as a UNION ALL of related tables.

## CREATE TABLE

Table T1 is a parent of table T3, delete rule as noted below  
Table T2 is a parent of table T3, delete rule CASCADE

```
CREATE VIEW V1 AS SELECT * FROM T1 UNION ALL SELECT * FROM T2
```

```
DELETE FROM V1
```

If table T1 is a parent of table T3 with delete rule of RESTRICT, a restrict violation will be raised (SQLSTATE 23001) if there are any child rows for parent keys of T1 in T3.

If table T1 is a parent of table T3 with delete rule of NO ACTION, the child rows may be deleted by the delete rule of CASCADE when deleting rows from T2 before the NO ACTION delete rule is enforced for the deletes from T1. If deletes from T2 did not result in deleting all child rows for parent keys of T1 in T3, then a constraint violation will be raised (SQLSTATE 23504).

Note that the SQLSTATE returned is different depending on whether the delete or update rule is RESTRICT or NO ACTION.

- For tables in table spaces defined on multiple partition nodegroups, table collocation should be considered in choosing the partitioning keys. Following is a list of items to consider:
  - The tables must be in the same nodegroup for collocation. The table spaces may be different, but must be defined in the same nodegroup.
  - The partitioning keys of the tables must have the same number of columns, and the corresponding key columns must be partition compatible for collocation. For more information, see “Partition Compatibility” on page 87.
  - The choice of partitioning key also has an impact on performance of joins. If a table is frequently joined with another table, you should consider the joining column(s) as a partitioning key for both tables.
- The NOT LOGGED INITIALLY option is useful for situations where a large result set needs to be created with data from an alternate source (another table or a file) and recovery of the table is not necessary. Using this option will save the overhead of logging the data. The following considerations apply when this option is specified:
  - When the unit of work is committed, all changes that were made to the table during the unit of work are flushed to disk.
  - When you run the Rollforward utility and it encounters a log record that indicates that a table in the database was either populated by the Load utility or created with the NOT LOGGED INITIALLY option, the table will be marked as unavailable. The table will be dropped by the Rollforward utility if it later encounters a DROP TABLE log. Otherwise, after the database is recovered, an error will be issued if any attempt is made to access the table (SQLSTATE 55019). The only operation permitted is to drop the table.
  - Once such a table is backed up as part of a database or table space back up, recovery of the table becomes possible.

## CREATE TABLE

- A REFRESH DEFERRED summary table may be used to optimize the processing queries. In order for this optimization be able to use a summary table, the fullselect must conform to certain rules in addition to those already described. The fullselect must:
  - be a subselect with a GROUP BY clause
  - not include DISTINCT anywhere in the select list
  - not include any grouping sets (including CUBE or ROLLUP)
  - not include any special registers
  - not include functions that are not deterministic.

If the query specified when creating a REFRESH DEFERRED summary table does not conform to these rules, a warning is returned (SQLSTATE 01633).

- **Inoperative summary tables:** An inoperative summary table is a table that is no longer available for SQL statements. A summary table becomes inoperative if:
  - A privilege upon which the summary table definition is dependent is revoked.
  - An object such as a table, alias or function, upon which the summary table definition is dependent is dropped.

In practical terms, an inoperative summary table is one in which the summary table definition has been unintentionally dropped. For example, when an alias is dropped, any summary table defined using that alias is made inoperative. All packages dependent on the summary table are no longer valid.

Until the inoperative summary table is explicitly recreated or dropped, a statement using that inoperative summary table cannot be compiled (SQLSTATE 51024) with the exception of the CREATE ALIAS, CREATE TABLE, DROP TABLE, and COMMENT ON TABLE statements. Until the inoperative summary table has been explicitly dropped, its qualified name cannot be used to create another view, base table or alias. (SQLSTATE 42710).

An inoperative summary table may be recreated by issuing a CREATE TABLE statement using the definition text of the inoperative summary table. This summary table query text is stored in the TEXT column of the SYSCAT.VIEWS catalog. When recreating an inoperative summary table, it is necessary to explicitly grant any privileges required on that table by others, due to the fact that all authorization records on a summary table are deleted if the summary table is marked inoperative. Note that there is no need to explicitly drop the inoperative summary table in order to recreate it. Issuing a CREATE TABLE statement that defines a summary table with the same *table-name* as an inoperative summary table will cause that inoperative summary table to be replaced, and the CREATE TABLE statement will return a warning (SQLSTATE 01595).

Inoperative summary tables are indicated by an X in the VALID column of the SYSCAT.VIEWS catalog view and an X in the STATUS column of the SYSCAT.TABLES catalog view.

- **Privileges:** When any table is created, the definer of the table is granted CONTROL privilege. When a subtable is created, the SELECT privilege that each

## CREATE TABLE

user or group has on the the immediate supertable is automatically granted on the subtable with the table definer as the grantor.

- **Row Size:** The maximum number of bytes allowed in the row of a table is dependent on the page size of the the table space in which the table is created (*tablespace-name1*). The following list shows the row size limit and number of columns limit associated with each table space page size.

Table 19. Limits for Number of Columns and Row Size in Each Table Space Page Size

Page Size	Row Size Limit	Column Count Limit
4K	4005	500
8K	8101	1012

- **Byte Counts:** The following list contains the byte counts of columns by data type for columns that do not allow null values. For a column that allows null values the byte count is one more than shown in the list.

If the table is created based on a structured type, an additional 4 bytes of overhead is reserved to identify rows of subtables regardless of whether or not subtables are defined. Also, additional subtable columns must be considered nullable for byte count purposes, even when defined as not nullable.

Data type	Byte count
INTEGER	4
SMALLINT	2
BIGINT	8
REAL	4
DOUBLE	8
DECIMAL	The integral part of $(p/2)+1$ , where $p$ is the precision.
CHAR( $n$ )	$n$
VARCHAR( $n$ )	$n+4$
LONG VARCHAR	24
GRAPHIC( $n$ )	$n*2$
VARGRAPHIC( $n$ )	$(n*2)+4$
LONG VARGRAPHIC	24
DATE	4
TIME	3
TIMESTAMP	10
DATALINK( $n$ )	$n+54$

## CREATE TABLE

LOB types

Each LOB value has a *LOB descriptor* in the base record that points to the location of the actual value. The size of the descriptor varies according to the maximum length defined for the column. The following table shows typical sizes:

Maximum LOB Length	LOB Descriptor Size
1 024	72
8 192	96
65 536	120
524 000	144
4 190 000	168
134 000 000	200
536 000 000	224
1 070 000 000	256
1 470 000 000	280
2 147 483 647	316

Distinct type

Length of the source type of the distinct type.

Reference type

Length of the built-in data type on which the reference type is based.

## Examples

*Example 1:* Create table TDEPT in the DEPARTX table space. DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT are column names. CHAR means the column will contain character data. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. The primary key consists of the column DEPTNO.

```
CREATE TABLE TDEPT
  (DEPTNO CHAR(3) NOT NULL,
   DEPTNAME VARCHAR(36) NOT NULL,
   MGRNO CHAR(6),
   ADMRDEPT CHAR(3) NOT NULL,
   PRIMARY KEY(DEPTNO))
IN DEPARTX
```

*Example 2:* Create table PROJ in the SCHED table space. PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF, PRSTDATE, PRENDATE, and MAJPROJ are column names. CHAR means the column will contain character data. DECIMAL means the column will contain packed decimal data. 5,2 means the following: 5 indicates the number of decimal digits, and 2 indicates the number of digits to the right of the decimal point. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. DATE means the column will contain date information in a three-part format (year, month, and day).



## CREATE TABLE

```
CREATE TABLE 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 SCHED
```

*Example 3:* Create a table called EMPLOYEE\_SALARY where any unknown salary is considered 0. No table space is specified, so that the table will be created in a table space selected by the system based on the rules described for the *IN tablespace-name1* clause.

```
CREATE TABLE EMPLOYEE_SALARY
  (DEPTNO CHAR(3) NOT NULL,
   DEPTNAME VARCHAR(36) NOT NULL,
   EMPNO CHAR(6) NOT NULL,
   SALARY DECIMAL(9,2) NOT NULL WITH DEFAULT)
```

*Example 4:* Create distinct types for total salary and miles and use them for columns of a table created in the default table space. In a dynamic SQL statement assume the CURRENT SCHEMA special register is JOHNDOE and the CURRENT PATH is the default ("SYSIBM","SYSFUN","JOHNDOE").

If a value for SALARY is not specified it must be set to 0 and if a value for LIVING\_DIST is not specified it must be set to 1 mile.

```
CREATE DISTINCT TYPE JOHNDOE.T_SALARY AS INTEGER WITH COMPARISONS
```

```
CREATE DISTINCT TYPE JOHNDOE.MILES AS FLOAT WITH COMPARISONS
```

```
CREATE TABLE EMPLOYEE
  (ID INTEGER NOT NULL,
   NAME CHAR(30),
   SALARY T_SALARY NOT NULL WITH DEFAULT,
   LIVING_DIST MILES DEFAULT MILES(1) )
```

*Example 5:* Create distinct types for image and audio and use them for columns of a table. No table space is specified, so that the table will be created in a table space selected by the system based on the rules described for the *IN tablespace-name1* clause. Assume the CURRENT PATH is the default.

## CREATE TABLE

```
CREATE DISTINCT TYPE IMAGE AS BLOB (10M)
```

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1G)
```

```
CREATE TABLE PERSON
(SSN    INTEGER NOT NULL,
 NAME   CHAR (30),
 VOICE  AUDIO,
 PHOTO  IMAGE)
```

*Example 6:* Create table EMPLOYEE in the HUMRES table space. The constraints defined on the table are the following:

- The values of department number must lie in the range 10 to 100.
- The job of an employee can only be either 'Sales', 'Mgr' or 'Clerk'.
- Every employee that has been with the company since 1986 must make more than \$40,500.

**Note:** If the columns included in the check constraints are nullable they could also be NULL.

```
CREATE TABLE EMPLOYEE
(ID          SMALLINT NOT NULL,
 NAME       VARCHAR(9),
 DEPT       SMALLINT CHECK (DEPT BETWEEN 10 AND 100),
 JOB        CHAR(5) CHECK (JOB IN ('Sales', 'Mgr', 'Clerk')),
 HIREDATE   DATE,
 SALARY     DECIMAL(7,2),
 COMM       DECIMAL(7,2),
 PRIMARY KEY (ID),
 CONSTRAINT YEARSAL CHECK (YEAR(HIREDATE) > 1986 OR SALARY > 40500)
)
IN HUMRES
```

*Example 7:* Create a table that is wholly contained in the PAYROLL table space.

```
CREATE TABLE EMPLOYEE .....
IN PAYROLL
```

*Example 8:* Create a table with its data part in ACCOUNTING and its index part in ACCOUNT\_IDX.

```
CREATE TABLE SALARY.....
IN ACCOUNTING INDEX IN ACCOUNT_IDX
```

*Example 9:* Create a table and log SQL changes in the default format.

```
CREATE TABLE SALARY1 .....
```

or

```
CREATE TABLE SALARY1 .....
```

```
DATA CAPTURE NONE
```

*Example 10:* Create a table and log SQL changes in an expanded format.

## CREATE TABLE

```
CREATE TABLE SALARY2 .....  
DATA CAPTURE CHANGES
```

*Example 11:* Create a table EMP\_ACT in the SCHED table space. EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, and EMENDATE are column names. Constraints defined on the table are:

- The value for the set of columns, EMPNO, PROJNO, and ACTNO, in any row must be unique.
- The value of PROJNO must match an existing value for the PROJNO column in the PROJECT table and if the project is deleted all rows referring to the project in EMP\_ACT should also be deleted.

```
CREATE TABLE EMP_ACT  
(EMPNO      CHAR(6) NOT NULL,  
  PROJNO    CHAR(6) NOT NULL,  
  ACTNO     SMALLINT NOT NULL,  
  EMPTIME   DECIMAL(5,2),  
  EMSTDATE  DATE,  
  EMENDATE  DATE,  
  CONSTRAINT EMP_ACT_UNIQ UNIQUE (EMPNO,PROJNO,ACTNO),  
  CONSTRAINT FK_ACT_PROJ FOREIGN KEY (PROJNO)  
                        REFERENCES PROJECT (PROJNO) ON DELETE CASCADE  
)  
IN SCHED
```

A unique index called EMP\_ACT\_UNIQ is automatically created in the same schema to enforce the unique constraint.

*Example 12:* Create a table that is to hold information about famous goals for the ice hockey hall of fame. The table will list information about the player who scored the goal, the goaltender against who it was scored, the date and place, and a description. When available, it will also point to places where newspaper articles about the game are stored and where still and moving pictures of the goal are stored. The newspaper articles are to be linked so they cannot be deleted or renamed but all existing display and update applications must continue to operate. The still pictures and movies are to be linked with access under complete control of DB2. The still pictures are to have recovery and are to be returned to their original owner if unlinked. The movie pictures are not to have recovery and are to be deleted if unlinked. The description column and the three DATALINK columns are nullable.

## CREATE TABLE

```
CREATE TABLE HOCKEY_GOALS
( BY_PLAYER      VARCHAR(30)  NOT NULL,
  BY_TEAM        VARCHAR(30)  NOT NULL,
  AGAINST_PLAYER VARCHAR(30)  NOT NULL,
  AGAINST_TEAM   VARCHAR(30)  NOT NULL,
  DATE_OF_GOAL   DATE         NOT NULL,
  DESCRIPTION     CLOB(5000),
  ARTICLES        DATALINK    LINKTYPE URL FILE LINK CONTROL MODE DB2OPTIONS,
  SNAPSHOT        DATALINK    LINKTYPE URL FILE LINK CONTROL
                                INTEGRITY ALL
                                READ PERMISSION DB WRITE PERMISSION BLOCKED
                                RECOVERY YES ON UNLINK RESTORE,
  MOVIE           DATALINK    LINKTYPE URL FILE LINK CONTROL
                                INTEGRITY ALL
                                READ PERMISSION DB WRITE PERMISSION BLOCKED
                                RECOVERY NO ON UNLINK DELETE )
```

---

### CREATE TABLESPACE

The CREATE TABLESPACE statement creates a new table space within the database, assigns containers to the table space, and records the table space definition and attributes in the catalog.

#### Invocation

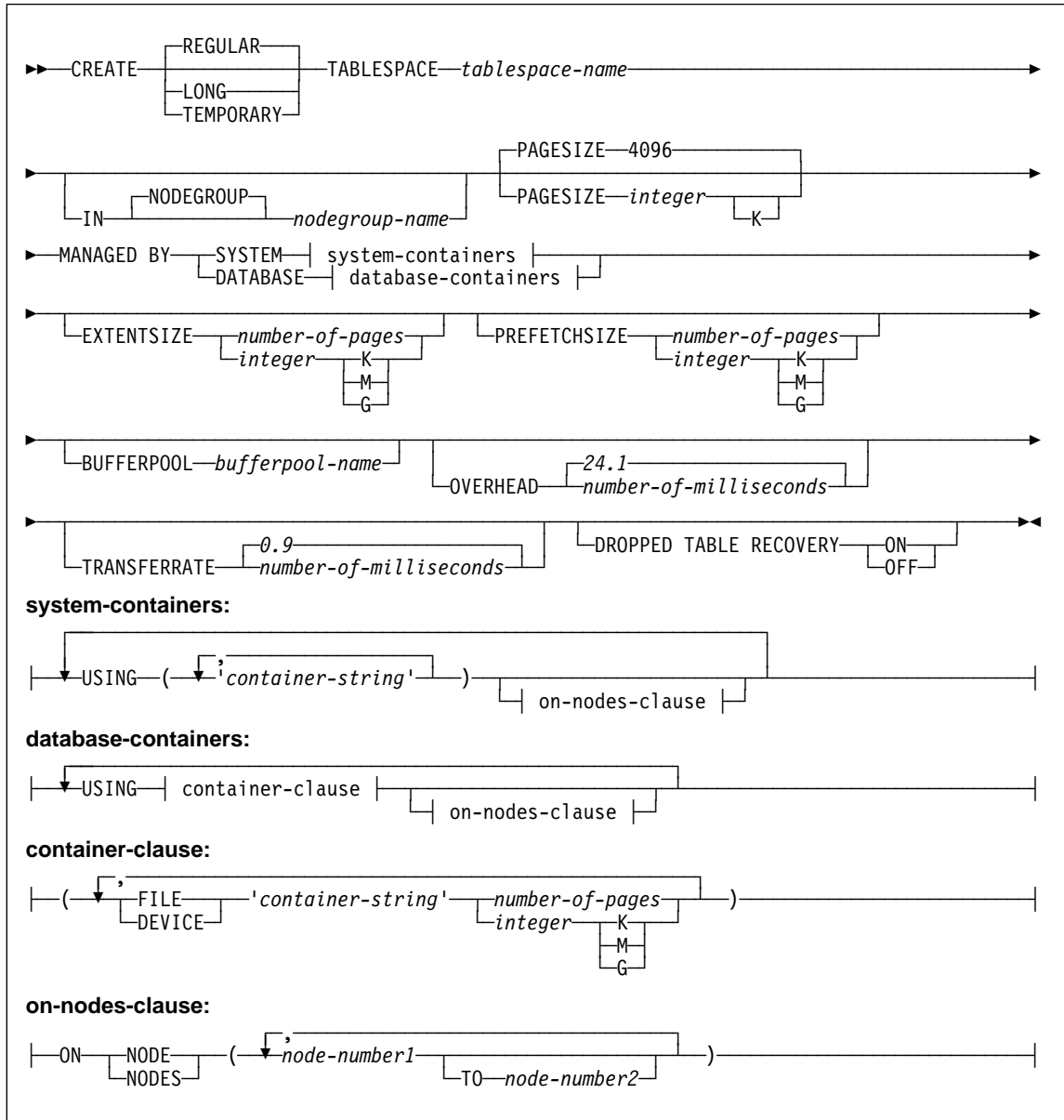
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

#### Authorization

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

#### Syntax

# CREATE TABLESPACE



## Description

### REGULAR

Stores all data except for temporary tables.

## CREATE TABLESPACE

### LONG

Stores long or LOB table columns. The table space must be a DMS table space.

### TEMPORARY

Stores temporary tables. (Temporary tables are work areas used by the database manager to perform operations such as sorts or joins.) Note that a database must always have at least one TEMPORARY table space, as temporary tables can only be stored in such a table space. A temporary table space is created automatically when a database is created.

(See CREATE DATABASE in the *Command Reference*.)

### *tablespace-name*

Names the table space. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *tablespace-name* must not identify a table space that already exists in the catalog (SQLSTATE 42710). The *tablespace-name* must not begin with the characters SYS (SQLSTATE 42939).

### IN NODEGROUP *nodegroup-name*

Specifies the nodegroup for the table space. The nodegroup must exist. The only nodegroup that can be specified when creating a TEMPORARY table space is IBMTEMPGROUP. The NODEGROUP keyword is optional.

If the nodegroup is not specified, the default nodegroup (IBMDEFAULTGROUP) is used unless TEMPORARY is specified and then IBMTEMPGROUP is used.

### PAGESIZE *integer* [K]

Defines the size of pages used for the table space. The valid values for *integer* without the suffix K are 4096 or 8192. The valid values for *integer* with the suffix K are 4 or 8. An error occurs if the page size is not one of these values (SQLSTATE 428DE) or the page size is not the same as the page size of the bufferpool associated with the table space (SQLSTATE 428CB). If a LONG TABLESPACE is being defined, the specified page size must be 4096 or 4K (SQLSTATE 42997). The default is 4096 byte (4K) pages. Any number of spaces is allowed between *integer* and K, including no space.

### MANAGED BY SYSTEM

Specifies that the table space is to be a system managed space (SMS) table space.

### system-containers

Specify the containers for an SMS table space.

### USING ('*container-string*',...)

For a SMS table space, identifies one or more containers that will belong to the table space and into which the table space's data will be stored. The *container-string* cannot exceed 240 bytes in length.

Each *container-string* can be an absolute or relative directory name. The directory name, if not absolute, is relative to the database directory. If any component of the directory name does not exist, it is created by the database manager. When a table space is dropped, all components created by the data-

## CREATE TABLESPACE

base manager are deleted. If the directory identified by *container-string* exist, it must not contain any files or subdirectories (SQLSTATE 428B2).

The format of *container-string* is dependent on the operating system. The containers are specified in the normal manner for the operating system. For example, an OS/2 Windows 95 and Windows NT directory path begins with a drive letter and a “:”, while on UNIX-based systems, a path begins with a “/”.

Note that remote resources (such as LAN-redirected drives on OS/2, Windows 95 and Windows NT or NFS-mounted file systems on AIX) are not supported.

### *on-nodes-clause*

Specifies the partition or partitions on which the containers are created in a partitioned database . If this clause or any other *on-nodes-clause* of this statement is not specified, then the containers are created on all partitions or nodes currently in the nodegroup. For a TEMPORARY table space when the clause is not specified, the containers will also be created on all new partitions or nodes added to the database. See page 563 for details on specifying this clause.

### MANAGED BY DATABASE

Specifies that the table space is to be a database managed space (DMS) table space.

### database-containers

Specify the containers for a DMS table space.

### USING

Introduces a container-clause.

### *container-clause*

Specifies the containers for a DMS table space.

### (FILE|DEVICE '*container-string*' *number-of-pages*,...)

For a DMS table space, identifies one or more containers that will belong to the table space and into which the table space's data will be stored. The type of the container (either FILE or DEVICE) and its size (in PAGESIZE pages) are specified. The size can also be specified as an integer value followed by K (for kilobytes), M (for megabytes) or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages for the container. A mixture of FILE and DEVICE containers can be specified. The *container-string* cannot exceed 254 bytes in length.

For a FILE container, the *container-string* must be an absolute or relative file name. The file name, if not absolute, is relative to the database directory. If any component of the directory name does not exist, it is created by the database manager. If the file does not exist, it will be created and initialized to the specified size by the database manager. When a table space is dropped, all components created by the database manager are deleted.



## CREATE TABLESPACE

**Note:** If the file exists it is overwritten and if it is smaller than specified it is extended. The file will not be truncated if it is larger than specified.

For a DEVICE container, the *container-string* must be a device name. The device must already exist.

All containers must be unique across all databases; a container can belong to only one table space. The size of the containers can differ, however optimal performance is achieved when all containers are the same size. The exact format of *container-string* is dependent on the operating system. The containers will be specified in the normal manner for the operating system. For more detail on declaring containers, refer to the *Administration Guide*.

Remote resources (such as LAN-redirection drives on OS/2, Windows 95 and Windows NT or NFS-mounted file systems on AIX) are not supported.

### *on-nodes-clause*

Specifies the partition or partitions on which the containers are created in a partitioned database . If this clause or any other *on-nodes-clause* of this statement is not specified, then the containers are created on all partitions currently in the nodegroup. For a TEMPORARY table space when the clause is not specified, the containers will also be created on all new partitions added to the database. See page 563 for details on specifying this clause.

### **on-nodes-clause**

Specifies the partitions on which containers are created in a partitioned database .

### **ON NODES**

Keywords that indicate that specific partitions are specified. NODE is a synonym for NODES.

#### *node-number1*

Specify a specific partition (or node) number.

#### **TO** *node-number2*

Specify a range of partition (or node) numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers are included in the partitions for which the containers are created if the node is included in the nodegroup of the table space.

The partition specified by number and every partition (or node) in the range of partition must exist in the nodegroup on which the table space is defined (SQLSTATE 42729) . A partition -number may only appear explicitly or within a range in exactly one *on-nodes-clause* for the statement (SQLSTATE 42613).

### **EXTENTSIZE** *number-of-pages*

Specifies the number of PAGESIZE pages that will be written to a container before skipping to the next container. The extent size value can also be speci-

## CREATE TABLESPACE

fied as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages value for extent size. The database manager cycles repeatedly through the containers as data is stored.

The default value is provided by the DFT\_EXTENT\_SZ configuration parameter.

### **PREFETCHSIZE** *number-of-pages*

Specifies the number of PAGESIZE pages that will be read from the table space when data prefetching is being performed. The prefetch size value can also be specified as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages value for prefetch size. Prefetching reads in data needed by a query prior to it being referenced by the query, so that the query need not wait for I/O to be performed.

The default value is provided by the DFT\_PREFETCH\_SZ configuration parameter. (This configuration parameter, like all configuration parameters, is explained in detail in the *Administration Guide*.)

### **BUFFERPOOL** *bufferpool-name*

The name of the buffer pool used for tables in this table space. The buffer pool must exist (SQLSTATE 42704). If not specified, the default buffer pool (IBMDEFAULTBP) is used. The page size of the bufferpool must match the page size specified (or defaulted) for the table space (SQLSTATE 428CB). The nodegroup of the table space must be defined for the bufferpool (SQLSTATE 42735).

### **OVERHEAD** *number-of-milliseconds*

Any numeric literal (integer, decimal, or floating point) that specifies the I/O controller overhead and disk seek and latency time, in milliseconds. The number should be an average for all containers that belong to the table space, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

### **TRANSFERRATE** *number-of-milliseconds*

Any numeric literal (integer, decimal, or floating point) that specifies the time to read one page (4K or 8K) into memory, in milliseconds. The number should be an average for all containers that belong to the table space, if not the same for all containers. This value is used to determine the cost of I/O during query optimization.

### **DROPPED TABLE RECOVERY**

Dropped tables in the specified table space may be recovered using the RECOVER TABLE ON option of the ROLLFORWARD command.

## CREATE TABLESPACE

### Notes

- For information on how to determine the correct EXTENTSIZE, PREFETCHSIZE, OVERHEAD, and TRANSFERRATE values, refer to the *Administration Guide*.
- Choosing between a database-managed space or a system-managed space for a table space is a fundamental choice involving tradeoffs. See the *Administration Guide* for a discussion of those tradeoffs.
- When more than one TEMPORARY table space exists in the database, they will be used in round-robin fashion in order to balance their usage. See the *Administration Guide* for information on using more than one table space, rebalancing and recommended values for EXTENTSIZE, PREFETCHSIZE, OVERHEAD, and TRANSFERRATE.
- In a partitioned database if more than one partition resides on the same physical node, then the same device or specific path cannot be specified for such partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each partition or use a relative path name.
- You can specify a node expression for container string syntax when creating either SMS or DMS containers. You would typically specify the node expression if you are using multiple logical nodes in the partitioned database system. This ensures that container names are unique across nodes (database partition servers). When you specify the expression, either the node number is part of the container name, or, if you specify additional arguments, the result of the argument is part of the container name.

You use the argument “ \$N” ([blank]\$N) to indicate the node expression. The argument must occur at the end of the container string and can only be used in one of the following forms. In the table that follows, the node number is assumed to be 5:

<i>Table 20. Arguments for Creating Containers</i>		
Syntax	Example	Value
[blank]\$N	" \$N"	5
[blank]\$N+[number]	" \$N+1011"	1016
[blank]\$N%[number]	" \$N%3"	2
[blank]\$N+[number]%[number]	" \$N+12%13"	4
[blank]\$N%[number]+[number]	" \$N%3+20"	22
<b>Note:</b>		
<ul style="list-style-type: none"> <li>• % is modulus</li> <li>• In all cases, the operators are evaluated from left to right.</li> </ul>		

Some examples are as follows: *Example 1:*

```
CREATE TABLESPACE TS1 MANAGED BY DATABASE USING
(device '/dev/rcont $N' 20000)
```

On a two-node system, the following containers would be used:

```
/dev/rcont0 - on NODE 0
/dev/rcont1 - on NODE 1
```

## CREATE TABLESPACE

*Example 2:*

```
CREATE TABLESPACE TS2 MANAGED BY DATABASE USING  
(file '/DB2/containers/TS2/container $N+100' 10000)
```

On a four-node system, the following containers would be created:

```
/DB2/containers/TS2/container100 - on NODE 0  
/DB2/containers/TS2/container101 - on NODE 1  
/DB2/containers/TS2/container102 - on NODE 2  
/DB2/containers/TS2/container103 - on NODE 3
```

*Example 3:*

```
CREATE TABLESPACE TS3 MANAGED BY SYSTEM USING  
( '/TS3/cont $N%2', '/TS3/cont $N%2+2')
```

On a two-node system, the following containers would be created:

```
/TS3/cont0 - On NODE 0  
/TS3/cont2 - On NODE 0  
/TS3/cont1 - On NODE 1  
/TS3/cont3 - On NODE 1
```

## Examples

*Example 1:* Create a regular DMS table space on a UNIX-based system using 3 devices of 10000 4K pages each. Specify their I/O characteristics.

```
CREATE TABLESPACE PAYROLL  
MANAGED BY DATABASE  
USING (DEVICE '/dev/rhdisk6' 10000,  
      DEVICE '/dev/rhdisk7' 10000,  
      DEVICE '/dev/rhdisk8' 10000)  
OVERHEAD 24.1  
TRANSFERRATE 0.9
```

*Example 2:* Create a regular SMS table space on OS/2 or Windows NT using 3 directories on three separate drives, with a 64-page extent size, and a 32-page prefetch size.

```
CREATE TABLESPACE ACCOUNTING  
MANAGED BY SYSTEM  
USING ('d:\acc_tbsp', 'e:\acc_tbsp', 'f:\acc_tbsp')  
EXTENTSIZE 64  
PREFETCHSIZE 32
```

*Example 3:* Create a temporary DMS table space on Unix using 2 files of 50,000 pages each, and a 256-page extent size.

```
CREATE TEMPORARY TABLESPACE TEMPSPACE2  
MANAGED BY DATABASE  
USING (FILE '/tmp/tempespace2.f1' 50000,  
      FILE '/tmp/tempespace2.f2' 50000)  
EXTENTSIZE 256
```

## CREATE TABLESPACE

*Example 4:* Create a DMS table space on nodegroup ODDNODEGROUP (nodes 1,3,5) on a Unix partitioned database . On all partitions (or nodes) , use the device /dev/rhdisk0 for 10000 4K pages. Also specify a partition specific device for each partition with 40000 4K pages.

```
CREATE TABLESPACE PLANS
  MANAGED BY DATABASE
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn1hd01' 40000)
  ON NODE (1)
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn3hd03' 40000)
  ON NODE (3)
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn5hd05' 40000)
  ON NODE (5)
```

## CREATE TRIGGER

---

### CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger in the database.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement when the trigger is created must include at least one of the following:

- SYSADM or DBADM authority.
- ALTER privilege on the table on which the trigger is defined, or ALTERIN privilege on the schema of the table on which the trigger is defined and one of:
  - IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the trigger does not exist
  - CREATEIN privilege on the schema, if the schema name of the trigger refers to an existing schema .

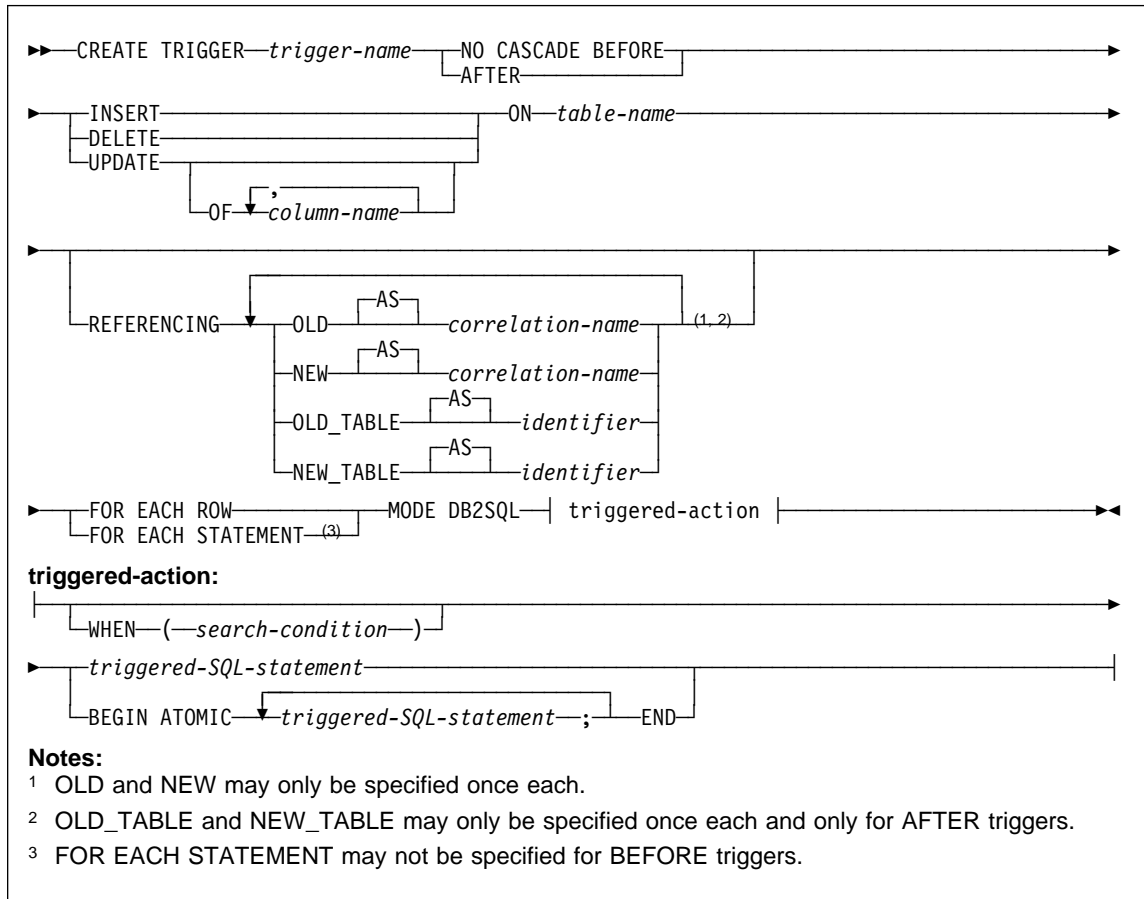
If the authorization ID of the statement does not have SYSADM or DBADM authority, the privileges that the authorization ID of the statement holds (without considering PUBLIC or group privileges) must include all of the following as long as the trigger exists:

- SELECT privilege on the table on which the trigger is defined, if any transition variables or tables are specified
- SELECT privilege on any table or view referenced in the triggered action condition
- Necessary privileges to invoke the triggered SQL statements specified.

If a trigger definer can only create the trigger because the definer has SYSADM authority, then the definer is granted explicit DBADM authority for the purpose of creating the trigger.

#### Syntax

## CREATE TRIGGER



### Description

#### *trigger-name*

Names the trigger. The name, including the implicit or explicit schema name must not identify a trigger already described in the catalog (SQLSTATE 42710). If a two part name is specified, the schema name cannot begin with "SYS" (SQLSTATE 42939).

#### **NO CASCADE BEFORE**

Specifies that the associated triggered action is to be applied before any changes caused by the actual update of the subject table are applied to the database. It also specifies that the triggered action of the trigger will not cause other triggers to be activated.

#### **AFTER**

Specifies that the associated triggered action is to be applied after the changes caused by the actual update of the subject table are applied to the database.

## CREATE TRIGGER

### INSERT

Specifies that the triggered action associated with the trigger is to be executed whenever an INSERT operation is applied to the designated base table.

### DELETE

Specifies that the triggered action associated with the trigger is to be executed whenever a DELETE operation is applied to the designated base table.

### UPDATE

Specifies that the triggered action associated with the trigger is to be executed whenever an UPDATE operation is applied to the designated base table subject to the columns specified or implied.

If the optional *column-name* list is not specified, every column of the table is implied. Therefore, omission of the *column-name* list implies that the trigger will be activated by the update of any column of the table.

### OF *column-name*,...

Each *column-name* specified must be a column of the base table (SQLSTATE 42703). No *column-name* shall appear more than once in the *column-name* list (SQLSTATE 42711). The trigger will only be activated by the update of a column identified in the *column-name* list.

### ON *table-name*

Designates the subject table of the trigger definition. The name must specify a base table or an alias that resolves to a base table (SQLSTATE 42809). The name must not specify a catalog table (SQLSTATE 42832), a typed table or a summary table (SQLSTATE 42997).

### REFERENCING

Specifies the correlation names for the *transition variables* and the table names for the *transition tables*. Correlation names identify a specific row in the set of rows affected by the triggering SQL operation. Table names identify the complete set of affected rows. Each row affected by the triggering SQL operation is available to the triggered action by qualifying columns with *correlation-names* specified as follows.

### OLD AS *correlation-name*

Specifies a correlation name which identifies the row state prior to the triggering SQL operation.

### NEW AS *correlation-name*

Specifies a correlation name which identifies the row state as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

The complete set of rows affected by the triggering SQL operation is available to the triggered action by using a temporary table name specified as follows.

### OLD\_TABLE AS *identifier*

Specifies a temporary table name which identifies the set of affected rows prior to the triggering SQL operation.



## CREATE TRIGGER

### **NEW\_TABLE AS** *identifier*

Specifies a temporary table name which identifies the affected rows as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

The following rules apply to the REFERENCING clause:

- None of the OLD and NEW correlation names and the OLD\_TABLE and NEW\_TABLE names can be identical (SQLSTATE 42712).
- Only one OLD and one NEW *correlation-name* may be specified for a trigger (SQLSTATE 42613).
- Only one OLD\_TABLE and one NEW\_TABLE *identifier* may be specified for a trigger (SQLSTATE 42613).
- The OLD *correlation-name* and the OLD\_TABLE *identifier* can only be used if the trigger event is either a DELETE operation or an UPDATE operation (SQLSTATE 42898). If the operation is a DELETE operation, OLD *correlation-name* captures the value of the deleted row. If it is an UPDATE operation, it captures the value of the row before the UPDATE operation. The same applies to the OLD\_TABLE *identifier* and the set of affected rows.
- The NEW *correlation-name* and the NEW\_TABLE *identifier* can only be used if the trigger event is either an INSERT operation or an UPDATE operation (SQLSTATE 42898). In both operations, the value of NEW captures the new state of the row as provided by the original operation and as modified by any BEFORE trigger that has executed to this point. The same applies to the NEW\_TABLE *identifier* and the set of affected rows.
- OLD\_TABLE and NEW\_TABLE *identifiers* cannot be defined for a BEFORE trigger (SQLSTATE 42898).
- OLD and NEW *correlation-names* cannot be defined for a FOR EACH STATEMENT trigger (SQLSTATE 42899).
- Transition tables cannot be modified (SQLSTATE 42807).
- The total of the references to the transition table columns and transition variables in the triggered-action cannot exceed the limit for the number of columns in a table or the sum of their lengths cannot exceed the maximum length of a row in a table (SQLSTATE 54040).
- The scope of each *correlation-name* and each *identifier* is the entire trigger definition.

### **FOR EACH ROW**

Specifies that the triggered action is to be applied once for each row of the subject table that is affected by the triggering SQL operation.

### **FOR EACH STATEMENT**

Specifies that the triggered action is to be applied only once for the whole statement. This type of trigger granularity cannot be specified for a BEFORE trigger (SQLSTATE 42613). If specified, an UPDATE or DELETE trigger is activated even when no rows are affected by the triggering UPDATE or DELETE statement.

## CREATE TRIGGER

### MODE DB2SQL

This clause is used to specify the mode of triggers. This is the only valid mode currently supported.

### triggered-action

Specifies the action to be performed when a trigger is activated. A triggered-action is composed of one or several *triggered-SQL-statements* and by an optional condition for the execution of the *triggered-SQL-statements*. If there is more than one *triggered-SQL-statement* in the triggered-action for a given trigger, they must be enclosed within the BEGIN ATOMIC and END keywords, separated by a semi-colon,<sup>66</sup> and are executed in the order they are specified.

### WHEN (*search-condition*)

Specifies a condition that is true, false, or unknown. The *search-condition* provides a capability to determine whether or not a certain triggered action should be executed.

The associated action is performed only if the specified search condition evaluates as true. If the WHEN clause is omitted, the associated *triggered-SQL-statements* are always performed.

### *triggered-SQL-statement*

If the trigger is a BEFORE trigger, then a triggered SQL statement must be one of the following (SQLSTATE 42987):

- a fullselect <sup>67</sup>
- a SET transition-variable SQL statement.
- a SIGNAL SQLSTATE statement

If the trigger is an AFTER trigger, then a triggered SQL statement must be one of the following (SQLSTATE 42987):

- an INSERT SQL statement
- a searched UPDATE SQL statement
- a searched DELETE SQL statement
- a SIGNAL SQLSTATE statement
- a fullselect <sup>67</sup>

The *triggered-SQL-statement* can not reference an undefined transition variable (SQLSTATE 42703).

The *triggered-SQL-statement* in a BEFORE trigger can not reference a summary table defined with REFRESH IMMEDIATE (SQLSTATE 42997).

---

<sup>66</sup> When using this form in the Command Line Processor, the statement terminating character cannot be the semi-colon. See the *Command Reference* for information on specifying an alternative terminating character.

<sup>67</sup> A common-table-expression may precede a fullselect.

### Notes

- Adding a trigger to a table that already has rows in it will not cause any triggered actions to be activated. Thus, if the trigger is designed to enforce constraints on the data in the table, those constraints may not be satisfied by the existing rows.
- If the events for two triggers occur simultaneously (for example, if they have the same event, activation time, and subject tables), then the first trigger created is the first to execute.
- If a column is added to the subject table after triggers have been defined, the following rules apply:
  - If the trigger is an UPDATE trigger that was specified without an explicit column list, then an update to the new column will cause the activation of the trigger.
  - The column will not be visible in the triggered action of any previously defined trigger.
  - The OLD\_TABLE and NEW\_TABLE transition tables will not contain this column. Thus, the result of performing a "SELECT \*" on a transition table will not contain the added column.
- If a column is added to any table referenced in a triggered action, the new column will not be visible to the triggered action.
- The result of a fullselect specified as a *triggered-SQL-statement* is not available inside or outside of the trigger.
- A before delete trigger defined on a table involved in a cycle of cascaded referential constraints should not include references to the table on which it is defined or any other table modified by cascading during the evaluation of the cycle of referential integrity constraints. The results of such a trigger are data dependent and therefore may not produce consistent results.

In its simplest form, this means that a before delete trigger on a table with a self-referencing referential constraint and a delete rule of CASCADE should not include any references to the table in the *triggered-action*.

- The creation of a trigger causes certain packages to be marked invalid:
  - If an update trigger without an explicit column list is created, then packages with an update usage on the target table are invalidated.
  - If an update trigger with a column list is created, then packages with update usage on the target table are only invalidated if the package also has an update usage on at least one column in the *column-name* list of the CREATE TRIGGER statement.
  - If an insert trigger is created, packages that have an insert usage on the target table are invalidated.
  - If a delete trigger is created, packages that have a delete usage on the target table are invalidated.
- A package remains invalid until the application program is explicitly bound or rebound, or it is executed and the database manager automatically rebinds it.

## CREATE TRIGGER

- **Inoperative triggers:** An *inoperative trigger* is a trigger that is no longer available and is therefore never activated. A trigger becomes inoperative if:
  - A privilege that the creator of the trigger is required to have for the trigger to execute is revoked.
  - An object such as a table, view or alias, upon which the triggered action is dependent, is dropped.
  - A view, upon which the triggered action is dependent, becomes inoperative.
  - An alias that is the subject table of the trigger is dropped.

In practical terms, an inoperative trigger is one in which a trigger definition has been dropped as a result of cascading rules for DROP or REVOKE statements. For example, when an view is dropped, any trigger with a *triggered-SQL-statement* defined using that view is made inoperative.

When a trigger is made inoperative, all packages with statements performing operations that were activating the trigger will be marked invalid. When the package is rebound (explicitly or implicitly) the **inoperative trigger is completely ignored**. Similarly, applications with dynamic SQL statements performing operations that were activating the trigger will also completely ignore any inoperative triggers.

The trigger name can still be specified in the DROP TRIGGER and COMMENT ON TRIGGER statements.

An inoperative trigger may be recreated by issuing a CREATE TRIGGER statement using the definition text of the inoperative trigger. This trigger definition text is stored in the TEXT column of SYSCAT.TRIGGERS. Note that there is no need to explicitly drop the inoperative trigger in order to recreate it. Issuing a CREATE TRIGGER statement with the same *trigger-name* as an inoperative trigger will cause that inoperative trigger to be replaced with a warning (SQLSTATE 01595).

Inoperative triggers are indicated by an X in the VALID column of the SYSCAT.TRIGGERS catalog view.

- **Errors executing triggers:** Errors that occur during the execution of triggered-SQL-statements are returned using SQLSTATE 09000 unless the error is considered severe. If the error is severe, the severe error SQLSTATE is returned. The SQLERRMC field of the SQLCA for non-severe error will include the trigger name, SQLCODE, SQLSTATE and as many tokens as will fit from the tokens of the failure.

A *triggered-SQL-statement* could be a SIGNAL SQLSTATE statement or contain a RAISE\_ERROR function. In both these cases, the SQLSTATE returned is the one specified in the SIGNAL SQLSTATE statement or the RAISE\_ERROR condition.

- Creating a trigger with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

## CREATE TRIGGER

### Examples

*Example 1:* Create two triggers that will result in the automatic tracking of the number of employees a company manages. The triggers will interact with the following tables:

EMPLOYEE table with these columns: ID, NAME, ADDRESS, and POSITION.  
COMPANY\_STATS table with these columns: NBEMP, NBPRODUCT, and REVENUE.

The first trigger 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:

```
CREATE TRIGGER NEW_HIRED
AFTER INSERT ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

The second trigger decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE:

```
CREATE TRIGGER FORMER_EMP
AFTER DELETE ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

*Example 2:* Create a trigger that ensures that whenever a parts record is updated, the following check and (if necessary) action is taken:

If the on-hand quantity is less than 10% of the maximum stocked quantity, then issue a shipping request ordering the number of items for the affected part to be equal to the maximum stocked quantity minus the on-hand quantity.

The trigger will interact with the PARTS table with these columns: PARTNO, DESCRIPTION, ON\_HAND, MAX\_STOCKED, and PRICE.

ISSUE\_SHIP\_REQUEST is a user-defined function that sends an order form for additional parts to the appropriate company.

```
CREATE TRIGGER REORDER
AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (N.ON_HAND < 0.10 * N.MAX_STOCKED)
BEGIN ATOMIC
VALUES(ISSUE_SHIP_REQUEST(N.MAX_STOCKED - N.ON_HAND, N.PARTNO));
END
```

*Example 3:* Create a trigger that will cause an error when an update occurs that would result in a salary increase greater than ten percent of the current salary.

## CREATE TRIGGER

```
CREATE TRIGGER RAISE_LIMIT
AFTER UPDATE OF SALARY ON EMPLOYEE
REFERENCING NEW AS N OLD AS O
FOR EACH ROW MODE DB2SQL
WHEN (N.SALARY > 1.1 * O.SALARY)
    SIGNAL SQLSTATE '75000' ('Salary increase>10%')
```

*Example 4:* Consider an application which records and tracks changes to stock prices. The database contains two tables, CURRENTQUOTE and QUOTEHISTORY.

Tables: CURRENTQUOTE (SYMBOL, QUOTE, STATUS)  
QUOTEHISTORY (SYMBOL, QUOTE, QUOTE\_TIMESTAMP)

When the QUOTE column of CURRENTQUOTE is updated, the new quote should be copied, with a timestamp, to the QUOTEHISTORY table. Also, the STATUS column of CURRENTQUOTE should be updated to reflect whether the stock is:

1. rising in value;
2. at a new high for the year;
3. dropping in value;
4. at a new low for the year;
5. steady in value.

CREATE TRIGGER statements that accomplish this are as follows.

- Trigger Definition to set the status:

```
CREATE TRIGGER STOCK_STATUS
NO CASCADE BEFORE UPDATE OF QUOTE ON CURRENTQUOTE
REFERENCING NEW AS NEWQUOTE OLD AS OLDQUOTE
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    SET NEWQUOTE.STATUS =
        CASE
            WHEN NEWQUOTE.QUOTE >
                (SELECT MAX(QUOTE) FROM QUOTEHISTORY
                 WHERE SYMBOL = NEWQUOTE.SYMBOL
                 AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
            THEN 'High'
            WHEN NEWQUOTE.QUOTE <
                (SELECT MIN(QUOTE) FROM QUOTEHISTORY
                 WHERE SYMBOL = NEWQUOTE.SYMBOL
                 AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
            THEN 'Low'
            WHEN NEWQUOTE.QUOTE > OLDQUOTE.QUOTE
            THEN 'Rising'
            WHEN NEWQUOTE.QUOTE < OLDQUOTE.QUOTE
            THEN 'Dropping'
            WHEN NEWQUOTE.QUOTE = OLDQUOTE.QUOTE
            THEN 'Steady'
        END;
END;
```

- Trigger Definition to record change in QUOTEHISTORY table:

## CREATE TRIGGER

```
CREATE TRIGGER RECORD_HISTORY
AFTER UPDATE OF QUOTE ON CURRENTQUOTE
REFERENCING NEW AS NEWQUOTE
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    INSERT INTO QUOTEHISTORY
        VALUES (NEWQUOTE.SYMBOL, NEWQUOTE.QUOTE, CURRENT_TIMESTAMP);
END
```

## CREATE TYPE (Structured)

---

### CREATE TYPE (Structured)

The CREATE TYPE statement defines a user-defined structured type. A user-defined structured type may include zero or more attributes. A structured type may be a subtype allowing attributes to be inherited from a supertype. Successful execution of the statement also generates functions to cast between the reference type and its representation type and generates support for the comparison operators (=, <>, <, <=, >, and >=) for use with the reference type.

### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- IMPLICIT\_SCHEMA authority on the database, if the schema name of the type does not refer to an existing schema.
- CREATEIN privilege on the schema, if the schema name of the type refers to an existing schema.

If UNDER is specified and authorization ID of the statement is not the same as the definer of the root type of the type hierarchy, then SYSADM or DBADM authority is required.

### Syntax

```
CREATE TYPE type-name [ UNDER supertype-name ]
AS ( ( attribute-definition ) ) [ WITHOUT COMPARISONS ]
[ NOT FINAL ] [ MODE DB2SQL ]

attribute-definition:
[ attribute-name ] data-type [ lob-options ]
[ datalink-options ]
```

### Description

*type-name*

Names the type. The name, including the implicit or explicit qualifier, must not identify any other type (built-in, structured, or distinct) already described in the catalog. The unqualified name must not be the same as the name of a built-in data type or BOOLEAN (SQLSTATE 42918). In dynamic SQL statements, the CURRENT



## CREATE TYPE (Structured)

SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

A number of names used as keywords in predicates are reserved for system use, and may not be used as a *type-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in “Basic Predicate” on page 136. Failure to observe this rule will lead to an error (SQLSTATE 42939).

If a two-part *type-name* is specified, the schema name cannot begin with "SYS"; otherwise, an error (SQLSTATE 42939) is raised.

### **UNDER** *supertype-name*

Specifies that this structured type is a subtype under the specified *supertype-name*. The *supertype-name* must identify an existing structured type (SQLSTATE 42704). If *supertype-name* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The structured type includes all the attributes of the supertype followed by the additional attributes given in the *attribute-definition*.

### *attribute-definition*

Defines the attributes of the structured type.

### *attribute-name*

The name of an attribute. The name cannot be the same as any other attribute of this structured type or any supertype of this structured type (SQLSTATE 42711).

A number of names used as keywords in predicates are reserved for system use, and may not be used as an *attribute-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in “Basic Predicate” on page 136. Failure to observe this rule will lead to an error (SQLSTATE 42939).

### *data-type*

The data type of the attribute. It is one of the data types listed under “CREATE TABLE” on page 522 other than LONG VARCHAR, LONG VARGRAPHIC, a distinct type based on LONG VARCHAR or LONG VARGRAPHIC (SQLSTATE 42601). The data type must identify an existing data type (SQLSTATE 42704). If *data-type* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The description of various data types is given in “CREATE TABLE” on page 522. If the attribute data type is a reference type, the target type of the reference must be a structured type that exists or is created by this statement (SQLSTATE 42704).

### *lob-options*

Specifies the options associated with LOB types (or distinct types based on LOB types). For a detailed description of *lob-options*, see “CREATE TABLE” on page 522.

## CREATE TYPE (Structured)

### *datalink-options*

Specifies the options associated with DATALINK types (or distinct types based on DATALINK types). For a detailed description of *datalink-options*, see “CREATE TABLE” on page 522.

Note that if no options are specified for a DATALINK type or distinct type sourced on DATALINK, LINKTYPE URL and NO LINK CONTROL options are the defaults.

### **WITHOUT COMPARISONS**

Indicates that there are no comparison functions supported for instances of the structured type.

### **NOT FINAL**

Indicates that the structured type may be used as a supertype.

### **MODE DB2SQL**

This clause is used to specify the mode of the type. This is the only value for mode currently supported.

## Notes

- Creating a structured type with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- The reference type for any structured type is represented internally by a value of type VARCHAR(16) FOR BIT DATA.
- Creation of a structured type automatically generates a set of functions for use with the type. All the functions are generated in the same schema as the structured type. If the signature of the generated function conflicts with the signature of an existing function in this schema, the statement fails (SQLSTATE 42710, SQLCODE -600). The generated functions cannot be dropped without dropping the structured type (SQLSTATE 42917, SQLCODE -658). The following functions are generated:
  - Reference Comparisons

Six comparison functions with names =, <>, <, <=, >, >= are generated for the reference type REF(*type-name*). Each of these functions takes two parameters of type REF(*type-name*) and returns true, false, or unknown. The comparison operators for REF(*type-name*) are defined to have the same behaviour as the comparison operators for the underlying data type of REF(*type-name*).<sup>68</sup>

The scope of the reference type is not considered in the comparison.
  - Cast functions

---

<sup>68</sup> All references in a type hierarchy have the same reference representation type. This enables REF(S) and REF(T) to be compared provided that S and T have a common supertype. Since uniqueness of the OID column is enforced only within a table hierarchy, it is possible that a value of REF(T) in one table hierarchy may be "equal" to a value of REF(T) in another table hierarchy, even though they reference different rows.

## CREATE TYPE (Structured)

Two cast functions are generated to cast between the generated reference type `REF(type-name)` and the underlying data type of this reference type.

- The name of the function to cast from the underlying type to the reference type is the name of the type being created.

The format of this function is:

```
CREATE FUNCTION type-name (VARCHAR(16) FOR BIT DATA)
    RETURNS REF(type-name) ...
```

- The name of the function to cast from the reference type to the underlying type of the reference type is `VARCHAR`.

The format of this function is:

```
CREATE FUNCTION VARCHAR ( REF(type-name) ) RETURNS VARCHAR(16) FOR BIT DATA
```

The schema name of the structured type must be included in the SQL path (see "SET PATH" on page 731 or the `FUNCPATH BIND` option as described in the *Embedded SQL Programming Guide*) for successful use of these operators and cast functions in SQL statements.

### Examples

*Example 1:* Create a type for department.

```
CREATE TYPE DEPT AS
    (NAME    VARCHAR(32),
     NUMBER  CHAR(6))
    WITHOUT COMPARISONS
    NOT FINAL
    MODE DB2SQL
```

*Example 2:* Create a type hierarchy consisting of a type for employees and a subtype for managers.

```
CREATE TYPE EMP AS
    (NAME    VARCHAR(32),
     DEPTREF REF(DEPT),
     SALARY  DECIMAL(10,2) )
    WITHOUT COMPARISONS
    NOT FINAL
    MODE DB2SQL
```

```
CREATE TYPE MGR UNDER EMP AS
    (HEADCOUNT INTEGER,
     BUDGET      DECIMAL(10,2) )
    WITHOUT COMPARISONS
    NOT FINAL
    MODE DB2SQL
```

## CREATE VIEW

---

### CREATE VIEW

The CREATE VIEW statement creates a view on one or more tables or views.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority or
- For each table or view identified in any fullselect:
  - CONTROL privilege on that table or view, or
  - SELECT privilege on that table or view

and at least one of the following:

- IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema name of the view does not exist
- CREATEIN privilege on the schema, if the schema name of the view refers to an existing schema .

If creating a subview, the authorization ID of the statement must:

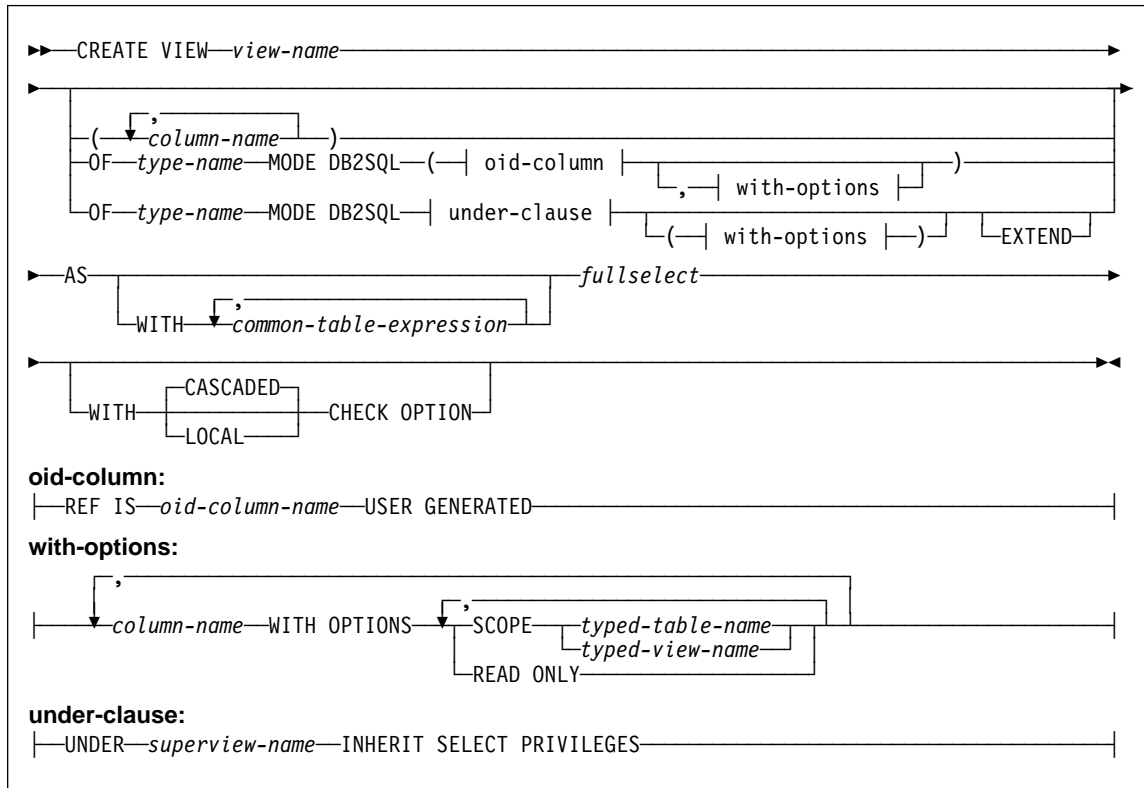
- be the same as the definer of the root table of the table hierarchy.
- have SELECT WITH GRANT on the underlying table of the subview or the superview must not have SELECT privilege granted to any user other than the view definer.

Group privileges are not considered for any table or view specified in the CREATE VIEW statement.

If a view definer can only create the view because the definer has SYSADM authority, then the definer is granted explicit DBADM authority for the purpose of creating the view.

#### Syntax

## CREATE VIEW



**Note:** See Chapter 5, “Queries” on page 315 for the syntax of *common-table-expression* and *fullselect*.

### Description

#### *view-name*

Names the view. The name, including the implicit or explicit qualifier, must not identify a table, view, or alias described in the catalog. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

The name can be the same as the name of an inoperative view (see Inoperative views on page 590). In this case the new view specified in the CREATE VIEW statement will replace the inoperative view. The user will get a warning (SQLSTATE 01595) when an inoperative view is replaced. No warning is returned if the application was bound with the bind option SQLWARN set to NO.

#### *column-name*

Names the columns in the view. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the view inherit the names of the columns of the result table of the fullselect.

## CREATE VIEW

A list of column names must be specified if the result table of the fullselect has duplicate column names or an unnamed column (SQLSTATE 42908). An unnamed column is a column derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

### **OF** *type-name*

Specifies that the columns of the view are based on the attributes of the structured type identified by *type-name*. If *type-name* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The type name must be the name of an existing user-defined type (SQLSTATE 42704) and it must be a structured type (SQLSTATE 428DP).

### **MODE DB2SQL**

This clause is used to specify the mode of the typed view. This is the only valid mode currently supported.

### **UNDER** *superview-name*

Indicates that the view is a subview of *superview-name*. The superview must be an existing view (SQLSTATE 42704) and the view must be defined using a structured type that is the immediate supertype of *type-name* (SQLSTATE 428DB). The schema name of *view-name* and *superview-name* must be the same (SQLSTATE 428DQ). The view identified by *superview-name* must not have any existing subview already defined using *type-name* (SQLSTATE 42742).

The columns of the view include the object identifier column of the superview with its type modified to be REF(*type-name*), followed by columns based on the attributes of *type-name* (remember that the type includes the attributes of its supertype).

### **INHERIT SELECT PRIVILEGES**

Any user or group holding a SELECT privilege on the superview will be granted an equivalent privilege on the newly created subview. The subview definer is considered to be the grantor of this privilege.

### *OID-column*

Defines the object identifier column for the typed view.

### **REF IS** *OID-column-name* **USER GENERATED**

Specifies that an object identifier (OID) column is defined in the view as the first column. An OID is required for the root view of a view hierarchy (SQLSTATE 428DX). The view must be a typed view (the OF clause must be present) that is not a subview (SQLSTATE 42613). The name for the column is defined as *OID-column-name* and cannot be the same as the name of any attribute of the structured type *type-name* (SQLSTATE 42711). The first column specified in *fullselect* must be of type REF(*type-name*) (you may need to cast it so that it has the appropriate type) and it must be based on a not nullable column on which uniqueness is enforced through an index (primary key, unique constraint, unique index, or OID-column). This column will be referred to as the *object identifier column* or *OID column*. The keywords

## CREATE VIEW

USER GENERATED indicate that the initial value for the OID column must be provided by the user when inserting a row. Once a row is inserted, the OID column cannot be updated (SQLSTATE 42808).

### *with-options*

Defines additional options that apply to columns of a typed view.

### *column-name* **WITH OPTIONS**

Specifies the name of the column for which additional options are specified. The *column-name* must correspond to the name of an attribute defined in (not inherited by) the *type-name* of the view. The column must be a reference type (SQLSTATE 42842). It cannot correspond to a column that also exists in the superview (SQLSTATE 428DJ). A column name can only appear in one WITH OPTIONS SCOPE clause in the statement (SQLSTATE 42613).

### **SCOPE**

Identifies the scope of the reference type column. A scope must be specified for any column that is intended to be used as the left operand of a dereference operator or as the argument of the Deref function.

Specifying the scope for a reference type column may be deferred to a subsequent ALTER VIEW statement (if the scope is not inherited) to allow the target table or view to be defined, usually in the case of mutually referencing views and tables. If no scope is specified for a reference type column of the view and the underlying table or view column was scoped, then the underlying column's scope is inherited by the reference type column. The column remains unscoped if the underlying table or view column did not have a scope. See "Notes" on page 589 for more information about scope and reference type columns.

### *typed-table-name*

The name of a typed table. The table must already exist or be the same as the name of the table being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

### *typed-view-name*

The name of a typed view. The view must already exist or be the same as the name of the view being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

### **READ ONLY**

Identifies the column as a read-only column. This option is used to force a column to be read-only so that subview definitions can specify an expression for the same column that is implicitly read-only.

## CREATE VIEW

**AS** Identifies the view definition.

**WITH** *common-table-expression*

Defines a common table expression for use with the *fullselect* that follows. A common table expression cannot be specified when defining a typed view. See “common-table-expression” on page 356.

*fullselect*

Defines the view. At any time, the view consists of the rows that would result if the SELECT statement were executed. *Fullselect* must be the *fullselect* form of a SELECT statement that does not reference host variables or parameter markers. When defining a typed view, the *fullselect* must not include references to the NODENUMBER or PARTITION functions, non-deterministic functions, or functions defined to have external action (SQLSTATE 428EA). See Chapter 5, “Queries” on page 315 for an explanation of *fullselect*.

When defining a typed view that is the root view of the view hierarchy, the following rules apply to the *fullselect*.

- The *fullselect* must consist of a single subselect with a FROM clause that consists of a single table or view (called the *underlying* table or view). If the FROM clause references a view, it must be a typed view since the first column must be based on an object identifier column. If subviews will be defined, ensure that the object identifier column values will be distinct from those of the subview by using predicates or by specifying ONLY around the underlying subtable or subview.
- The first expression in the SELECT list must have a data type of REF(*type-name*). CAST specifications or cast functions may be used in the expression to convert the underlying column to the correct type. The column must be an OID column or a non-nullable column of a primary key, unique constraint, or unique index that have all the other columns fixed to a constant value by an equality predicate in the WHERE clause (SQLSTATE 428EA).
- The data types of the second through last columns in the SELECT list must be assignable to the data types of the attributes of *type-name* (SQLSTATE 42825).
- The subselect must not include a GROUP BY clause or HAVING clause (SQLSTATE 42803).

When defining a typed view that is a subview, the following rules apply to the *fullselect*.

- The *fullselect* must consist of a single subselect with a FROM clause that consists of a single table or view (called the *underlying* table or view). The underlying table or view must be a subtable or view of the underlying table or view of the superview. Use predicates in the WHERE clause or ONLY in the FROM clause to ensure that the database manager can prove that the OID column values of the selected rows for the subview are distinct from the set of OID column values selected by any other view in the typed view hierarchy (SQLSTATE 428EA).
- If EXTEND AS is specified, the SELECT list must only include expressions to correspond to the attributes of *type-name* that are not inherited from its super-



type. The other columns are inherited from the superview including the object identifier column.

If AS is specified without EXTEND, the SELECT list must include all expressions to correspond to the attributes of *type-name*, as well as the OID column. The OID column must be based on the same column as in the definition of the root view of the typed view hierarchy. An expression in the select list that corresponds to a column of the superview cannot change the column to read-only (SQLSTATE 428EB) by specifying an expression that is implicitly not updatable. Therefore, if an expression in the select list is different from the inherited expression defined in the superview, the column must have been defined as read-only in the superview (implicitly or with option READ ONLY).

- The data types of these expressions in the SELECT list must be assignable to the data types of the corresponding attributes of *type-name* (SQLSTATE 42825).
- The subselect must not include a GROUP BY clause or HAVING clause (SQLSTATE 42803).

### 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. A row that does not conform to the definition of the view is a row that does not satisfy the search conditions of the view.

WITH CHECK OPTION must not be specified if the view is read-only (SQLSTATE 42813). If WITH CHECK OPTION is specified for an updatable view that does not allow inserts, then the constraint applies to updates only.

WITH CHECK OPTION must not be specified if the view references the NODENUMBER or PARTITION function, a non-deterministic function, or a function with external action (SQLSTATE 42997).

WITH CHECK OPTION must not be specified if the view is a typed view (SQLSTATE 42997).

If WITH CHECK OPTION is omitted, the definition of the view is not used in the checking of any insert or update operations that use the view. Some checking might still occur during insert or update operations if the view is directly or indirectly dependent on another view that includes WITH CHECK OPTION. Because the definition of the view is not used, rows might be inserted or updated through the view that do not conform to the definition of the view.

### CASCADED

The WITH CASCADED CHECK OPTION constraint on a view *V* means that *V* inherits the search conditions as constraints from any updatable view on which *V* is dependent. Furthermore, every updatable view that is dependent on *V* is also subject to these constraints. Thus, the search conditions of *V* and each view on which *V* is dependent are ANDed together to form a constraint that is applied for an insert or update of *V* or of any view dependent on *V*.

## CREATE VIEW

### LOCAL

The WITH LOCAL CHECK OPTION constraint on a view *V* means the search condition of *V* is applied as a constraint for an insert or update of *V* or of any view that is dependent on *V*.

The difference between CASCADED and LOCAL is shown in the following example. Consider the following updatable views (substituting for *Y* from column headings of the table that follows):

```
V1 defined on table T
V2 defined on V1 WITH Y CHECK OPTION
V3 defined on V2
V4 defined on V3 WITH Y CHECK OPTION
V5 defined on V4
```

The following table shows the search conditions against which inserted or updated rows are checked:

	Y is LOCAL	Y is CASCADED
V1 checked against:	no view	no view
V2 checked against:	V2	V2, V1
V3 checked against:	V2	V2, V1
V4 checked against:	V2, V4	V4, V3, V2, V1
V5 checked against:	V2, V4	V4, V3, V2, V1

Consider the following updatable view which shows the impact of the WITH CHECK OPTION using the default CASCADED option:

```
CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10
```

```
CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CHECK OPTION
```

```
CREATE VIEW V3 AS SELECT COL1 FROM V2 WHERE COL1 < 100
```

The following INSERT statement using *V1* will succeed because *V1* does not have a WITH CHECK OPTION and *V1* is not dependent on any other view that has a WITH CHECK OPTION.

```
INSERT INTO V1 VALUES(5)
```

The following INSERT statement using *V2* will result in an error because *V2* has a WITH CHECK OPTION and the insert would produce a row that did not conform to the definition of *V2*.

```
INSERT INTO V2 VALUES(5)
```

The following INSERT statement using *V3* will result in an error even though it does not have WITH CHECK OPTION because *V3* is dependent on *V2* which does have a WITH CHECK OPTION (SQLSTATE 44000).

```
INSERT INTO V3 VALUES(5)
```

The following INSERT statement using V3 will succeed even though it does not conform to the definition of V3 (V3 does not have a WITH CHECK OPTION); it does conform to the definition of V2 which does have a WITH CHECK OPTION.

```
INSERT INTO V3 VALUES(200)
```

### Notes

- Creating a view with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT\_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- View columns inherit the NOT NULL WITH DEFAULT attribute from the base table or view except when columns are derived from an expression. When a row is inserted or updated into an updatable view, it is checked against the constraints (primary key, referential integrity, and check) if any are defined on the base table.
- A new view cannot be created if it uses an inoperative view in its definition. (SQLSTATE 51024)
- When subviews are defined, the values of the rows of the object identifier column must be proven by the database manager to be distinct from the set of object identifier values in the rest of the typed view hierarchy. If it is determined that they may not be unique, an error is returned (SQLSTATE 428EA, reason code 2). The following are some guidelines to use when defining subviews to help specify a fullselect that results in distinct OID values between subviews.
  - When defining a typed view hierarchy over a typed table hierarchy, use ONLY in the FROM clause.
  - When using expressions in WHERE clauses that are intended to be the same, ensure that exactly the same syntax is used throughout the typed view hierarchy.
  - For predicates that contribute to the uniqueness of the OID column values between subviews:
    - always specify the expression that includes column names as the left-most expression
    - use a constant for other expression(s)
    - do not include subselects or dereference operations.
- **Deletable views:** A view is *deletable* if all of the following are true:
  - each FROM clause of the outer fullselect identifies only one base table (with no OUTER clause), deletable view (with no OUTER clause), deletable nested table expression, or deletable common table expression
  - the outer fullselect does not include a VALUES clause
  - the outer fullselect does not include a GROUP BY clause or HAVING clause
  - the outer fullselect does not include column functions in the select list

## CREATE VIEW

- the outer fullselect does not include SET operations (UNION, EXCEPT or INTERSECT) with the exception of UNION ALL
- the base tables in the operands of a UNION ALL must not be the same table and each operand must be deletable
- the select list of the outer fullselect does not include DISTINCT
- **Updatable views:** A column of a view is *updatable* if all of the following are true:
  - the view is deletable
  - the column resolves to a column of a base table (not using a dereference operation) and READ ONLY option is not specified
  - all the corresponding columns of the operands of a UNION ALL have exactly matching data types (including length or precision and scale) and matching default values if the fullselect of the view includes a UNION ALL

A view is *updatable* if ANY column of the view is updatable.

- **Insertable views:**

A view is *insertable* if ALL columns of the view are updatable and the fullselect of the view does not include UNION ALL.

- **Read-only views:** A view is *read-only* if it is NOT deletable.

The READONLY column in the SYSCAT.VIEWS catalog view indicates if a view is read-only.

- Common table expressions and nested table expressions follow the same set of rules for determining whether they are deletable, updatable, insertable or read-only.

- **Inoperative views:** An *inoperative view* is a view that is no longer available for SQL statements. A view becomes inoperative if:

- A privilege, upon which the view definition is dependent, is revoked.
- An object such as a table, alias or function, upon which the view definition is dependent, is dropped.
- A view, upon which the view definition is dependent, becomes inoperative.
- A view that is the superview of the view definition (the subview) becomes inoperative.

In practical terms, an inoperative view is one in which the view definition has been unintentionally dropped. For example, when an alias is dropped, any view defined using that alias is made inoperative. All dependent views also become inoperative and packages dependent on the view are no longer valid.

Until the inoperative view is explicitly recreated or dropped, a statement using that inoperative view cannot be compiled (SQLSTATE 51024) with the exception of the CREATE ALIAS, CREATE VIEW, DROP VIEW, and COMMENT ON TABLE statements. Until the inoperative view has been explicitly dropped, its qualified name cannot be used to create another table or alias. (SQLSTATE 42710)

## CREATE VIEW

An inoperative view may be recreated by issuing a CREATE VIEW statement using the definition text of the inoperative view. This view definition text is stored in the TEXT column of the SYSCAT.VIEWS catalog. When recreating an inoperative view, it is necessary to explicitly grant any privileges required on that view by others, due to the fact that all authorization records on a view are deleted if the view is marked inoperative. Note that there is no need to explicitly drop the inoperative view in order to recreate it. Issuing a CREATE VIEW statement with the same *view-name* as an inoperative view will cause that inoperative view to be replaced, and the CREATE VIEW statement will return a warning (SQLSTATE 01595).

Inoperative views are indicated by an X in the VALID column of the SYSCAT.VIEWS catalog view and an X in the STATUS column of the SYSCAT.TABLES catalog view.

- **Privileges**

The definer of a view always receives the SELECT privilege on the view as well as the right to drop the view. The definer of a view will get CONTROL privilege on the view only if the definer has CONTROL privilege on every base table or view identified in the fullselect, or if the definer has SYSADM or DBADM authority.

The definer of the view is granted INSERT, UPDATE, column level UPDATE or DELETE privileges on the view if the view is not read-only and the definer has the corresponding privileges on the underlying objects.

The definer of a view only acquires privileges if the privileges from which they are derived exist at the time the view is created. The definer must have these privileges either directly or because PUBLIC has the privilege. Privileges held by groups of which the view definer is a member, are not considered.

When a subview is created, the SELECT privileges held on the immediate super-view are automatically granted on the subview.

- **Scope and REF columns**

When selecting a reference type column in the fullselect of a view definition, consider the target type and scope that is required.

- If the required target type and scope is the same as the underlying table or view, the column can simply be selected.
- If the scope needs to be changed, use the WITH OPTIONS SCOPE clause to define the required scope table or view.
- If the target type of the reference needs to be changed, the column must be cast first to the representation type of the reference and then to the new reference type. The scope in this case can be specified in the cast to the reference type or using the WITH OPTIONS SCOPE clause. For example, assume you select column Y defined as REF(TYP1) SCOPE TAB1. You want this to be defined as REF(VTYP1) SCOPE VIEW1. The select list item would be as follows:

```
CAST(CAST(Y AS VARCHAR(16) FOR BIT DATA) AS REF(VTYP1) SCOPE VIEW1)
```

## CREATE VIEW

### Examples

*Example 1:* Create a view named MA\_PROJ upon the PROJECT table that contains only those rows with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE VIEW MA_PROJ AS SELECT *
FROM PROJECT
WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

*Example 2:* Create a view as in example 1, but select only the columns for project number (PROJNO), project name (PROJNAME) and employee in charge of the project (RESPEMP).

```
CREATE VIEW MA_PROJ
AS SELECT PROJNO, PROJNAME, RESPEMP
FROM PROJECT
WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

*Example 3:* Create a view as in example 2, but, in the view, call the column for the employee in charge of the project IN\_CHARGE.

```
CREATE VIEW MA_PROJ
(PROJNO, PROJNAME, IN_CHARGE)
AS SELECT PROJNO, PROJNAME, RESPEMP
FROM PROJECT
WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Note: Even though only one of the column names is being changed, the names of all three columns in the view must be listed in the parentheses that follow MA\_PROJ.

*Example 4:* Create a view named PRJ\_LEADER that contains the first four columns (PROJNO, PROJNAME, DEPTNO, RESPEMP) from the PROJECT table together with the last name (LASTNAME) of the person who is responsible for the project (RESPEMP). Obtain the name from the EMPLOYEE table by matching EMPNO in EMPLOYEE to RESPEMP in PROJECT.

```
CREATE VIEW PRJ_LEADER
AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME
FROM PROJECT, EMPLOYEE
WHERE RESPEMP = EMPNO
```

*Example 5:* Create a view as in example 4, but in addition to the columns PROJNO, PROJNAME, DEPTNO, RESPEMP, and LASTNAME, show the total pay (SALARY + BONUS + COMM) of the employee who is responsible. Also select only those projects with mean staffing (PRSTAFF) greater than one.

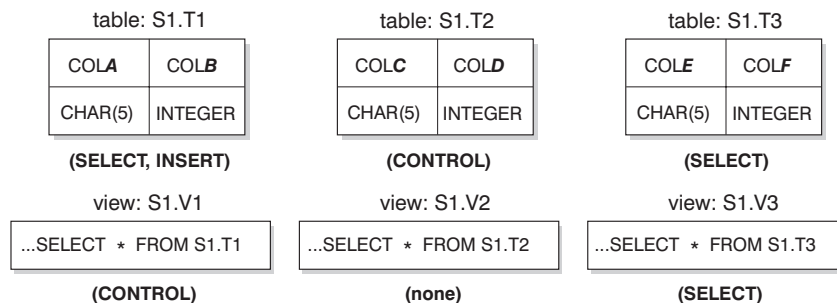
```
CREATE VIEW PRJ_LEADER
(PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, TOTAL_PAY )
AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, SALARY+BONUS+COMM
FROM PROJECT, EMPLOYEE
WHERE RESPEMP = EMPNO
AND PRSTAFF > 1
```

## CREATE VIEW

Specifying the column name list could be avoided by naming the expression SALARY+BONUS+COMM as TOTAL\_PAY in the fullselect.

```
CREATE VIEW PRJ_LEADER
AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP,
           LASTNAME, SALARY+BONUS+COMM AS TOTAL_PAY
FROM PROJECT, EMPLOYEE
WHERE RESPEMP = EMPNO AND PRSTAFF > 1
```

*Example 6:* Given the set of tables and views shown in the following figure:



*Figure 13. Tables and Views for Example 6*

User ZORPIE (who does not have either DBADM or SYSADM authority) has been granted the privileges shown in brackets below each object:

1. ZORPIE will get CONTROL privilege on the view that she creates with:

```
CREATE VIEW VA AS SELECT * FROM S1.V1
```

because she has CONTROL on S1.V1.<sup>69</sup> It does not matter which, if any, privileges she has on the underlying base table.

2. ZORPIE will not be allowed to create the view:

```
CREATE VIEW VB AS SELECT * FROM S1.V2
```

because she has neither CONTROL nor SELECT on S1.V2. It does not matter that she has CONTROL on the underlying base table (S1.T2).

3. ZORPIE will get CONTROL privilege on the view that she creates with:

```
CREATE VIEW VC (COLA, COLB, COLC, COLD)
AS SELECT * FROM S1.V1, S1.T2
WHERE COLA = COLC
```

because the fullselect of ZORPIE.VC references view S1.V1 and table S1.T2 and she has CONTROL on both of these. Note that the view VC is read-only, so ZORPIE does not get INSERT, UPDATE or DELETE privileges.

<sup>69</sup> CONTROL on S1.V1 must have been granted to ZORPIE by someone with DBADM or SYSADM authority.

## CREATE VIEW

4. ZORPIE will get SELECT privilege on the view that she creates with:

```
CREATE VIEW VD (COLA,COLB, COLE, COLF)
AS SELECT * FROM S1.V1, S1.V3
WHERE COLA = COLE
```

because the fullselect of ZORPIE.VD references the two views S1.V1 and S1.V3, one on which she has only SELECT privilege, and one on which she has CONTROL privilege. She is given the lesser of the two privileges, SELECT, on ZORPIE.VD.

5. ZORPIE will get INSERT, UPDATE and DELETE privilege WITH GRANT OPTION and SELECT privilege on the view VE in the following view definition.

```
CREATE VIEW VE
AS SELECT * FROM S1.V1
WHERE COLA > ANY
  (SELECT COLE FROM S1.V3)
```

ZORPIE's privileges on VE are determined primarily by her privileges on S1.V1. Since S1.V3 is only referenced in a subquery, she only needs SELECT privilege on S1.V3 to create the view VE. The definer of a view only gets CONTROL on the view if they have CONTROL on all objects referenced in the view definition. ZORPIE does not have CONTROL on S1.V3, consequently she does not get CONTROL on VE.



---

## DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

### Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is not an executable statement and cannot be dynamically prepared.

### Authorization

The term “SELECT statement of the cursor” is used in order to specify the authorization rules. The SELECT statement of the cursor is one of the following:

- The prepared select-statement identified by the *statement-name*
- The specified *select-statement*.

For each table or view identified (directly or using an alias) in the SELECT statement of the cursor, the privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority.
- For each table or view identified in the *select-statement*:
  - SELECT privilege on the table or view, or
  - CONTROL privilege of the table or view.

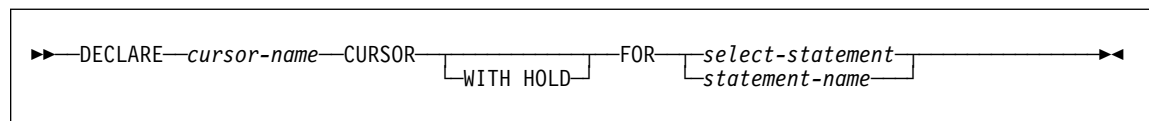
#### If *statement-name* is specified:

- The authorization ID of the statement is the run-time authorization ID.
- 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:

- GROUP privileges are not checked.
- The authorization ID of the statement is the authorization ID specified during program preparation.

### Syntax



### Description

#### *cursor-name*

Specifies the name of the cursor created when the source program is run. The name must not be the same as the name of another cursor declared in the source program. The cursor must be opened before use (see “OPEN” on page 668).

## DECLARE CURSOR

### WITH HOLD

Maintains resources across multiple units of work. The effect of the WITH HOLD cursor attribute is as follows:

- For units of work ending with COMMIT:
  - Open cursors defined WITH HOLD remain open. The cursor is positioned before the next logical row of the results table.

If a DISCONNECT statement is issued after a COMMIT statement for a connection with WITH HOLD cursors, the held cursors must be explicitly closed or the connection will be assumed to have performed work (simply by having open WITH HELD cursors even though no SQL statements were issued) and the DISCONNECT statement will fail.
  - All locks are released, except locks protecting the current cursor position of open WITH HOLD cursors. The locks held include the locks on the table, and for parallel environments, the locks on rows where the cursors are currently positioned. Locks on packages and dynamic SQL sections (if any) are held.
  - Valid operations on cursors defined WITH HOLD immediately following a COMMIT request are:
    - FETCH: Fetches the next row of the cursor.
    - CLOSE: Closes the cursor.
  - UPDATE and DELETE CURRENT OF CURSOR are valid only for rows that are fetched within the same unit of work.
  - LOB locators are freed.
- For units of work ending with ROLLBACK:
  - All open cursors are closed.
  - All locks acquired during the unit of work are released.
  - LOB locators are freed.
- For special COMMIT case:
  - Packages may be recreated either explicitly, by binding the package, or implicitly, because the package has been invalidated and then dynamically recreated the first time it is referenced. All held cursors are closed during package rebind. This may result in errors during subsequent execution.

***select-statement:*** Identifies the SELECT statement of the cursor. The select-statement must not include parameter markers, but may 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 355 for an explanation of select-statement.

***statement-name:*** The SELECT statement of the cursor is the prepared SELECT statement identified by the statement-name when the cursor is opened. The statement-

## DECLARE CURSOR

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 673.

### Notes

- A program called from another program or from a different source file within the same program cannot use the cursor that was opened by the calling program.
- If the SELECT statement of a cursor contains CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, all references to these special registers will yield the same value on each FETCH. This value is determined when the cursor is opened.
- For more efficient processing of data, the database manager can block data for read-only cursors when retrieving data from a remote server. The use of the FOR UPDATE clause helps the database manager decide whether a cursor is updatable or not. Updatability is also used to determine the access path selection as well. If a cursor is not going to be used in a Positioned UPDATE or DELETE statement, it should be declared as FOR READ ONLY.
- 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.
- A cursor is *deletable* if all of the following are true:
  - each FROM clause of the outer fullselect identifies only one base table or deletable view (cannot identify a nested or common table expression) without use of the OUTER clause
  - the outer fullselect does not include a VALUES clause
  - the outer fullselect does not include a GROUP BY clause or HAVING clause
  - the outer fullselect does not include column functions in the select list
  - the outer fullselect does not include SET operations (UNION, EXCEPT, or INTERSECT) with the exception of UNION ALL
  - the select list of the outer fullselect does not include DISTINCT
  - the select-statement does not include an ORDER BY clause
  - the select-statement does not include a FOR READ ONLY clause <sup>71</sup>
  - one or more of the following is true:
    - the FOR UPDATE clause <sup>70</sup> is specified
    - the cursor is statically defined
    - the LANGLEVEL bind option is MIA or SQL92E

<sup>70</sup> The FOR UPDATE clause is defined in the “update-clause” on page 361.

<sup>71</sup> The FOR READ ONLY clause is defined in the “read-only-clause” on page 362.

## DECLARE CURSOR

A column in the select list of the outer fullselect associated with a cursor is *updatable* if all of the following are true:

- the cursor is deletable
- the column resolves to a column of the base table
- the LANGLEVEL bind option is MIA, SQL92E or the select-statement includes the FOR UPDATE clause <sup>70</sup> (the column must be specified explicitly or implicitly in the FOR UPDATE clause).

A cursor is *read-only* if it is not deletable.

A cursor is *ambiguous* if all of the following are true:

- the select-statement is dynamically prepared
- the select-statement does not include either the FOR READ ONLY clause <sup>71</sup> or the FOR UPDATE clause <sup>70</sup>
- the LANGLEVEL bind option is SAA1
- the cursor otherwise satisfies the conditions of a deletable cursor.

An ambiguous cursor is considered read-only if the BLOCKING bind option is ALL, otherwise it is considered deletable.

- Cursors in stored procedures that are called by application programs written using CLI can be used to define result sets that are returned directly to the client application. See the “Notes” on page 413.

### Example

The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DEPARTMENT
  WHERE ADMRDEPT = 'A00';
```

---

## DELETE

The DELETE statement deletes rows from a table or view. 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

A DELETE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

To execute either form of this statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- DELETE privilege on the table or view for which rows are to be deleted
- CONTROL privilege on the table or view for which rows are to be deleted
- SYSADM or DBADM authority.

To execute a Searched DELETE statement, the privileges held by the authorization ID of the statement must also include at least one of the following for each table or view referenced by a subquery:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

When the package is precompiled with SQL92 rules<sup>72</sup> and the searched form of a DELETE includes a reference to a column of the table or view in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

Group privileges are not checked for static DELETE statements.

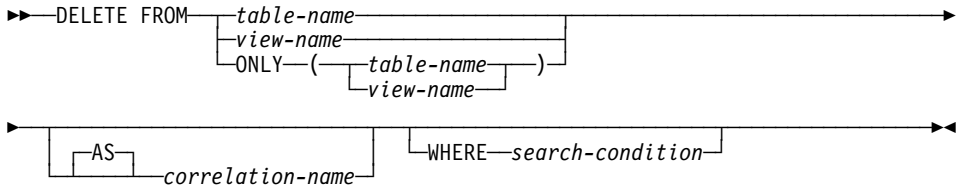
---

<sup>72</sup> The package used to process the statement is precompiled using option LANGLEVEL with value SQL92E or MIA.

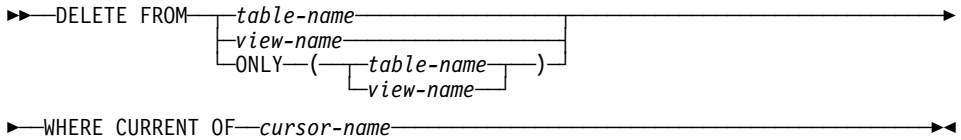
# DELETE

## Syntax

### Searched DELETE:



### Positioned DELETE:



## Description

### FROM *table-name* or *view-name*

Identifies the table or view from which rows are to be deleted. The name must identify a table or view that exists in the catalog, but it must not identify a catalog table, a catalog view, a summary table or a read-only view. (For an explanation of read-only views, see “CREATE VIEW” on page 582.)

If *table-name* is a typed table, rows of the table or any of its proper subtables may get deleted by the statement. Only the columns of the specified table may be referenced in the WHERE clause. For a positioned DELETE, the associated cursor must also have specified the table or view in the FROM clause without using ONLY.

### FROM ONLY (*table-name*)

Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

### FROM ONLY (*view-name*)

Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

### correlation-name

May be used within the search-condition to designate the table or view. (For an explanation of correlation-name, see Chapter 3, “Language Elements” on page 45.)

## WHERE

Specifies a condition that selects the rows to be deleted. The clause can be omitted, a search condition specified, or a cursor named. If the clause is omitted, all rows of the table or view are deleted.

### *search-condition*

Is any search condition as described in “Search Conditions” on page 153. 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 once, whereas a subquery with a correlated reference may have to be executed once for each row. If a subquery refers to the object table of a DELETE statement or a dependent table with a delete rule of CASCADE or SET NULL, the subquery is completely evaluated before any rows are deleted.

### **CURRENT OF** *cursor-name*

Identifies a cursor that is defined in a DECLARE CURSOR statement of the program. The DECLARE CURSOR statement must precede the DELETE statement.

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 595.)

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.

## Rules

- If the identified table or the base table of the identified view is a parent, the rows selected for delete must not have any dependents in a relationship with a delete rule of RESTRICT, and the DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT.

If the delete operation is not prevented by a RESTRICT delete rule, the selected rows are deleted. Any rows that are dependents of the selected rows are also affected:

- The nullable columns of the foreign keys of any rows that are their dependents in a relationship with a delete rule of SET NULL are set to the null value.
- Any rows that are their dependents in a relationship with a delete rule of CASCADE are also deleted, and the above rules apply, in turn, to those rows.

## DELETE

The delete rule of NO ACTION is checked to enforce that any non-null foreign key refers to an existing parent row after the other referential constraints have been enforced.

### Notes

- If an error occurs during the execution of a multiple row DELETE, no changes are made to the database.
- Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful DELETE statement. Issuing a COMMIT or ROLLBACK statement will release the locks. 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
  - Another application process using isolation level UR.

The locks can prevent other application processes from performing operations on the table.

- If an application process deletes a row on which any of its cursors are positioned, those cursors are positioned before the next row of their 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.
- SQLERRD(3) in the SQLCA shows the number of rows deleted from the object table after the statement executes. It does not include rows that were deleted as a result of a CASCADE delete rule. SQLERRD(5) in the SQLCA shows the number of rows affected by referential constraints and by triggered statements. It includes rows that were deleted as a result of a CASCADE delete rule and rows in which foreign keys were set to NULL as the result of a SET NULL delete rule. With regards to triggered statements, it includes the number of rows that were inserted, updated, or deleted. (For a description of the SQLCA, see Appendix B, "SQL Communication Area (SQLCA)" on page 759.)
- If an error occurs that prevents deleting all rows matching the search condition and all operations required by existing referential constraints, no changes are made to the table and the error is returned.
- For any deleted row that includes currently linked files through DATALINK columns, the files are unlinked, and will be either restored or deleted, depending on the datalink column definition.

An error may occur when attempting to delete a DATALINK value if the file server of value is no longer registered with the database server (SQLSTATE 55022).

An error may also occur when deleting a row that has a link to a server that is unavailable at the time of deletion (SQLSTATE 57050).



## DELETE

### Examples

*Example 1:* Delete department (DEPTNO) 'D11' from the DEPARTMENT table.

```
DELETE FROM DEPARTMENT  
WHERE DEPTNO = 'D11'
```

*Example 2:* Delete all the departments from the DEPARTMENT table (that is, empty the table).

```
DELETE FROM DEPARTMENT
```

## DESCRIBE

---

### DESCRIBE

The DESCRIBE statement obtains information about a prepared statement. For an explanation of prepared statements, see “PREPARE” on page 673.

### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

None required.

### Syntax

```
►►—DESCRIBE—statement-name—INTO—descriptor-name—◄◄
```

### Description

#### *statement-name*

Identifies the statement about which information is required. When the DESCRIBE statement is executed, the name must identify a prepared statement.

#### **INTO** *descriptor-name*

Identifies an SQL descriptor area (SQLDA), which is described in Appendix C, “Appendix C. SQL Descriptor Area (SQLDA)” on page 763. Before the DESCRIBE statement is executed, the following variables in the SQLDA must be set:

**SQLN**            Indicates the number of variables represented by SQLVAR. (SQLN provides the dimension of the SQLVAR array.) SQLN must be set to a value greater than or equal to zero before the DESCRIBE statement is executed.

When the DESCRIBE statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

**SQLDAID**        The first 6 bytes are set to 'SQLDA ' (that is, 5 letters followed by the space character).  
The seventh byte, called SQLDOUBLED, is set to '2' if the SQLDA contains two SQLVAR entries for every select-list item (or, *column* of the result table). This technique is used in order to accommodate LOB and/or distinct type result columns. Otherwise, SQLDOUBLED is set to the space character.  
The doubled flag is set to space if there is not enough room in the SQLDA to contain the entire DESCRIBE reply.  
The eighth byte is set to the space character.

## DESCRIBE

SQLDABC	Length of the SQLDA.
SQLD	If the prepared statement is a SELECT, the number of columns in its result table; otherwise, 0.
SQLVAR	<p>If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.</p> <p>If the value is <math>n</math>, where <math>n</math> is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first <math>n</math> occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column of the result table, the second occurrence of SQLVAR contains a description of the second column of the result table, and so on. The description of a column consists of the values assigned to SQLTYPE, SQLLEN, SQLNAME, SQLLONGLEN, and SQLDATATYPE_NAME.</p>

### *Basic SQLVAR*

SQLTYPE	A code showing the data type of the column and whether or not it can contain null values.
SQLLEN	A length value depending on the data type of the result columns. SQLLEN is 0 for LOB data types.
SQLNAME	If the derived column is not a simple column reference, then sqlname contains an ASCII numeric literal value, which represents the derived column's original position within the select list; otherwise, sqlname contains the name of the column.

### *Secondary SQLVAR*

These variables are only used if the number of SQLVAR entries are doubled to accommodate LOB or distinct type columns.

### SQLLONGLEN

The length attribute of a BLOB, CLOB, or DBCLOB column.

### SQLDATATYPE\_NAME

For a distinct type column, the database manager sets this to the fully qualified distinct type name. Otherwise, schema name is SYSIBM and the low order portion of the name is the name in the TYPENAME column of the SYSCAT.DATATYPES catalog view.

## Notes

- Before the DESCRIBE statement is executed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. To obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR must not be less than the number of columns.

## DESCRIBE

- If a large object is expected, that is, a LOB value greater than 1 megabyte, then remember that manipulating this large object will affect application memory. Given this condition, consider using locators or file reference variables. Modify the SQLDA after the DESCRIBE statement is executed but prior to allocating storage so that an SQLTYPE of SQL\_TYP\_xLOB is changed to SQL\_TYP\_xLOB\_LOCATOR or SQL\_TYP\_xLOB\_FILE with corresponding changes to other fields such as SQLLEN. Then allocate storage based on SQLTYPE and continue.

See the *Embedded SQL Programming Guide* for more information on using locators and file reference variables with the SQLDA.

- Code page conversions between extended Unix code (EUC) code pages and DBCS code pages can result in the expansion and contraction of character lengths. See the *Embedded SQL Programming Guide* for information on handling such situations.
- **Allocating the SQLDA:** Among the possible ways to allocate the SQLDA are the three described below.

*First Technique:* Allocate an SQLDA with enough occurrences of SQLVAR to accommodate any select list that the application will have to process. If the table contains any LOB or distinct type columns, the number of SQLVARs should be double the maximum number of columns; otherwise the number should be the same as the maximum number of columns. Having done the allocation, the application can use this 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. The value returned for SQLD is the number of columns in the result table. This is either the required number of occurrences of SQLVAR or half the required number. Because there were no SQLVAR entries, a warning with SQLSTATE 01005 will be issued. If the SQLCODE accompanying that warning is equal to one of +237, +238 or +239, the number of SQLVAR entries should be double the value returned in SQLD.<sup>73</sup>
2. Allocate an SQLDA with enough occurrences of SQLVAR. Then execute the DESCRIBE statement again, using this 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, and perhaps all, select lists but is also reasonably small. Execute DESCRIBE and

---

<sup>73</sup> The return of these positive SQLCODES assumes that the SQLWARN bind option setting was YES (return positive SQLCODES). If SQLWARN was set to NO, +238 is still returned to indicate that the number of SQLVAR entries must be double the value returned in SQLD.

## DESCRIBE

check the SQLD value. Use the SQLD value for the number of occurrences of SQLVAR to allocate a larger SQLDA, if necessary.

This technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

### Example

In a C 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.

```
EXEC SQL BEGIN DECLARE SECTION;
char stmt1_str[200];
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 */
    /* and allocate buffers to receive the data */
EXEC SQL OPEN DYN_CURSOR;

... /* loop to fetch rows from result table */
EXEC SQL FETCH DYN_CURSOR USING DESCRIPTOR :sqlda;
.
.
.
```

## DISCONNECT

---

### DISCONNECT

The DISCONNECT statement destroys one or more connections when there is no active unit of work (that is, after a commit or rollback operation).<sup>74</sup>

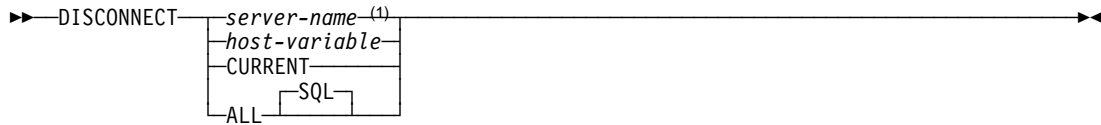
#### Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

#### Authorization

None Required.

#### Syntax



**Note:**

<sup>1</sup> Note that an application server named CURRENT or ALL can only be identified by a host variable.

#### Description

*server-name* or *host-variable*

Identifies the application server by the specified *server-name* or a *host-variable* which contains the server-name.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The server-name that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The specified database-alias or the database-alias contained in the host variable must identify an existing connection of the application process. If the database-alias does not identify an existing connection, an error (SQLSTATE 08003) is raised.

---

<sup>74</sup> If a single connection is the target of the DISCONNECT statement, then the connection is destroyed only if the database has participated in any existing unit of work, not whether there is an active unit of work. For example, if several other databases have done work but the target in question has not, it can still be disconnected without destroying the connection.

## DISCONNECT

### CURRENT

Identifies the current connection of the application process. The application process must be in the connected state. If not, an error (SQLSTATE 08003) is raised.

### ALL

Indicates that all existing connections of the application process are to be destroyed. An error or warning does not occur if no connections exist when the statement is executed. The optional keyword SQL is included to be consistent with the syntax of the RELEASE statement.

## Rules

- Generally, the DISCONNECT statement cannot be executed while within a unit of work. If attempted, an error (SQLSTATE 25000) is raised. The exception to this rule is if a single connection is specified to be disconnected and the database has not participated in an existing unit of work. In this case, it does not matter if there is an active unit of work when the DISCONNECT statement is issued.
- The DISCONNECT statement cannot be executed at all in the Transaction Processing (TP) Monitor environment (SQLSTATE 25000). It is used when the SYNCPOINT precompiler option is set to TWOPHASE.

## Notes

- If the DISCONNECT statement is successful, each identified connection is destroyed.  
  
If the DISCONNECT statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.
- If DISCONNECT is used to destroy the current connection, the next executed SQL statement should be CONNECT or SET CONNECTION.
- Type 1 CONNECT semantics do not preclude the use of DISCONNECT. However, though DISCONNECT CURRENT and DISCONNECT ALL can be used, they will not result in a commit operation like a CONNECT RESET statement would do.  
  
If *server-name* or *host-variable* is specified in the DISCONNECT statement, it must identify the current connection because Type 1 CONNECT only supports one connection at a time. Generally, DISCONNECT will fail if within a unit of work with the exception noted in “Rules”.
- Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be destroyed as soon as possible.
- Connections can also be destroyed during a commit operation because the connection option is in effect. The connection option could be AUTOMATIC, CONDITIONAL, or EXPLICIT, which can be set as a precompiler option or through the SET CLIENT API at run time. See “Options that Govern Distributed Unit of Work Semantics” on page 33 for information about the specification of the DISCONNECT option.

## DISCONNECT

### Examples

*Example 1:* The SQL connection to IBMSTHDB is no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy the connection.

```
EXEC SQL DISCONNECT IBMSTHDB;
```

*Example 2:* The current connection is no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy the connection.

```
EXEC SQL DISCONNECT CURRENT;
```

*Example 3:* The existing connections are no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy all the connections.

```
EXEC SQL DISCONNECT ALL;
```



---

**DROP**

The DROP statement deletes an object. Any objects that are directly or indirectly dependent on that object are either deleted or made inoperative. (See Inoperative Trigger on page 574 and Inoperative views on page 590 for details.) Whenever an object is deleted, its description is deleted from the catalog and any packages that reference the object are invalidated.

**Invocation**

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

**Authorization**

The privileges that must be held by the authorization ID of the DROP statement when dropping objects that allow two-part names must include one of the following:

- SYSADM or DBADM authority
- DROPIN privilege on the schema for the object
- definer of the object as recorded in the DEFINER column of the catalog view for the object
- CONTROL privilege on the object (applicable to index, package, table and view objects only).

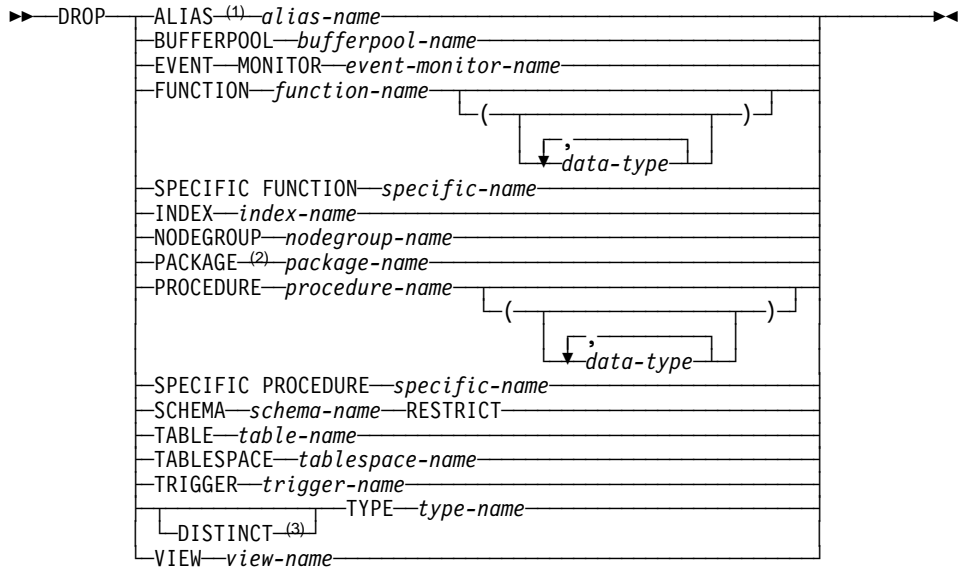
The authorization ID of the DROP statement when dropping a schema must have SYSADM or DBADM authority or be the schema owner as recorded in the OWNER column of SYSCAT.SCHEMATA.

The authorization ID of the DROP statement when dropping a buffer pool, nodegroup, or table space must have SYSADM or SYSCTRL authority.

The authorization ID of the DROP statement when dropping an event monitor must have SYSADM or DBADM authority

**Syntax**

# DROP



## Notes:

- 1 SYNONYM can be used as a synonym for ALIAS.
- 2 PROGRAM can be used as a synonym for PACKAGE.
- 3 DATA can also be used when dropping any user-defined type.

## Description

### ALIAS *alias-name*

Identifies the alias that is to be dropped. The *alias-name* must identify an alias that is described in the catalog (SQLSTATE 42704). The specified alias is deleted.

All tables, views and triggers<sup>75</sup> that reference the alias are made inoperative.

### BUFFERPOOL *bufferpool-name*

Identifies the buffer pool that is to be dropped. The *bufferpool-name* must identify a buffer pool that is described in the catalog (SQLSTATE 42704). There can be no table spaces assigned to the buffer pool (SQLSTATE 42893). The IBMDEFAULTBP buffer pool cannot be dropped (SQLSTATE 42832). The storage for the buffer pool will not be released until the database is stopped.

### EVENT MONITOR *event-monitor-name*

Identifies the event monitor that is to be dropped. The *event-monitor-name* must identify an event monitor that is described in the catalog (SQLSTATE 42704).

If the identified event monitor is ON, an error (SQLSTATE 55034) is raised. Otherwise, the event monitor is deleted.

<sup>75</sup> This includes both the table referenced in the ON clause of the CREATE TRIGGER statement and all tables referenced within the triggered SQL statements.

If there are event files in the target path of the event monitor when the event monitor is dropped, the event files are not deleted. However, if a new event monitor is created which specifies the same target path, then the event files are deleted.

## FUNCTION

Identifies an instance of a user-defined function that is to be dropped. The function instance specified must be a user-defined function described in the catalog. Functions implicitly generated by the CREATE DISTINCT TYPE statement cannot be dropped.

There are several different ways available to identify the function instance:

### FUNCTION *function-name*

Identifies the particular function, and is valid only if there is exactly one function instance with the *function-name*. The function thus identified may have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the function in the named or implied schema, an error (SQLSTATE 42854) is raised.

### FUNCTION *function-name (data-type,...)*

Provides the function signature, which uniquely identifies the function to be dropped. The function selection algorithm is **not** used.

#### *function-name*

Gives the function name of the function to be dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

#### *(data-type,...)*

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 is used to identify the specific function instance which is to be dropped.

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead, an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

## DROP

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE PROCEDURE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n* since  $0 < n < 25$  means REAL and  $24 < n < 54$  means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no function with the specified signature exists in named or implied schema, an error (SQLSTATE 42883) is raised.

### **SPECIFIC FUNCTION** *specific-name*

Identifies the particular user-defined function that is to be dropped, using the specific name either specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

It is not possible to drop a function that is in either the SYSIBM schema or the SYSFUN schema (SQLSTATE 42832).

Other objects can be dependent upon a function. All such dependencies must be removed before the function can be dropped, with the exception of packages which are marked inoperative. An attempt to drop a function with such dependencies will result in an error (SQLSTATE 42893). See page 618 for a list of these dependencies.

If the function can be dropped, it is dropped.

Any package dependent on the specific function being dropped is marked as inoperative. Such a package is not implicitly rebound. It must either be rebound by use of the BIND or REBIND command or it must be reprepared by use of the PREP command. See the *Command Reference* for information on these commands.

### **INDEX** *index-name*

Identifies the index that is to be dropped. The *index-name* must identify an index that is described in the catalog (SQLSTATE 42704). It cannot be an index required by the system for a primary key or unique constraint or for a replicated summary table (SQLSTATE 42917). The specified index is deleted.

Packages having a dependency on a dropped index will be invalidated.

### **NODEGROUP** *nodegroup-name*

Identifies the nodegroup that is to be dropped. *nodegroup-name* must identify a nodegroup that is described in the catalog (SQLSTATE 42704). This is a one-part name.

Dropping a nodegroup drops all table spaces defined in the nodegroup. All existing database objects with dependencies on the tables in the table spaces (such as packages, referential constraints, etc.) are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.

System defined nodegroups cannot be dropped (SQLSTATE 42832).

If a DROP NODEGROUP is issued against a nodegroup that is currently undergoing a data redistribution, the DROP NODEGROUP operation fails an error is returned (SQLSTATE 55038) . However, a partially redistributed nodegroup can be dropped. A nodegroup can become partially redistributed if a REDISTRIBUTE NODEGROUP command does not execute to completion. This can happen if it gets interrupted by either an error or a force application all command<sup>76</sup>.

**PACKAGE** *package-name*

Identifies the package that is to be dropped. The *package-name* must identify a package that is described in the catalog (SQLSTATE 42704). The specified package is deleted. All privileges on the package are also deleted.

**PROCEDURE**

Identifies an instance of a stored procedure that is to be dropped. The procedure instance specified must be a stored procedure described in the catalog.

There are several different ways available to identify the procedure instance:

**PROCEDURE** *procedure-name*

Identifies the particular procedure, and is valid only if there is exactly one procedure instance with the *procedure-name* in the schema . The procedure thus identified may have any number of parameters defined for it. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifer for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If there is more than one specific instance of the procedure in the named or implied schema, an error (SQLSTATE 42854) is raised.

**PROCEDURE** *procedure-name (data-type,...)*

Provides the procedure signature, which uniquely identifies the procedure to be dropped. The procedure selection algorithm is **not** used.

*procedure-name*

Gives the procedure name of the procedure to be dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

*(data-type,...)*

Must match the data types that were specified on the CREATE PROCEDURE statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be dropped.

<sup>76</sup> For a partially redistributed nodegroup, the REBALANCE\_PMAP\_ID in the SYSCAT.NODEGROUPS catalog is not -1.

## DROP

If the *data-type* is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead, an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(n) does not need to match the defined value for n since  $0 < n < 25$  means REAL and  $24 < n < 54$  means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in named or implied schema, an error (SQLSTATE 42883) is raised.

### **SPECIFIC PROCEDURE** *specific-name*

Identifies the particular stored procedure that is to be dropped, using the specific name either specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

### **SCHEMA** *schema-name* **RESTRICT**

Identifies the schema that is to be dropped. The *schema-name* must identify a schema that is described in the catalog (SQLSTATE 42704). The RESTRICT keyword enforces the rule that no objects can be defined in the specified schema for the schema to be deleted from the database (SQLSTATE 42893) .

### **TABLE** *table-name*

Identifies the base table or summary table that is to be dropped. The *table-name* must identify a table that is described in the catalog (SQLSTATE 42704). The subtables of a typed table are dependent on their supertables. All subtables must be dropped before a supertable can be dropped (SQLSTATE 42893). The specified table is deleted from the database.

All indexes, primary keys, foreign keys, and check constraints referencing the table are dropped. All views and triggers<sup>77</sup> that reference the table are made inoperative. All packages depending on any object dropped or marked inoperative will be invali-

---

<sup>77</sup> This includes both the table referenced in the ON clause of the CREATE TRIGGER statement and all tables referenced within the triggered SQL statements.

dated. This includes packages dependent on any supertables above the subtable in the hierarchy. Any reference columns for which the dropped table is defined as the scope of the reference become unscoped.

All files that are linked through any DATALINK columns are unlinked. The unlink operation is performed asynchronously so the files may not be immediately available for other operations.

When a subtable is dropped from a table hierarchy, the columns associated with the subtable are no longer accessible although they continue to be considered with respect to limits on the number of columns and size of the row.

**TABLESPACE** *tablespace-name*

Identifies the table space that is to be dropped. *tablespace-name* must identify a table space that is described in the catalog (SQLSTATE 42704). This is a one-part name.

The table space will not be dropped (SQLSTATE 55024) if there is any table that stores at least one of its parts in this table space and has one or more of its parts in another table space (these tables would need to be dropped first). System table spaces cannot be dropped (SQLSTATE 42832). A temporary table space can not be dropped (SQLSTATE 55026) if it is the only temporary table space that exists in the database.

Dropping a table space drops all objects defined in the table space. All existing database objects with dependencies on the table space, such as packages, referential constraints, etc. are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.

Containers created by the user are not deleted. Any directories in the path of the container name that were created by the database manager on CREATE TABLESPACE will be deleted. All containers that are below the database directory are deleted.

**TRIGGER** *trigger-name*

Identifies the trigger that is to be dropped. The *trigger-name* must identify a trigger that is described in the catalog (SQLSTATE 42704). The specified trigger is deleted.

Dropping triggers causes certain packages to be marked invalid. See the “Notes” section in “CREATE TRIGGER” on page 568 concerning the creation of triggers (which follows the same rules).

**TYPE** *type-name*

Identifies the user-defined type to be dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. For a structured type, the associated reference type is also dropped. The *type-name* must identify a user-defined type described in the catalog. If DISTINCT is specified, then the *type-name* must identify a distinct type described in the catalog. The type is not dropped (SQLSTATE 42893) if any of the following are true.

## DROP

- The type is a distinct type used as the type of a column of a table or view.
- The type has a subtype.
- The type is a structured type used as a data type of a typed table.
- The type is used as the target type of a reference (REF) type.
- The type or a reference to the type is a parameter type or a return value type of a function that cannot be dropped.
- The type is used in a check constraint, trigger or view definition.

If the user-defined type can be dropped, then for every function, F, that has parameters or a return value of the type being dropped or a reference to the type being dropped, the following DROP FUNCTION statement is effectively executed:

**DROP FUNCTION F**

It is possible that this statement also would cascade to drop dependent functions. If all of these functions are also in the list to be dropped because of a dependency on the user-defined type, the drop of the user-defined type will succeed (otherwise it fails with SQLSTATE 42893).

### **VIEW** *view-name*

Identifies the view that is to be dropped. The *view-name* must identify a view that is described in the catalog (SQLSTATE 42704). The subviews of a typed view are dependent on their superviews. All subviews must be dropped before a superview can be dropped (SQLSTATE 42893).

The specified view is deleted. The definition of any view or trigger that is directly or indirectly dependent on that view is marked inoperative. Any packages dependent on a view that is dropped or marked inoperative will be invalidated. This includes packages dependent on any superviews above the subview in the hierarchy. Any reference columns for which the dropped view is defined as the scope of the reference become unscoped.

## Rules

- **Dependencies:** Table 21 on page 619 shows the dependencies<sup>78</sup> that objects have on each other. Four different types of dependencies are shown:
  - R Restrict semantics. The underlying object cannot be dropped as long as the object that depends on it exists.
  - C Cascade semantics. Dropping the underlying object causes the object that depends on it (the depending object) to be dropped as well. However, if the depending object cannot be dropped because it has a Restrict dependency on some other object, the drop of the underlying object will fail.
  - X Inoperative semantics. Dropping the underlying object causes the object that depends on it to become inoperative. It remains inoperative until a user takes some explicit action.

---

<sup>78</sup> Not all dependencies are explicitly recorded in the catalog. For example, there is no record of which constraints a package has a dependency on.



## DROP

- A Automatic Invalidation/Revalidation semantics. Dropping the underlying object causes the object that depends on it to become invalid. The database manager attempts to revalidate the invalid object.

Table 21. Dependencies

Object Type →	B U F F E R P O O L S		C O N S T R A I N T S		F U N C T I O N S		P A C K A G E S		P A R T I T I O N I N G K E Y S		P R O C E D U R E S		N O D E G R O U P S		T A B L E S		T R I G G E R S		V I E W S		
Statement ↓	A	R	R	T	T	I	A	G	I	D	E	U	R	O	A	B	A	G	E	R	W
DROP ALIAS	-	-	-	-	-	-	A <sup>3</sup>	-	-	-	-	-	R <sup>3</sup>	-	X <sup>3</sup>	X <sup>3</sup>					
DROP BUFFERPOOL	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	-	-				
ALTER TABLE DROP CONSTRAINT	-	-	C	-	-	-	A <sup>1</sup>	-	-	-	-	-	-	-	-	-	-				
DROP FUNCTION	-	-	R	-	R <sup>7</sup>	-	X	-	-	-	-	-	R	-	R	R					
DROP INDEX	-	-	R	-	-	-	A	-	-	-	-	-	R <sup>19</sup>	-	-	R <sup>17</sup>					
DROP PACKAGE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-				
ALTER TABLE DROP PARTITIONING KEY	-	-	-	-	-	-	A <sup>1</sup>	-	-	-	-	-	-	-	-	-	-				
REVOKE a privilege <sup>10</sup>	-	-	-	-	-	-	A <sup>1</sup>	-	-	-	-	-	X <sup>8</sup>	-	X	X <sup>8</sup>					
DROP PROCEDURE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-				
DROP NODEGROUP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	C	-	-				
DROP TABLE	-	-	C	-	-	C	A <sup>9</sup>	-	-	-	-	-	RX <sup>11</sup>	-	X <sup>16</sup>	X <sup>16</sup>					
DROP TABLESPACE	-	-	-	-	-	C <sup>6</sup>	-	-	-	-	-	-	CR <sup>6</sup>	-	-	-					
DROP TRIGGER	-	-	-	-	-	-	A <sup>1</sup>	-	-	-	-	-	-	-	-	-					
DROP TYPE	-	-	R <sup>13</sup>	R <sup>4</sup>	C <sup>5</sup>	-	A <sup>12</sup>	-	-	-	-	-	R <sup>18</sup>	-	R <sup>13</sup>	R <sup>14</sup>					
DROP VIEW	-	-	-	-	-	-	A <sup>2</sup>	-	-	-	-	-	-	-	X <sup>16</sup>	X <sup>15</sup>					

<sup>1</sup> This dependency is implicit in depending on a table with these constraints, triggers or a partitioning key.

## DROP

- 2 If a package has an INSERT, UPDATE, or DELETE statement acting upon a view, then the package has an insert, update or delete usage on the underlying base table of the view. In the case of UPDATE, the package has an update usage on each column of the underlying base table that is modified by the UPDATE.

If a package has a statement acting on a typed view, creating or dropping any view in the same view hierarchy will invalidate the package.

- 3 If a package, summary table, view, or trigger uses an alias, it becomes dependent both on the alias and the object that the alias references. If the alias is in a chain, then a dependency is created on each alias in the chain.

Aliases themselves are not dependent on anything. It is possible for an alias to be defined on an object that does not exist.

- 4 A user-defined type can depend on another user-defined type if the depending user-defined type:

- names a user-defined type as the data type of an attribute
- names a user-defined structured type within a reference type as the data type of an attribute
- has a user-defined type as a supertype.

- 5 Dropping a data type cascades to drop the functions that use that data type. Dropping of these functions will not be prevented by the fact that they depend on each other.

- 6 Dropping a table space causes all tables that are completely contained in the table space to be dropped. However, if a table spans table spaces (indexes or long columns in different table spaces) none of those table spaces can be dropped as long as the table exists.

- 7 A function can depend on another specific function if the depending function names the base function in a SOURCE clause.

- 8 Only loss of SELECT privilege will cause a summary table or view to become inoperative. If the view that is made inoperative is included in a typed view hierarchy, all of its subviews also become inoperative.

- 9 If a package has an INSERT, UPDATE, or DELETE statement acting on table T, then the package has an insert, update or delete usage on T. In the case of UPDATE, the package has an update usage on each column of T that is modified by the UPDATE.

If a package has a statement acting on a typed table, creating or dropping any table in the same table hierarchy will invalidate the package.

- 10 Dependencies do not exist at the column level because privileges on columns cannot be revoked individually.

If a package, trigger or view includes the use of OUTER(Z) in the FROM clause, there is a dependency on the SELECT privilege on every subtable or subview of Z. Similarly, if a package, trigger, or view includes the use of

- DEREF(*Y*) where *Y* is a reference type with a target table or view *Z*, there is a dependency on the SELECT privilege on every subtable or subview of *Z*.
- 11 A summary table is dependent on the underlying table or tables specified in the fullselect of the table definition.  
A subtable is dependent on its supertable up to the root table. A supertable cannot be dropped until all its subtables are dropped.
  - 12 A package can depend on structured types as a result of using the TYPE predicate. The package has a dependency on the subtypes of each structured type specified in the right side of the TYPE predicate. Dropping or creating a structured type that alters the subtypes on which the package is dependent causes invalidation.
  - 13 A check constraint or trigger is dependent on a type if the type is used anywhere in the constraint or trigger. There is no dependency on the subtypes of a structured type used in a TYPE predicate within a check constraint or trigger.
  - 14 A view is dependent on a type if the type is used anywhere in the view definition (this includes the type of typed view). There is no dependency on the subtypes of a structured type used in a TYPE predicate within a view definition.
  - 15 A subview is dependent on its superview up to the root view. A superview cannot be dropped until all its subviews are dropped. Refer to <sup>16</sup> for additional view dependencies.
  - 16 A trigger or view is also dependent on the target table or target view of a dereference operation or Deref function. A trigger or view with a FROM clause that includes OUTER(*Z*) is dependent on all the subtables or subviews of *Z* that existed at the time the trigger or view was created.
  - 17 A typed view can depend on the existence of a unique index to ensure the uniqueness of the object identifier column.
  - 18 A table may depend on a user defined data type because:
    - it is used as the type of a column
    - it is used as the type of the table
    - it is used as an attribute of a type of the table
    - it is used as the target type of a reference type that is the type of a column of the table or an attribute of the type of the table.
  - 19 A replicated summary table depends on the existence of a unique index.

## Notes

- It is valid to drop a user-defined function while it is in use. Also, a cursor can be open over a statement which contains a reference to a user-defined function, and while this cursor is open the function can be dropped without causing the cursor fetches to fail.
- If a package which depends on a user-defined function is executing, it is not possible for another authorization ID to drop the function until the package completes

## DROP

its current unit of work. At that point, the function is dropped and the package becomes inoperative. The next request for this package results in an error indicating that the package must be explicitly rebound.

- The removal of a function body (this is very different from dropping the function) can occur while an application which needs the function body is executing. This may or may not cause the statement to fail, depending on whether the function body still needs to be loaded into storage by the database manager on behalf of the statement.
- For any dropped table that includes currently linked files through DATALINK columns, the files are unlinked, and will be either restored or deleted, depending on the datalink column definition.
- If a table containing a DATALINK column is dropped while any DB2 File Managers configured to the database are unavailable, either through DROP TABLE or DROP TABLESPACE, then the operation will fail (SQLSTATE 57050).

## Examples

*Example 1:* Drop table TDEPT.

```
DROP TABLE TDEPT
```

*Example 2:* Drop the view VDEPT.

```
DROP VIEW VDEPT
```

*Example 3:* The authorization ID HEDGES attempts to drop an alias.

```
DROP ALIAS A1
```

The alias HEDGES.A1 is removed from the catalogs.

*Example 4:* Hedges attempts to drop an alias, but specifies T1 as the alias-name, where T1 is the name of an existing table (not the name of an alias).

```
DROP ALIAS T1
```

This statement fails (SQLSTATE 42809).

*Example 5:*

Drop the BUSINESS\_OPS nodegroup. To drop the nodegroup, the two table spaces (ACCOUNTING and PLANS) in the nodegroup must first be dropped.

```
DROP TABLESPACE ACCOUNTING  
DROP TABLESPACE PLANS  
DROP NODEGROUP BUSINESS_OPS
```

*Example 6:* Pellow wants to drop the CENTRE function, which he created in his PELLOW schema, using the signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTRE (INT,FLOAT)
```

## DROP

*Example 7:* McBride wants to drop the FOCUS92 function, which she created in the PELLOW schema, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION PELLOW.FOCUS92
```

*Example 8:* Drop the function ATOMIC\_WEIGHT from the CHEM schema, where it is known that there is only one function with that name.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT
```

*Example 9:* Drop the trigger SALARY\_BONUS, which caused employees under a specified condition to receive a bonus to their salary.

```
DROP TRIGGER SALARY_BONUS
```

*Example 10:* Drop the distinct data type named shoesize, if it is not currently in use.

```
DROP DISTINCT TYPE SHOESIZE
```

*Example 11:* Drop the SMITHPAY event monitor.

```
DROP EVENT MONITOR SMITHPAY
```

*Example 12:* Drop the schema from Example 2 under CREATE SCHEMA using RESTRICT. Notice that the table called PART must be dropped first.

```
DROP TABLE PART
```

```
DROP SCHEMA INVENTORY RESTRICT
```

*Example 13:* Macdonald wants to drop the DESTROY procedure, which he created in the EIGLER schema, using the specific name to identify the procedure instance to be dropped.

```
DROP SPECIFIC PROCEDURE EIGLER.DESTROY
```

*Example 14:* Drop the procedure OSMOSIS from the BIOLOGY schema, where it is known that there is only one procedure with that name.

```
DROP PROCEDURE BIOLOGY.OSMOSIS
```

## END DECLARE SECTION

---

### 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. It must not be specified in REXX.

#### Authorization

None required.

#### Syntax



►—END DECLARE SECTION—◄

The diagram shows the text 'END DECLARE SECTION' enclosed in a rectangular box. A horizontal line with arrowheads at both ends passes through the text, indicating the scope of the statement.

#### Description

The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear according to the rules of the host language. It indicates the end of a host variable declaration section. A host variable section starts with a BEGIN DECLARE SECTION statement (see “BEGIN DECLARE SECTION” on page 407).

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

Host variable declarations can be specified by using the SQL INCLUDE statement. Otherwise, a host variable declaration section must not contain any statements other than host variable declarations.

Host variables referenced in SQL statements must be declared in a host variable declare section in all host languages, other than REXX.<sup>79</sup> Furthermore, the declaration of each variable must appear before the first reference to the variable.

Variables declared outside a declare section must not have the same name as variables declared within a declare section.

---

<sup>79</sup> See “Rules” on page 407 for information on how host variables can be declared in REXX in the case of LOB locators and file reference variables.

**END DECLARE SECTION**

**Example**

See “BEGIN DECLARE SECTION” on page 407 for examples that use the END DECLARE SECTION statement.

## EXECUTE

---

### EXECUTE

The EXECUTE statement executes a prepared SQL statement.

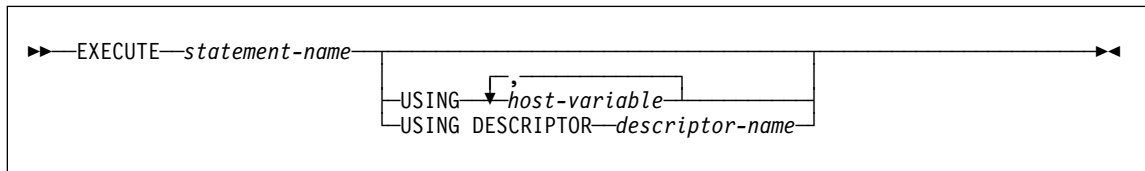
#### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

#### Authorization

For statements where authorization checking is performed at statement execution time (DDL, GRANT, and REVOKE statements), the privileges held by the authorization ID of the statement must include those required to execute the SQL statement specified by the PREPARE statement. For statements where authorization checking is performed at statement preparation time (DML), no authorization is required to use this statement.

#### Syntax



#### Description

##### *statement-name*

Identifies the prepared statement to be executed. The *statement-name* must identify a statement that was previously prepared and the prepared statement must not be a SELECT statement.

##### **USING**

Introduces a list of host variables for which values are substituted for the parameter markers (question marks) in the prepared statement. (For an explanation of parameter markers, see “PREPARE” on page 673.) If the prepared statement includes parameter markers, USING must be used.

##### *host-variable, ...*

Identifies a host variable that is declared in the program in accordance with the rules for declaring host 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. Locator variables and file reference variables, where appropriate, can be provided as the source of values for parameter markers.

##### **DESCRIPTOR** *descriptor-name*

Identifies an input SQLDA that must contain a valid description of host variables.



## EXECUTE

Before the EXECUTE statement is processed, the user must set the following fields in the input SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to  $16 + \text{SQLN} * (N)$ , where N is the length of an SQLVAR occurrence.

If LOB input data needs to be accommodated, there must be two SQLVAR entries for every parameter marker.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. For more information, see Appendix C, "Appendix C. SQL Descriptor Area (SQLDA)" on page 763.

## Notes

- Before the prepared statement is executed, each parameter marker is effectively replaced by the value of its corresponding host variable. 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. See "Rules" on page 674 for the rules affecting parameter markers.

Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:

- 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 with two blanks.

- **Dynamic SQL Statement Caching:**

The information required to execute dynamic and static SQL statements is placed in the database package cache when static SQL statements are first referenced or

## EXECUTE

when dynamic SQL statements are first prepared. This information stays in the package cache until it becomes invalid, the cache space is required for another statement, or the database is shut down.

When an SQL statement is executed or prepared, the package information relevant to the application issuing the request is loaded from the system catalog into the package cache. The actual executable section for the individual SQL statement is also placed into the cache: static SQL sections are read in from the system catalog and placed in the package cache when the statement is first referenced; Dynamic SQL sections are placed directly in the cache after they have been created. Dynamic SQL sections can be created by an explicit statement, such as a PREPARE or EXECUTE IMMEDIATE statement. Once created, sections for dynamic SQL statements may be recreated by an implicit prepare of the statement performed by the system if the original section has been deleted for space management reasons or has become invalid due to changes in the environment.

Each SQL statement is cached at a database level and can be shared among applications. Static SQL statements are shared among applications using the same package; Dynamic SQL statements are shared among applications using the same compilation environment and the exact same statement text. The text of each SQL statement issued by an application is cached locally within the application for use in the event that an implicit prepare is required. Each PREPARE statement in the application program can cache one statement. All EXECUTE IMMEDIATE statements in an application program share the same space and only one cached statement exists for all these EXECUTE IMMEDIATE statements at a time. If the same PREPARE or any EXECUTE IMMEDIATE statement is issued multiple times with a different SQL statement each time, only the last statement will be cached for reuse. The optimal use of the cache is to issue a number of different PREPARE statements once at the start of the application and then to issue an EXECUTE or OPEN statement as required.

With the caching of dynamic SQL statements, once a statement has been created, it can be reused over multiple units of work without the need to prepare the statement again. The system will recompile the statement as required if environment changes occur.

The following events are examples of environment or data object changes which can cause cached dynamic statements to be implicitly prepared on the next PREPARE, EXECUTE, EXECUTE IMMEDIATE, or OPEN request:

- ALTER TABLE
- ALTER TABLESPACE
- CREATE FUNCTION
- CREATE INDEX
- CREATE TABLE
- CREATE TEMPORARY TABLESPACE
- CREATE TRIGGER
- DROP (all objects)
- RUNSTATS on any table or index
- any action that causes a view to become inoperative
- UPDATE of statistics in any system catalog table

## EXECUTE

- SET CURRENT DEGREE
- SET FUNCTION PATH
- SET QUERY OPTIMIZATION

The following list outlines the behavior that can be expected from cached dynamic SQL statements:

- *PREPARE Requests*: Subsequent preparations of the same statement will not incur the cost of compiling the statement if the section is still valid. The cost and cardinality estimates for the current cached section will be returned. These values may differ from the values returned from any previous PREPARE for the same SQL statement.

There will be no need to issue a PREPARE statement subsequent to a COMMIT or ROLLBACK statement.

- *EXECUTE Requests*: EXECUTE statements may occasionally incur the cost of implicitly preparing the statement if it has become invalid since the original PREPARE. If a section is implicitly prepared, it will use the current environment and not the environment of the original PREPARE statement.
- *EXECUTE IMMEDIATE Requests*: Subsequent EXECUTE IMMEDIATE statements for the same statement will not incur the cost of compiling the statement if the section is still valid.
- *OPEN Requests*: OPEN requests for dynamically defined cursors may occasionally incur the cost of implicitly preparing the statement if it has become invalid since the original PREPARE statement. If a section is implicitly prepared, it will use the current environment and not the environment of the original PREPARE statement.
- *FETCH Requests*: No behavior changes should be expected.
- *ROLLBACK*: Only those dynamic SQL statements prepared or implicitly prepared during the unit of work affected by the rollback operation will be invalidated.
- *COMMIT*: Dynamic SQL statements will not be invalidated but any locks acquired will be freed. Cursors not defined as WITH HOLD cursors will be closed and their locks freed. Open WITH HOLD cursors will hold onto their package and section locks to protect the active section during, and after, commit processing.

If an error occurs during an implicit prepare, an error will be returned for the request causing the implicit prepare (SQLSTATE 56098).

## Examples

*Example 1:* In this C example, an INSERT statement with parameter markers is prepared and executed. h1 - h4 are host variables that correspond to the format of TDEPT.

## EXECUTE

```
strcpy (s,"INSERT INTO TDEPT VALUES(?,?,?,?)");  
EXEC SQL PREPARE DEPT_INSERT FROM :s;  
.  
.  
(Check for successful execution and put values into :h1, :h2, :h3, :h4)  
.  
.  
EXEC SQL EXECUTE DEPT_INSERT USING :h1, :h2,  
:h3, :h4;
```

*Example 2:* This EXECUTE statement uses an SQLDA.

```
EXECUTE S3 USING DESCRIPTOR :sqlda3
```

---

## 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.

EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. It can be used to prepare and execute SQL statements that contain neither host variables nor parameter markers.

### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

The authorization rules are those defined for the SQL statement specified by EXECUTE IMMEDIATE.

### Syntax

►► EXECUTE IMMEDIATE *host-variable* ◀◀

### Description

*host-variable*

A host variable must be specified and it must identify a host variable that is described in the program in accordance with the rules for declaring character-string variables. It must be a character-string variable (either fixed-length or varying-length) and if it is a CLOB it must be less than 32 765 bytes.

The value of the identified host variable is called the statement string.

The statement string must be one of the following SQL statements:

- ALTER
- COMMENT ON
- COMMIT
- CREATE
- DELETE
- DROP
- GRANT
- INSERT
- LOCK TABLE
- REFRESH TABLE

## EXECUTE IMMEDIATE

- RENAME TABLE
- REVOKE
- ROLLBACK
- SET CONSTRAINTS
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET EVENT MONITOR STATE
- SET PATH
- SET SCHEMA
- SIGNAL SQLSTATE
- UPDATE

The statement string must not include parameter markers or references to host variables, and must not begin with EXEC SQL. It must not contain a statement terminator with the exception of the CREATE TRIGGER statement which can contain a semi-colon (;) to separate triggered SQL statements.

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.

### Notes

- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement. See Dynamic SQL Statement Caching on page 627 for information.

### Example

Use C program statements to move an SQL statement to the host variable qstring (char[80]) and prepare and execute whatever SQL statement is in the host variable qstring.

```
if ( strcmp(accounts,"BIG") == 0 )
    strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
            FROM EMP_ACT WHERE ACTNO < 100");
else
    strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
            FROM EMP_ACT WHERE ACTNO >= 100");
.
.
.
EXEC SQL EXECUTE IMMEDIATE :qstring;
```

## EXPLAIN

The EXPLAIN statement captures information about the access plan chosen for the supplied explainable statement and places this information into the Explain tables. (See Appendix K, “ Explain Tables and Definitions” on page 935 for information on the Explain tables and table definitions.)

An *explainable statement* is a DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES, or VALUES INTO SQL statement.

### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

The statement to be explained is not executed.

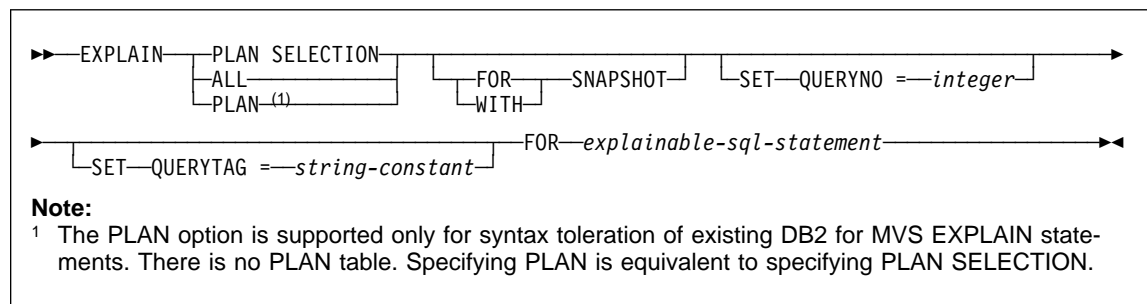
### Authorization

The authorization rules are those defined for the SQL statement specified in the EXPLAIN statement. For example, if a DELETE statement was used as the *explainable-sql-statement* (see statement syntax that follows), then the authorization rules for a DELETE statement would be applied when the DELETE statement is explained.

The authorization rules for static EXPLAIN statements are those rules that apply for static versions of the statement passed as the *explainable-sql-statement*. Dynamically prepared EXPLAIN statements use the authorization rules for the dynamic preparation of the statement provided for the *explainable-sql-statement* parameter.

The current authorization ID must have insert privilege on the Explain tables.

### Syntax



# EXPLAIN

## Description

### PLAN SELECTION

Indicates that the information from the plan selection phase of SQL compilation is to be inserted into the Explain tables.

### ALL

Specifying ALL is equivalent to specifying PLAN SELECTION.

### PLAN

The PLAN option provides syntax toleration for existing database applications from other systems. Specifying PLAN is equivalent to specifying PLAN SELECTION.

### FOR SNAPSHOT

This clause indicates that only an Explain Snapshot is to be taken and placed into the SNAPSHOT column of the EXPLAIN\_STATEMENT table. No other Explain information is captured other than that present in the EXPLAIN\_INSTANCE and EXPLAIN\_STATEMENT tables.

The Explain Snapshot information is intended for use with Visual Explain.

### WITH SNAPSHOT

This clause indicates that, in addition to the regular Explain information, an Explain Snapshot is to be taken.

The default behavior of the EXPLAIN statement is to only gather regular Explain information and not the Explain Snapshot.

The Explain Snapshot information is intended for use with Visual Explain.

default (neither FOR SNAPSHOT nor WITH SNAPSHOT specified)

Puts Explain information into the Explain tables. No snapshot is taken for use with Visual Explain.

### SET QUERYNO = *integer*

Associates *integer*, via the QUERYNO column in the EXPLAIN\_STATEMENT table, with *explainable-sql-statement*. The integer value supplied must be a positive value.

If this clause is not specified for a dynamic EXPLAIN statement, a default value of one (1) is assigned. For a static EXPLAIN statement, the default value assigned is the statement number assigned by the precompiler.

### SET QUERYTAG = *string-constant*

Associates *string-constant*, via the QUERYTAG column in the EXPLAIN\_STATEMENT table, with *explainable-sql-statement*. *string-constant* can be any character string up to 20 bytes in length. If the value supplied is less than 20 bytes in length, the value is padded on the right with blanks to the required length.

If this clause is not specified for an EXPLAIN statement, blanks are used as the default value.



## EXPLAIN

### FOR *explainable-sql-statement*

Specifies the SQL statement to be explained. This statement can be any valid DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES, or VALUES INTO SQL statement. If the EXPLAIN statement is embedded in a program, the *explainable-sql-statement* can contain references to host variables (these variables must be defined in the program). Similarly, if EXPLAIN is being dynamically prepared, the *explainable-sql-statement* can contain parameter markers.

The *explainable-sql-statement* must be a valid SQL statement that could be prepared and executed independently of the EXPLAIN statement. It cannot be a statement name or host variable. SQL statements referring to cursors defined through CLP are not valid for use with this statement.

To explain dynamic SQL within an application, the entire EXPLAIN statement must be dynamically prepared.

## Notes

The following table shows the interaction of the snapshot keywords and the Explain information.

Keyword Specified	Capture Explain Information?	Take Snapshot for Visual Explain?
none	Yes	No
FOR SNAPSHOT	No	Yes
WITH SNAPSHOT	Yes	Yes

If neither the FOR SNAPSHOT nor WITH SNAPSHOT clause is specified, then no Explain snapshot is taken.

The Explain tables must be created by the user prior to the invocation of EXPLAIN. (See Appendix K, “Explain Tables and Definitions” on page 935 for information on the Explain tables and table definitions.) The information generated by this statement is stored in these explain tables in the schema designated at the time the statement is compiled.

If any errors occur during the compilation of the *explainable-sql-statement* supplied, then no information is stored in the Explain tables.

The access plan generated for the *explainable-sql-statement* is not saved and thus, cannot be invoked at a later time. The Explain information for the *explainable-sql-statement* is inserted when the EXPLAIN statement itself is compiled.

For a static EXPLAIN SQL statement, the information is inserted into the Explain tables at bind time and during an explicit rebind (see REBIND in the *Command Reference*). During precompilation, the static EXPLAIN statements are commented out in the modified application source file. At bind time, the EXPLAIN statements are stored in the SYSCAT.STATEMENTS catalog. When the package is run, the EXPLAIN statement is not executed. Note that the section numbers for all statements in the application will be sequential and will include the EXPLAIN statements. An alternative to using a static

## EXPLAIN

EXPLAIN statement is to use a combination of the EXPLAIN and EXPLSNAP BIND/PREP options. Static EXPLAIN statements can be used to cause the Explain tables to be populated for one specific static SQL statement out of many; simply prefix the target statement with the appropriate EXPLAIN statement syntax and bind the application without using either of the Explain BIND/PREP options. The EXPLAIN statement can also be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual Explain invocation.

For dynamic EXPLAIN statements, the Explain tables are populated at the time the EXPLAIN statement is submitted for compilation. An Explain statement can be prepared with the PREPARE statement but, if executed, will perform no processing (though the statement will be successful (SQLSTATE 00000)). An alternative to issuing dynamic EXPLAIN statements is to use a combination of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special registers to explain dynamic SQL statements. The EXPLAIN statement should be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual Explain invocation.

### Examples

*Example 1:* Explain a simple SELECT statement and tag with QUERYNO = 13.

```
EXPLAIN PLAN SET QUERYNO = 13 FOR SELECT C1 FROM T1;
```

This statement is successful.

*Example 2:*

Explain a simple SELECT statement and tag with QUERYTAG = 'TEST13'.

```
EXPLAIN PLAN SELECTION SET QUERYTAG = 'TEST13'  
FOR SELECT C1 FROM T1;
```

This statement is successful.

*Example 3:* Explain a simple SELECT statement and tag with QUERYNO = 13 and QUERYTAG = 'TEST13'.

```
EXPLAIN PLAN SELECTION SET QUERYNO = 13 SET QUERYTAG = 'TEST13'  
FOR SELECT C1 FROM T1;
```

This statement is successful.

*Example 4:* Attempt to get Explain information when Explain tables do not exist.

```
EXPLAIN ALL FOR SELECT C1 FROM T1;
```

This statement would fail as the Explain tables have not been defined (SQLSTATE 42704).

## 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.

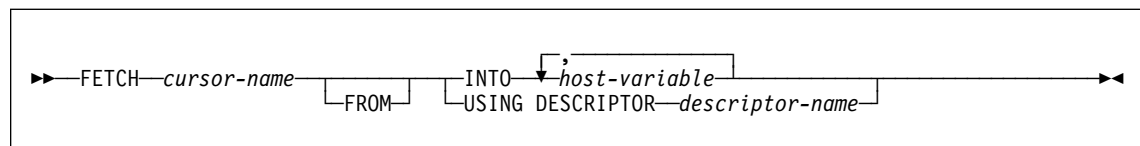
### Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

See “DECLARE CURSOR” on page 595 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 “DECLARE CURSOR” on page 595. The DECLARE CURSOR statement must precede the FETCH statement in the source program. 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 the result table:

- SQLCODE is set to +100, and SQLSTATE is set to '02000'.
- The cursor is positioned after the last row.
- Values are not assigned to host variables.

If the cursor is currently positioned before a row, it will be repositioned on that row, and values will be assigned to host variables as specified by INTO or USING.

If the cursor is currently positioned on a row other than the last row, it will be repositioned on the next row and values of that row will be assigned to host variables as specified by INTO or USING.

#### **INTO** *host-variable, ...*

Identifies one or more host variables that must be described in accordance with the rules for declaring host variables. The first value in the result row is assigned to the first host variable in the list, the second value to the second host variable, and so

## FETCH

on. For LOB values in the select-list, the target can be a regular host variable (if it is large enough), a locator variable, or a file-reference variable.

### **USING DESCRIPTOR** *descriptor-name*

Identifies an SQLDA that must contain a valid description of zero or more 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.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to  $16 + \text{SQLN} \times \text{N}$ , where N is the length of an SQLVAR occurrence.

If LOB result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table). See “Effect of DESCRIBE on the SQLDA” on page 767, which discusses SQLDOUBLED and LOB columns.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. For more information, see Appendix C, “Appendix C. SQL Descriptor Area (SQLDA)” on page 763.

The *n*th variable identified by the INTO clause or described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of each variable must be compatible with its corresponding column.

Each assignment to a variable is made according to the rules described in Chapter 3. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLDA is set to 'W'. Note that there is no warning if there are more variables than the number of result columns. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

## Notes

- 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 referenced 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.

- When retrieving into LOB locators in situations where it is not necessary to retain the locator across FETCH statements, it is good practice to issue a FREE LOCATOR statement before issuing the next FETCH statement, as locator resources are limited.
- It is possible for an error to occur that makes the state of the cursor unpredictable.
- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement. See the “Notes” on page 627 for information.
- DB2 CLI supports additional fetching capabilities. For instance when a cursor's result table is read-only, the SQLFetchScroll() function can be used to position the cursor at any spot within that result table.

### Examples

*Example 1:* In this C example, the FETCH statement fetches the results of the SELECT statement into the 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 TDEPT
  WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
}

EXEC SQL CLOSE C1;
```

*Example 2:* This FETCH statement uses an SQLDA.

```
FETCH CURS USING DESCRIPTOR :sqlda3
```

## FREE LOCATOR

---

### FREE LOCATOR

The FREE LOCATOR statement removes the association between a locator variable and its value.

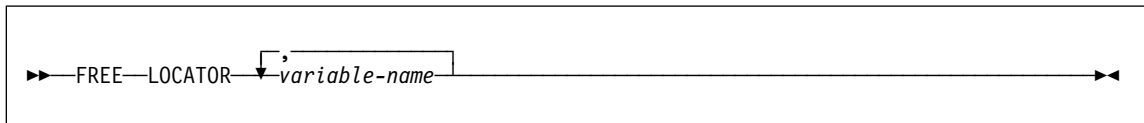
#### Invocation

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

**LOCATOR** *variable-name*, ...

Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables.

The locator-variable must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a FETCH statement or a SELECT INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is raised (SQLSTATE 0F001).

If more than one locator is specified, all locators that can be freed will be freed, regardless of errors detected in other locators in the list.

#### Example

In a COBOL program, free the BLOB locator variables TKN-VIDEO and TKN-BUF and the CLOB locator variable LIFE-STORY-LOCATOR.

```
EXEC SQL  
FREE LOCATOR :TKN-VIDEO, :TKN-BUF, :LIFE-STORY-LOCATOR  
END-EXEC.
```

## GRANT (Database Authorities)

### GRANT (Database Authorities)

This form of the GRANT statement grants authorities that apply to the entire database (rather than privileges that apply to specific objects within the database).

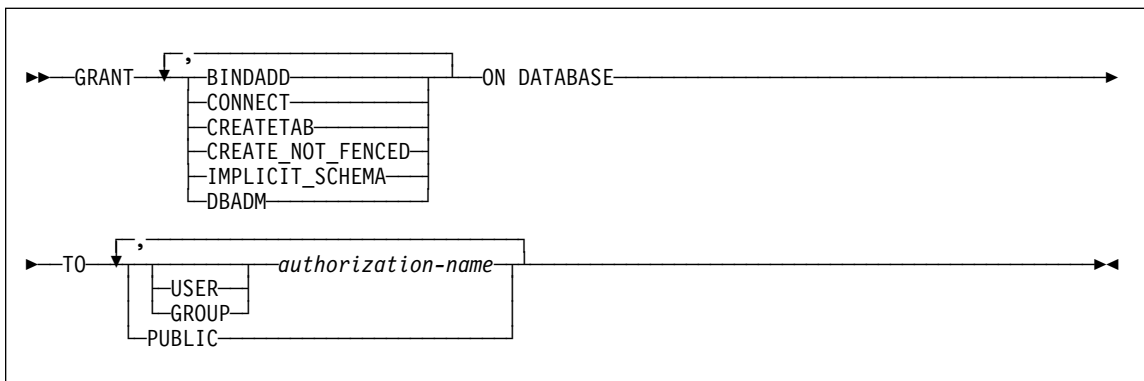
### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

To grant DBADM authority, SYSADM authority is required. To grant other authorities, either DBADM or SYSADM authority is required.

### Syntax



### Description

#### **BINDADD**

Grants the authority to create packages. The creator of a package automatically has the CONTROL privilege on that package and retains this privilege even if the BINDADD authority is subsequently revoked.

#### **CONNECT**

Grants the authority to access the database.

#### **CREATETAB**

Grants the authority to create base tables. The creator of a base table automatically has the CONTROL privilege on that table. The creator retains this privilege even if the CREATETAB authority is subsequently revoked.

There is no explicit authority required for view creation. A view can be created at any time if the authorization ID of the statement used to create the view has either CONTROL or SELECT privilege on each base table of the view.

## GRANT (Database Authorities)

### CREATE\_NOT\_FENCED

Grants the authority to register functions that execute in the database manager's process. Care must be taken that functions so registered will not have adverse side effects (see the FENCED or NOT FENCED clause on page 476 for more information.)

Once a function has been registered as not fenced, it continues to run in this manner even if CREATE\_NOT\_FENCED is subsequently revoked.

### IMPLICIT\_SCHEMA

Grants the authority to implicitly create a schema.

### DBADM

Grants the database administrator authority. A database administrator has all privileges against all objects in the database and may grant these privileges to others.

BINDADD, CONNECT, CREATETAB, CREATE\_NOT\_FENCED and IMPLICIT\_SCHEMA are automatically granted to an *authorization-name* that is granted DBADM authority.

**TO** Specifies to whom the authorities are granted.

### USER

Specifies that the *authorization-name* identifies a user.

### GROUP

Specifies that the *authorization-name* identifies a group name.

*authorization-name,...*

Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)

### PUBLIC

Grants the authorities to all users. DBADM cannot be granted to PUBLIC.

## Rules

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.



## GRANT (Database Authorities)

### Examples

*Example 1:* Give the users WINKEN, BLINKEN, and NOD the authority to connect to the database.

```
GRANT CONNECT ON DATABASE TO USER WINKEN, USER BLINKEN, USER NOD
```

*Example 2:* GRANT BINDADD authority on the database to a group named D024. There is both a group and a user called D024 in the system.

```
GRANT BINDADD ON DATABASE TO GROUP D024
```

Observe that, the GROUP keyword must be specified; otherwise, an error will occur since both a user and a group named D024 exist. Any member of the D024 group will be allowed to bind packages in the database, but the D024 user will not be allowed (unless this user is also a member of the group D024, had been granted BINDADD authority previously, or BINDADD authority had been granted to another group of which D024 was a member).

## GRANT (Index Privileges)

---

### GRANT (Index Privileges)

This form of the GRANT statement grants the CONTROL privilege on indexes.

#### Invocation

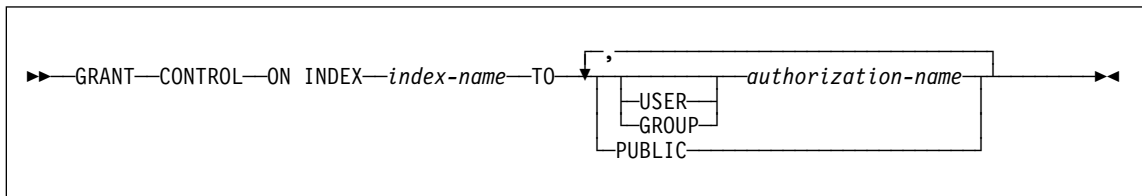
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- DBADM authority
- SYSADM authority.

#### Syntax



#### Description

##### CONTROL

Grants the privilege to drop the index. This is the CONTROL authority for indexes, which is automatically granted to creators of indexes.

##### ON INDEX *index-name*

Identifies the index for which the CONTROL privilege is to be granted.

**TO** Specifies to whom the privileges are granted.

##### USER

Specifies that the *authorization-name* identifies a user.

##### GROUP

Specifies that the *authorization-name* identifies a group name.

##### *authorization-name*,...

Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)

## GRANT (Index Privileges)

### **PUBLIC**

Grants the privileges to all users.

### **Rules**

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

### **Example**

```
GRANT CONTROL ON INDEX DEPTIDX TO USER USER4
```

## GRANT (Package Privileges)

---

### GRANT (Package Privileges)

This form of the GRANT statement grants privileges on a package.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

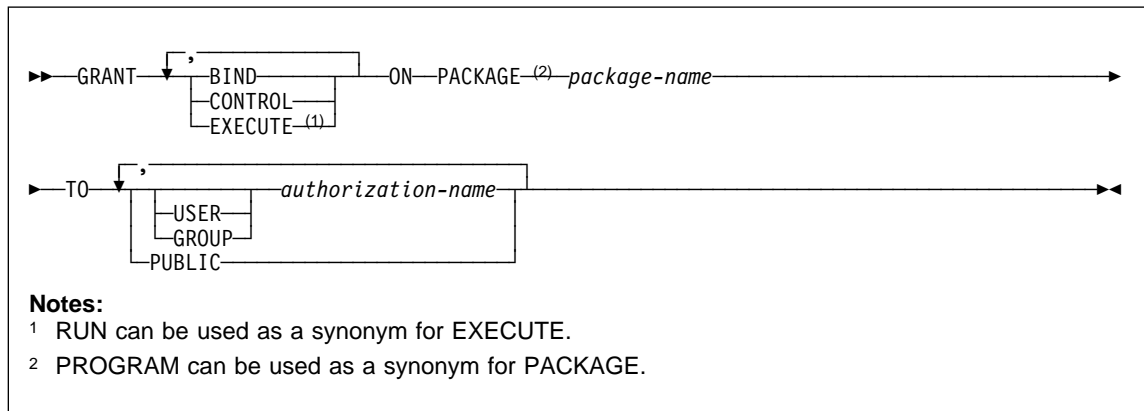
#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- CONTROL privilege on the referenced package
- SYSADM or DBADM authority.

To grant the CONTROL privilege, SYSADM or DBADM authority is required.

#### Syntax



#### Description

##### **BIND**

Grants the privilege to bind a package. The BIND privilege is really a rebind privilege, because the package must have already been bound (by someone with BINDADD authority) to have existed at all.

In addition to the BIND privilege, the user must hold the necessary privileges on each table referenced by static DML statements contained in the program. This is necessary because authorization on static DML statements is checked at bind time.

## GRANT (Package Privileges)

### CONTROL

Grants the privilege to rebind, drop, or execute the package, and extend package privileges to other users. The CONTROL privilege for packages is automatically granted to creators of packages

BIND and EXECUTE are automatically granted to an *authorization-name* that is granted CONTROL privilege.

### EXECUTE

Grants the privilege to execute the package.

### ON PACKAGE *package-name*

Specifies the name of the package on which privileges are to be granted.

**TO** Specifies to whom the privileges are granted.

### USER

Specifies that the *authorization-name* identifies a user.

### GROUP

Specifies that the *authorization-name* identifies a group name.

*authorization-name*,...

Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)

### PUBLIC

Grants the privileges to all users.

## Rules

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

## Examples

*Example 1:* Grant the EXECUTE privilege on PACKAGE CORPDATA.PKGA to PUBLIC.

```
GRANT EXECUTE
ON PACKAGE CORPDATA.PKGA
TO PUBLIC
```

*Example 2:* GRANT EXECUTE privilege on package CORPDATA.PKGA to a user named EMPLOYEE. There is neither a group nor a user called EMPLOYEE.

## GRANT (Package Privileges)

```
GRANT EXECUTE ON PACKAGE  
CORPDATA.PKGA TO EMPLOYEE
```

or

```
GRANT EXECUTE ON PACKAGE  
CORPDATA.PKGA TO USER EMPLOYEE
```

## GRANT (Schema Privileges)

### GRANT (Schema Privileges)

This form of the GRANT statement grants privileges on a schema.

#### Invocation

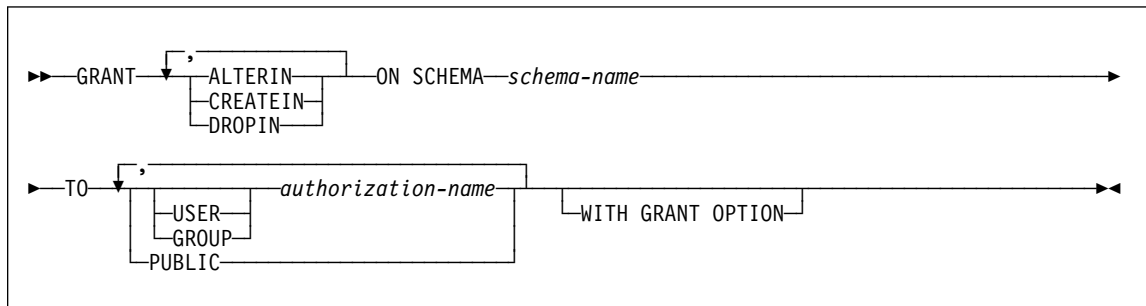
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- the WITH GRANT OPTION for each identified privilege
- SYSADM or DBADM authority

#### Syntax



#### Description

##### ALTERIN

Grants the privilege to alter or comment on all objects in the schema. The owner of an explicitly created schema automatically receives ALTERIN privilege.

##### CREATEIN

Grants the privilege to create objects in the schema. Other authorities or privileges required to create the object (such as CREATETAB) are still required. The owner of an explicitly created schema automatically receives CREATEIN privilege. An implicitly created schema has CREATEIN privilege automatically granted to PUBLIC.

##### DROPIN

Grants the privilege to drop all objects in the schema. The owner of an explicitly created schema automatically receives DROPIN privilege.

##### ON SCHEMA *schema-name*

Identifies the schema on which the privileges are to be granted.

## GRANT (Schema Privileges)

**TO** Specifies to whom the privileges are granted.

### **USER**

Specifies that the *authorization-name* identifies a user.

### **GROUP**

Specifies that the *authorization-name* identifies a group name.

*authorization-name,...*

Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)

### **PUBLIC**

Grants the privileges to all users.

### **WITH GRANT OPTION**

Allows the specified *authorization-names* to GRANT the privileges to others.

If the WITH GRANT OPTION is omitted, the specified *authorization-names* can only grant the privileges to others if they:

- have DBADM authority or
- received the ability to grant privileges from some other source.

## Rules

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.
- Privileges cannot be granted on schema names SYSIBM, SYSCAT, SYSFUN and SYSSTAT by any user.
- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges was not granted. If no privileges were granted, an error is returned (SQLSTATE 42501).<sup>80</sup>

---

<sup>80</sup> If the package used for processing the statement was precompiled with LANGLEVEL set to SQL92E for MIA, a warning is returned (SQLSTATE 01007) unless the grantor has NO privileges on the object of the grant.



## GRANT (Schema Privileges)

### Examples

*Example 1:* Grant USER2 to the ability to create objects in schema CORPDATA.

```
GRANT CREATEIN ON SCHEMA CORPDATA TO USER2
```

*Example 2:* Grant user BIGGUY the ability to create and drop objects in schema CORPDATA.

```
GRANT CREATEIN, DROPIN ON SCHEMA CORPDATA TO BIGGUY
```

## GRANT (Table or View Privileges)

---

### GRANT (Table or View Privileges)

This form of the GRANT statement grants privileges on a table or view.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

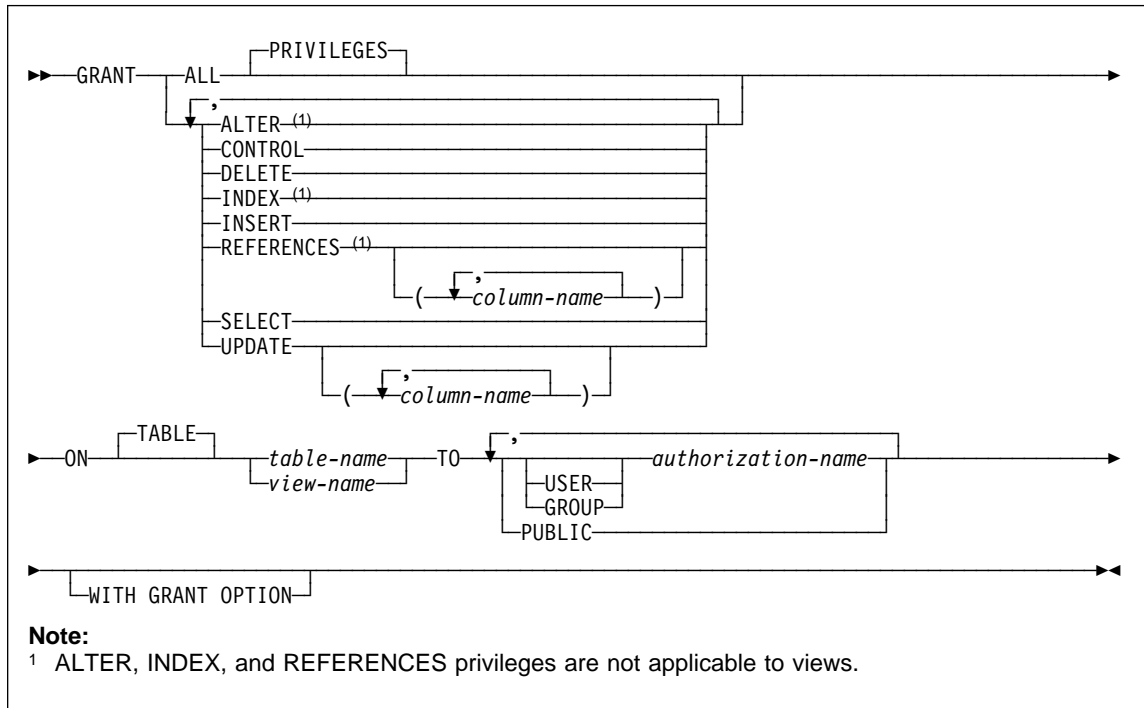
- CONTROL privilege on the referenced table or view
- The WITH GRANT OPTION for each identified privilege. If ALL is specified, the authorization ID must have some grantable privilege on the identified table or view.
- SYSADM or DBADM authority.

To grant the CONTROL privilege, SYSADM or DBADM authority is required.

To grant privileges on catalog tables and views, either SYSADM or DBADM authority is required.

#### Syntax

## GRANT (Table or View Privileges)



### Description

#### ALL or ALL PRIVILEGES

Grants all the appropriate privileges, except CONTROL, on the base table or view named in the ON clause.

If the authorization ID of the statement has CONTROL privilege on the table or view, or DBADM or SYSADM authority, then all the privileges applicable to the object (except CONTROL) are granted. Otherwise, the privileges granted are all those grantable privileges that the authorization ID of the statement has on the identified table or view.

If ALL is not specified, one or more of the keywords in the list of privileges must be specified.

#### ALTER

Grants the privilege to:

- Add columns to a base table definition.
- Create or drop a primary key or unique constraint on a base table. For more information on the authorization required to create or drop a primary key or a unique constraint, see “ALTER TABLE” on page 380.
- Create or drop a foreign key on a base table.

## GRANT (Table or View Privileges)

The REFERENCES privilege on each column of the parent table is also required.

- Create or drop a check constraint on a base table.
- Create a trigger on a base table.
- Add or change a comment on a base table or view.

### CONTROL

Grants:

- All of the appropriate privileges in the list, that is:
  - ALTER, CONTROL, DELETE, INSERT, INDEX, REFERENCES, SELECT, and UPDATE to base tables
  - CONTROL, DELETE, INSERT, SELECT, and UPDATE to views
- The ability to grant the above privileges (except for CONTROL) to others.
- The ability to drop the base table or view.

This ability cannot be extended to others on the basis of holding CONTROL privilege. The only way that it can be extended is by granting the CONTROL privilege itself and that can only be done by someone with SYSADM or DBADM authority.
- The ability to execute the RUNSTATS utility on the table and indexes. See the *Command Reference* for information on RUNSTATS.
- The ability to issue set constraints on the base table or summary table.

The definer of a base table or summary table automatically receives the CONTROL privilege.

The definer of a view automatically receives the CONTROL privilege if the definer holds the CONTROL privilege on all tables and views identified in the fullselect.

### DELETE

Grants the privilege to delete rows from the table or updatable view.

### INDEX

Grants the privilege to create an index on the table. The creator of an index automatically has the CONTROL privilege on the index (authorizing the creator to drop the index), and retains this privilege even if the INDEX privilege is revoked.

### INSERT

Grants the privilege to insert rows into the table or updatable view, and run the IMPORT utility.

### REFERENCES

Grants the privilege to create and drop a foreign key referencing the table as the parent.

If the authorization ID of the statement has one of:

- DBADM or SYSADM authority

## GRANT (Table or View Privileges)

- CONTROL privilege on the table
- REFERENCES WITH GRANT OPTION on the table

then the grantee(s) can create referential constraints using all columns of the table as parent key, even those added later using the ALTER TABLE statement. Otherwise, the privileges granted are all those grantable column REFERENCES privileges that the authorization ID of the statement has on the identified table. For more information on the authorization required to create or drop a foreign key, see “ALTER TABLE” on page 380.

### REFERENCES (*column-name*,...)

Grants the privilege to create and drop a foreign key using only those columns specified in the column list as a parent key. Each *column-name* must be an unqualified name that identifies a column of the table identified in the ON clause. Column level REFERENCES privilege cannot be granted on typed tables or typed views (SQLSTATE 42997).

### SELECT

Grants the privilege to retrieve rows from the table or view, create views on the table, and run the EXPORT utility. See the *Command Reference* for information on EXPORT.

### UPDATE

Grants the privilege to use the UPDATE statement on the table or updatable view identified in the ON clause.

If the authorization ID of the statement has one of:

- DBADM or SYSADM authority
- CONTROL privilege on the table or view
- UPDATE WITH GRANT OPTION on the table or view

then the grantee(s) can update all updatable columns of the table or view on which the grantor has with grant privilege as well as those columns added later using the ALTER TABLE statement. Otherwise, the privileges granted are all those grantable column UPDATE privileges that the authorization ID of the statement has on the identified table or view.

### UPDATE (*column-name*,...)

Grants the privilege to use the UPDATE statement to update only those columns specified in the column list. Each *column-name* must be an unqualified name that identifies a column of the table or view identified in the ON clause. Column level UPDATE privilege cannot be granted on typed tables or typed views (SQLSTATE 42997).

### ON TABLE *table-name* or *view-name*

Specifies the table or view on which privileges are to be granted.

No privileges may be granted on an inoperative view or an inoperative summary table (SQLSTATE 51024).

**TO** Specifies to whom the privileges are granted.

## GRANT (Table or View Privileges)

### USER

Specifies that the *authorization-name* identifies a user.

### GROUP

Specifies that the *authorization-name* identifies a group name.

*authorization-name,...*

Lists the authorization IDs of one or more users or groups.<sup>81</sup>

A privilege granted to a group is not used for authorization checking on static DML statements in a package. Nor is it used when checking authorization on a base table while processing a CREATE VIEW statement.

In DB2 Universal Database, table privileges granted to groups only apply to statements that are dynamically prepared. For example, if the INSERT privilege on the PROJECT table has been granted to group D204 but not UBIQUITY (a member of D204) UBIQUITY could issue the statement:

```
EXEC SQL EXECUTE IMMEDIATE :INSERT_STRING;
```

where the content of the string is:

```
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3114', 'TOOL PROGRAMMING', 'D21', '000260');
```

but could not precompile or bind a program with the statement:

```
EXEC SQL INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3114', 'TOOL PROGRAMMING', 'D21', '000260');
```

### PUBLIC

Grants the privileges to all users.<sup>82</sup>

### WITH GRANT OPTION

Allows the specified *authorization-names* to GRANT the privileges to others.

If the specified privileges include CONTROL, the WITH GRANT OPTION applies to all the applicable privileges except for CONTROL (SQLSTATE 01516).

## Rules

- If neither USER nor GROUP is specified, then
  - If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  - If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  - If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

---

<sup>81</sup> Restrictions in previous versions on grants to authorization ID of the user issuing the statement have been removed.

<sup>82</sup> Restrictions in previous versions on the use of privileges granted to PUBLIC for static SQL statements and CREATE VIEW statements have been removed.

## GRANT (Table or View Privileges)

- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges was not granted. If no privileges were granted, an error is returned (SQLSTATE 42501).<sup>83</sup> If CONTROL privilege is specified, privileges will only be granted if the authorization ID of the statement has SYSADM or DBADM authority (SQLSTATE 42501).

### Notes

- Privileges may be granted independently at every level of a table hierarchy. A user with a privilege on a supertable may affect the subtables. For example, an update specifying the supertable *T* may show up as a change to a row in the subtable *S* of *T* done by a user with UPDATE privilege on *T* but without UPDATE privilege on *S*. A user can only operate directly on the subtable if the necessary privilege is held on the subtable.

### Examples

*Example 1:* Grant all privileges on the table WESTERN\_CR to PUBLIC.

```
GRANT ALL ON WESTERN_CR  
TO PUBLIC
```

*Example 2:* Grant the appropriate privileges on the CALENDAR table so that users PHIL and CLAIRE can read it and insert new entries into it. Do not allow them to change or remove any existing entries.

```
GRANT SELECT, INSERT ON CALENDAR  
TO USER PHIL, USER CLAIRE
```

*Example 3:* Grant all privileges on the COUNCIL table to user FRANK and the ability to extend all privileges to others.

```
GRANT ALL ON COUNCIL  
TO USER FRANK WITH GRANT OPTION
```

*Example 4:* GRANT SELECT privilege on table CORPDATA.EMPLOYEE to a user named JOHN. There is a user called JOHN and no group called JOHN.

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO JOHN
```

or

```
GRANT SELECT  
ON CORPDATA.EMPLOYEE TO USER JOHN
```

*Example 5:* GRANT SELECT privilege on table CORPDATA.EMPLOYEE to a group named JOHN. There is a group called JOHN and no user called JOHN.

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO JOHN
```

---

<sup>83</sup> If the package used for processing the statement was precompiled with LANGLEVEL set to SQL92E or MIA, a warning is returned (SQLSTATE 01007) unless the grantor has NO privileges on the object of the grant.

## GRANT (Table or View Privileges)

or

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO GROUP JOHN
```

*Example 6:* GRANT INSERT and SELECT on table T1 to both a group named D024 and a user named D024.

```
GRANT INSERT, SELECT ON TABLE T1  
TO GROUP D024, USER D024
```

In this case, both the members of the D024 group and the user D024 would be allowed to INSERT into and SELECT from the table T1. Also, there would be two rows added to the SYSCAT.TABAUTH catalog view.

*Example 7:* GRANT INSERT, SELECT, and CONTROL on the CALENDAR table to user FRANK. FRANK must be able to pass the privileges on to others.

```
GRANT CONTROL ON TABLE CALENDAR  
TO FRANK WITH GRANT OPTION
```

The result of this statement is a warning (SQLSTATE 01516) that CONTROL was not given the WITH GRANT OPTION. Frank now has the ability to grant any privilege on CALENDAR including INSERT and SELECT as required. FRANK cannot grant CONTROL on CALENDAR to other users unless he has SYSADM or DBADM authority.



## INCLUDE

The INCLUDE statement inserts declarations into a source program.

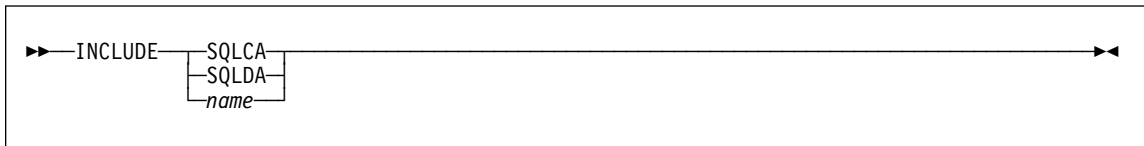
### Invocation

This statement can only be embedded in an application program. It is not an executable statement.

### Authorization

None required.

### Syntax



### Description

#### SQLCA

Indicates the description of an SQL communication area (SQLCA) is to be included. For a description of the SQLCA, see Appendix B, “SQL Communication Area (SQLCA)” on page 759.

#### SQLDA

Indicates the description of an SQL descriptor area (SQLDA) is to be included. For a description of the SQLDA, see Appendix C, “Appendix C. SQL Descriptor Area (SQLDA)” on page 763.

#### *name*

Identifies an external file containing text that is to be included in the source program being precompiled. It may be an SQL identifier without a filename extension or a literal in single quotes ( ' '). An SQL identifier assumes the filename extension of the source file being precompiled. If a filename extension is not provided by a literal in quotes then none is assumed.

For host language specific information, see the *Embedded SQL Programming Guide*.

### Notes

- When a program is precompiled, the INCLUDE statement is replaced by source statements. Thus, the INCLUDE statement should be specified at a point in the program such that the resulting source statements are acceptable to the compiler.
- The external source file must be written in the host language specified by the *name*. If it is greater than 18 characters or contains characters not allowed in an SQL identifier then it must be in single quotes. INCLUDE *name* statements may be

## INCLUDE

nested though not cyclical (for example, if A and B are modules and A contains an `INCLUDE name` statement, then it is not valid for A to call B and then B to call A).

- When the `LANGLEVEL` precompile option is specified with the `SQL92E` value, `INCLUDE SQLCA` should not be specified. `SQLSTATE` and `SQLCODE` variables may be defined within the host variable declare section.

## Example

Include an SQLCA in a C program.

```
EXEC SQL INCLUDE SQLCA;  
  
EXEC SQL DECLARE C1 CURSOR FOR  
      SELECT DEPTNO, DEPTNAME, MGRNO FROM TDEPT  
      WHERE ADMRDEPT = 'A00';  
  
EXEC SQL OPEN C1;  
  
while (SQLCODE==0) {  
    EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;  
  
    (Print results)  
  
}  
  
EXEC SQL CLOSE C1;
```

## INSERT

The INSERT statement inserts rows into a table or view. Inserting a row into a view also inserts the row into the table on which the view is based.

### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

To execute this statement, the privileges held by the authorization ID of the statement must include at least one of the following:

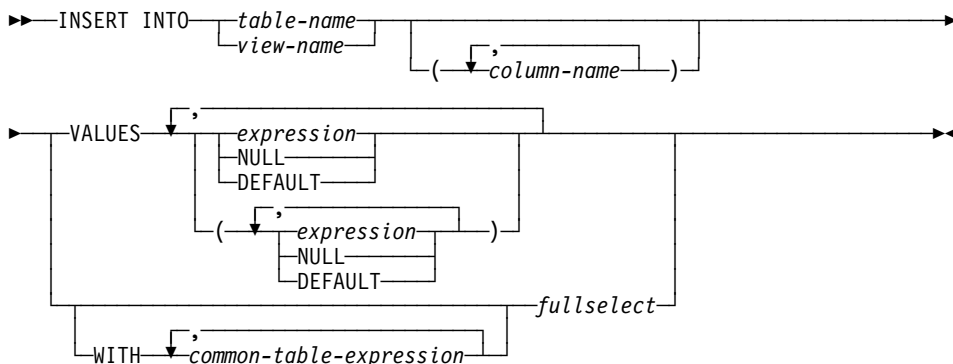
- INSERT privilege on the table or view where rows are to be inserted
- CONTROL privilege on the table or view where rows are to be inserted
- SYSADM or DBADM authority.

In addition, for each table or view referenced in any fullselect used in the INSERT statement, the privileges held by the authorization ID of the statement must include at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

GROUP privileges are not checked for static INSERT statements.

### Syntax



**Note:** See Chapter 5, “Queries” on page 315 for the syntax of *common-table-expression* and *fullselect*.

# INSERT

## Description

### **INTO** *table-name* or *view-name*

Identifies the object of the insert operation. The name must identify a table or view that exists at the application server, but it must not identify a catalog table, a summary table, a view of a catalog table, or a read-only view.

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 operation is a view with such columns, a list of column names must be specified, and the list must not identify these columns.

### *(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 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 prepared and therefore does not include columns that were added to a table after the statement was prepared.

The implicit column list is established at prepare time. Hence an INSERT statement embedded in an application program does not use any columns that might have been added to the table or view after prepare time.

## **VALUES**

Introduces one or more rows of values to be inserted.

Each host variable named must be described in the program in accordance with the rules for declaring host variables.

The number of values for each row 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.

### **expression**

An *expression* can be as defined in “Expressions” on page 117.

### **NULL**

Specifies the null value and should only be specified for nullable columns.

### **DEFAULT**

Specifies that the default value is to be used. The value that is inserted depends on how the column was defined, as follows:

- If the WITH DEFAULT clause is used, the default inserted is as defined for the column (see default-clause in “ALTER TABLE” on page 380).
- If the WITH DEFAULT clause or the NOT NULL clause is not used, the value inserted is NULL.

## INSERT

- If the NOT NULL clause is used and the WITH DEFAULT clause is not used or DEFAULT NULL is used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

### WITH *common-table-expression*

Defines a common table expression for use with the fullselect that follows. See “common-table-expression” on page 356 for an explanation of the *common-table-expression*.

### *fullselect*

Specifies a set of new rows in the form of the result table of a fullselect. There may be one, more than one, or none. If the result table is empty, SQLCODE is set to +100 and SQLSTATE is set to '02000'.

When the base object of the INSERT and the base object of the fullselect or any subquery of the fullselect, are the same table, the fullselect is completely evaluated before any rows are inserted.

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.

## Rules

- **Default values:** The value inserted in any column that is not in the column list is either the default value of the column or null. Columns that do not allow null values and are not defined with NOT NULL WITH DEFAULT must be included in the column list. Similarly, if you insert into a view, the value inserted into any column of the base table that is not in the view is either the default value of the column or null. Hence, all columns of the base table that are not in the view must have either a default value or allow null values.
- **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 either be 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.
- **Validity:** If the table named, or the base table of the view named, has one or more unique indexes, each row inserted into the table must conform to the constraints imposed by those indexes. If a view whose definition includes WITH CHECK OPTION is named, each row inserted into the view must conform to the definition of the view. For an explanation of the rules governing this situation, see “CREATE VIEW” on page 582.
- **Referential Integrity:** For each constraint defined on a table, each non-null insert value of the foreign key must be equal to a primary key value of the parent table.

## INSERT

- **Check Constraint:** Insert values must satisfy the check conditions of the check constraints defined on the table. An INSERT to a table with check constraints defined has the constraint conditions evaluated once for each row that is inserted.
- **Triggers:** Insert statements may cause triggers to be executed. A trigger may cause other statements to be executed or may raise error conditions based on the insert values.
- **Datalinks:** Insert statements that include DATALINK values will result in an attempt to link the file if a URL value is included (not empty string or blanks) and the column is defined with FILE LINK CONTROL. Errors in the DATALINK value or in linking the file will cause the insert to fail (SQLSTATE 428D1 or 57050).

## Notes

- After execution of an INSERT statement that is embedded within a program, the value of the third variable of the SQLERRD(3) portion of the SQLCA indicates the number of rows that were inserted. SQLERRD(5) contains the count of all triggered insert, update and delete operations.
- Unless appropriate locks already exist, one or more exclusive locks are acquired at the execution of a successful INSERT statement. Until the locks are released, an inserted row can only be accessed by:
  - The application process that performed the insert.
  - Another application process using isolation level UR through a read-only cursor, SELECT INTO statement, or subselect used in a subquery.
- For further information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements.
- If an application is running against a partitioned database, and it is bound with option INSERT BUF, then INSERT with VALUES statements which are not processed using EXECUTE IMMEDIATE may be buffered. DB2 assumes that such an INSERT statement is being processed inside a loop in the application's logic. Rather than execute the statement to completion, it attempts to buffer the new row values in one or more buffers. As a result the actual insertions of the rows into the table are performed later, asynchronous with the application's INSERT logic. Be aware that this asynchronous insertion may cause an error related to an INSERT to be returned on some other SQL statement that follows the INSERT in the application.

This has the potential to dramatically improve INSERT performance, but is best used with clean data, due to the asynchronous nature of the error handling. See buffered insert in the *Embedded SQL Programming Guide* for further details.

## Examples

*Example 1:* Insert a new department with the following specifications into the DEPARTMENT table:

- Department number (DEPTNO) is 'E31'
- Department name (DEPTNAME) is 'ARCHITECTURE'
- Managed by (MGRNO) a person with number '00390'

## INSERT

- Reports to (ADMRDEPT) department 'E01'.

```
INSERT INTO DEPARTMENT  
VALUES ('E31', 'ARCHITECTURE', '00390', 'E01')
```

*Example 2:* Insert a new department into the DEPARTMENT table as in example 1, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT )  
VALUES ('E31', 'ARCHITECTURE', 'E01')
```

*Example 3:* Insert two new departments using one statement into the DEPARTMENT table as in example 2, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)  
VALUES ('B11', 'PURCHASING', 'B01'), ('E41', 'DATABASE ADMINISTRATION', 'E01')
```

*Example 4:* Create a temporary table MA\_EMP\_ACT with the same columns as the EMP\_ACT table. Load MA\_EMP\_ACT with the rows from the EMP\_ACT table with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE TABLE MA_EMP_ACT  
  ( EMPNO CHAR(6) NOT NULL,  
    PROJNO CHAR(6) NOT NULL,  
    ACTNO SMALLINT NOT NULL,  
    EMPTIME DEC(5,2),  
    EMSTDATE DATE,  
    EMENDATE DATE )  
  
INSERT INTO MA_EMP_ACT  
SELECT * FROM EMP_ACT  
WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

*Example 5:* Use a C program statement to add a skeleton project to the PROJECT table. Obtain the project number (PROJNO), project name (PROJNAME), department number (DEPTNO), and responsible employee (RESPEMP) from host variables. Use the current date as the project start date (PRSTDATE). Assign a NULL value to the remaining columns in the table.

```
EXEC SQL INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE )  
VALUES (:PRJNO, :PRJNM, :DPTNO, :REMP, CURRENT DATE);
```

## LOCK TABLE

---

### LOCK TABLE

The LOCK TABLE statement either prevents concurrent application processes from changing a table or prevents concurrent application processes from using a table.

#### Invocation

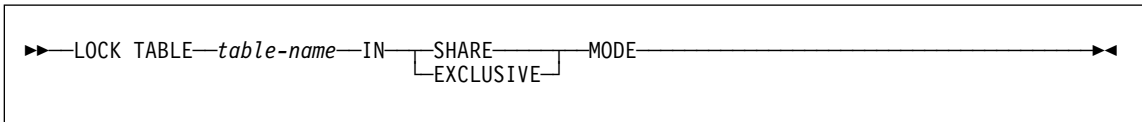
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SELECT privilege on the table
- CONTROL privilege on the table
- SYSADM or DBADM authority.

#### Syntax



#### Description

##### *table-name*

Identifies the table. The *table-name* must identify a table that exists at the application server, but it must not identify a catalog table. If the *table-name* is a typed table, it must be the root table of the table hierarchy (SQLSTATE 428DR).

##### **IN SHARE MODE**

Prevents concurrent application processes from executing any but read-only operations on the table.

##### **IN EXCLUSIVE MODE**

Prevents concurrent application processes from executing any operations on the table. Note that EXCLUSIVE MODE does not prevent concurrent application processes that are running at isolation level Uncommitted Read (UR) from executing read-only operations on the table.

#### Notes

- Locking is used to prevent concurrent operations. A lock is not necessarily acquired during the execution of the LOCK TABLE statement if a suitable lock already exists. The lock that prevents concurrent operations is held at least until the termination of the unit of work.



## LOCK TABLE

- In a partitioned database, a table lock is first acquired at the first partition in the nodegroup (the partition with the lowest number) and then at other partitions . If the LOCK TABLE statement is interrupted, the table may be locked on some partitions but not on others. If this occurs, either issue another LOCK TABLE statement to complete the locking on all partitions , or issue a COMMIT or ROLLBACK statement to release the current locks.
- This statement affects all partitions in the nodegroup.

### Example

Obtain a lock on the table EMP. Do not allow other programs either to read or update the table.

```
LOCK TABLE EMP IN EXCLUSIVE MODE
```

## OPEN

---

### OPEN

The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

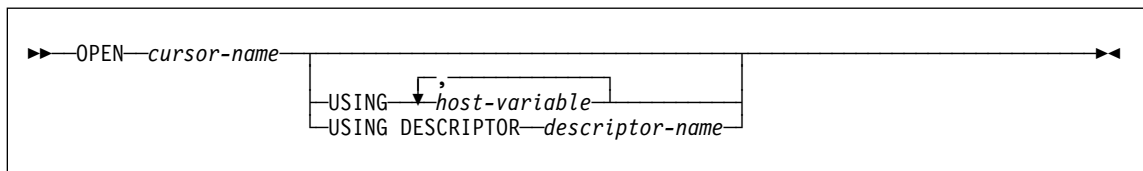
### Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

See “DECLARE CURSOR” on page 595 for the authorization required to use a cursor.

### Syntax



### Description

#### *cursor-name*

Names a cursor that is defined in a DECLARE CURSOR statement that was stated earlier in the program. When the OPEN statement is executed, the cursor must be in the closed state.

The DECLARE CURSOR statement must identify a SELECT statement, in one of the following ways:

- Including the SELECT statement in the DECLARE CURSOR statement
- Including a *statement-name* that names a prepared SELECT statement.

The result table of the cursor is derived by evaluating that SELECT statement, using the current values of any host variables specified in it or in the USING clause of the OPEN statement. The rows of the result table may be derived during the execution of the OPEN statement and a temporary table may be created to hold them; or they may be derived during the execution of subsequent 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 state of the cursor is effectively “after the last row”.

#### **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 673.) If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, USING must be

used. If the prepared statement does not include parameter markers, USING is ignored.

*host-variable*

Identifies a variable described in the program in accordance with the rules for declaring host variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Where appropriate, locator variables and file reference variables can be provided as the source of values for parameter markers.

**DESCRIPTOR** *descriptor-name*

Identifies an SQLDA that must contain a valid description of 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
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to  $16 + \text{SQLN} * (N)$ , where N is the length of an SQLVAR occurrence.

If LOB result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table). See “Effect of DESCRIBE on the SQLDA” on page 767, which discusses SQLDOUBLED and LOB columns.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. For more information, see Appendix C, “Appendix C. SQL Descriptor Area (SQLDA)” on page 763.

## Rules

- When the SELECT statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by its corresponding host variable. 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. See “Rules” on page 674 for the rules affecting parameter markers.
- Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:
  - V must be compatible with the target.

## OPEN

- 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 SELECT statement of the cursor is evaluated, 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 with two blanks.

- 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. 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 variables 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.

## Notes

- **Closed state of cursors:** All cursors in a program are in the closed state when the program is initiated and when it initiates a ROLLBACK statement.  
All cursors, except open cursors declared WITH HOLD, are in a closed state when a program issues a COMMIT statement.  
A cursor can also be in the closed state because a CLOSE statement was executed or an error was detected that made the position of the cursor unpredictable.
- To retrieve rows from the result table of a cursor, 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 temporary tables:** In some cases, the result table of a cursor is derived during the execution of FETCH statements. In other cases, the temporary table method is used instead. With this method the entire result table is transferred to a temporary table during the execution of the OPEN statement. When a temporary table is used, the results of a program can differ in these two ways:
  - An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
  - INSERT, UPDATE, and DELETE statements executed in the same transaction while the cursor is open cannot affect the result table.

Conversely, if a temporary table is not used, INSERT, UPDATE, and DELETE statements executed while the cursor is open can affect the result table if issued from the same unit of work. The *Embedded SQL Programming Guide* describes how locking can be used to control the effect of INSERT, UPDATE, and DELETE operations executed by concurrent units of work. Your result table can also be

affected by operations executed by your own unit of work, and 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 a new row is inserted into T, the effect of that insert on the result table is not predictable because its rows are not ordered. Thus a subsequent `FETCH C` may or may not retrieve the new row of T.

- Statement caching affects cursors declared open by the OPEN statement. See the “Notes” on page 627 for information.

## Examples

*Example 1:* Write the embedded statements in a COBOL program that will:

1. Define a cursor C1 that is to be used to retrieve all rows from the DEPARTMENT table for departments that are administered by (ADMRDEPT) department 'A00'.
2. Place the cursor C1 before the first row to be fetched.

```
EXEC SQL  DECLARE C1 CURSOR FOR
          SELECT DEPTNO, DEPTNAME, MGRNO
          FROM DEPARTMENT
          WHERE ADMRDEPT = 'A00'
END-EXEC.
```

```
EXEC SQL  OPEN C1
END-EXEC.
```

*Example 2:* Code an OPEN statement to associate a cursor DYN\_CURSOR with a dynamically defined select-statement in a C program. Assuming two parameter markers are used in the predicate of the select-statement, two host variable references are supplied with the OPEN statement to pass integer and varchar(64) values between the application and the database. (The related host variable definitions, PREPARE statement, and DECLARE CURSOR statement are also shown in the example below.)

```
EXEC SQL  BEGIN DECLARE SECTION;
          static short   hv_int;
          char           hv_vchar64[64];
          char           stmt1_str[200];
EXEC SQL  END DECLARE SECTION;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING :hv_int, :hv_vchar64;
```

*Example 3:* Code an OPEN statement as in example 2, but in this case the number and data types of the parameter markers in the WHERE clause are not known.

## OPEN

```
EXEC SQL BEGIN DECLARE SECTION;  
      char stmt1_str[200];  
EXEC SQL END DECLARE SECTION;  
EXEC SQL INCLUDE SQLDA;  
  
EXEC SQL PREPARE STMT1_NAME FROM :stmt1_str;  
EXEC SQL DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;  
  
EXEC SQL OPEN DYN_CURSOR USING DESCRIPTOR :sqlda;
```

## PREPARE

The PREPARE statement is used by application programs to dynamically prepare an SQL statement for execution. The PREPARE statement creates an executable SQL statement, called a *prepared statement*, from a character string form of the statement, called a *statement string*.

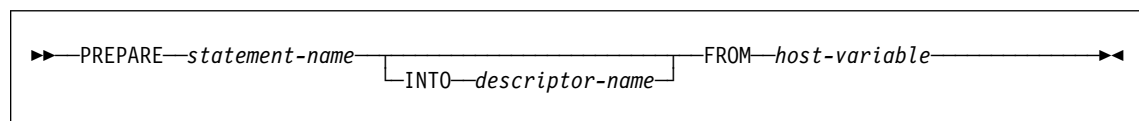
### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

For statements where authorization checking is performed at statement preparation time (DML), the privileges held by the authorization ID of the statement must include those required to execute the SQL statement specified by the PREPARE statement. For statements where authorization checking is performed at statement execution (DDL, GRANT, and REVOKE statements), no authorization is required to use the statement; however, the authorization is checked when the prepared statement is executed.

### Syntax



### Description

#### *statement-name*

Names the prepared statement. If the name identifies an existing prepared statement, that previously prepared statement is destroyed. The name must not identify a prepared statement that is the SELECT statement of an open cursor.

#### INTO

If INTO is used, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by the descriptor-name.

#### *descriptor-name*

Is the name of an SQLDA.<sup>84</sup>

#### FROM

Introduces the statement string. The statement string is the value of the specified host variable.

<sup>84</sup> The DESCRIBE statement may be used as an alternative to this clause. See "DESCRIBE" on page 604.

## PREPARE

### *host-variable*

Must identify a host variable that is described in the program in accordance with the rules for declaring character string variables. It must be a character-string variable (either fixed-length or varying-length).

## Rules

- **Rules for statement strings:** The statement string must be one of the following SQL statements:

- ALTER
- COMMENT ON
- COMMIT
- CREATE
- DELETE
- DROP
- GRANT
- INSERT
- LOCK TABLE
- REFRESH TABLE
- RENAME TABLE
- REVOKE
- ROLLBACK
- select-statement
- SET CONSTRAINTS
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET EVENT MONITOR STATE
- SET PATH
- SET SCHEMA
- SIGNAL SQLSTATE
- UPDATE

The statement string must not:

- Be a SELECT statement with an INTO clause
  - Be a VALUES statement with an INTO clause
  - 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 may include *parameter markers*; those can be replaced by the values of host variables when the prepared statement is executed. A parameter marker is a question mark (?) that is declared where a host variable could be stated if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see “OPEN” on page 668 and “EXECUTE” on page 626.



There are two types of parameter markers:

#### **Typed parameter marker**

A parameter marker that is specified along with its target data type. It has the general form:

```
CAST(? AS data-type)
```

This notation is not a function call, but a “promise” that the type of the parameter at run time will be of the data type specified or some data type that can be converted to the specified data type. For example, in:

```
UPDATE EMPLOYEE
SET LASTNAME = TRANSLATE(CAST(? AS VARCHAR(12)))
WHERE EMPNO = ?
```

the value of the argument of the TRANSLATE function will be provided at run time. The data type of that value will either be VARCHAR(12), or some type that can be converted to VARCHAR(12).

#### **Untyped parameter marker**

A parameter marker that is specified without its target data type. It has the form of a single question mark. The data type of an untyped parameter marker is provided by context. For example, the untyped parameter marker in the predicate of the above update statement 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. These locations and the resulting data type are found in Table 22. The locations are grouped in this table into expressions, predicates and functions to assist in determining applicability of an untyped parameter marker. When an untyped parameter marker is used in a function (including arithmetic operators, CONCAT and datetime operators) with an unqualified function name, the qualifier is set to 'SYSIBM' for the purposes of function resolution.

Table 22 (Page 1 of 5). Untyped Parameter Marker Usage

Untyped Parameter Marker Location	Data Type
<b>Expressions (including select list, CASE and VALUES)</b>	
Alone in a select list	Error
Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules.	Error
Includes cases such as:	
? + ? + 10	

## PREPARE

Table 22 (Page 2 of 5). Untyped Parameter Marker Usage

Untyped Parameter Marker Location	Data Type
One operand of a single operator in an arithmetic expression (not a datetime expression)  Includes cases such as:  ? + ? * 10	The data type of the other operand.
Labelled duration within a datetime expression. (Note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker.)	DECIMAL(15,0)
Any other operand of a datetime expression (for instance 'timecol + ?' or '? - datecol').	Error
Both operands of a CONCAT operator	Error
One operand of a CONCAT operator where the other operand is a non-CLOB character data type	If one operand is either CHAR(n) or VARCHAR(n), where n is less than 128, then other is VARCHAR(254 - n). In all other cases the data type is VARCHAR(254).
One operand of a CONCAT operator where the other operand is a non-DBCLOB graphic data type.	If one operand is either GRAPHIC(n) or VARGRAPHIC(n), where n is less than 64, then other is VARCHAR(127 - n). In all other cases the data type is VARCHAR(127).
One operand of a CONCAT operator where the other operand is a large object string.	Same as that of the other operand.
As a value on the right hand side of a SET clause of an UPDATE statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type.
The expression following CASE in a simple CASE expression	Error
At least one of the result-expressions in a CASE expression (both Simple and Searched) with the rest of the result-expressions either untyped parameter marker or NULL.	Error
Any or all expressions following WHEN in a simple CASE expression.	Result of applying the "Rules for Result Data Types" on page 82 to the expression following CASE and the expressions following WHEN that are not untyped parameter markers.
.A result-expression in a CASE expression (both Simple and Searched) where at least one result-expression is not NULL and not an untyped parameter marker.	Result of applying the Rules for Result Data Types to all result-expressions that are other than NULL or untyped parameter markers.
Alone as a column-expression in a single-row VALUES clause that is not within an INSERT statement.	Error

Table 22 (Page 3 of 5). Untyped Parameter Marker Usage

Untyped Parameter Marker Location	Data Type
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the column-expressions in the same position in all other row-expressions are untyped parameter markers.	Error
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the expression in the same position of at least one other row-expression is not an untyped parameter marker or NULL.	Result of applying the “Rules for Result Data Types” on page 82 on all operands that are other than untyped parameter markers.
Alone as a column-expression in a single-row VALUES clause within an INSERT statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type.
Alone as a column-expression in a multi-row VALUES clause within an INSERT statement.	The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type.
<b>Predicates</b>	
Both operands of a comparison operator	Error
One operand of a comparison operator where the other operand other than an untyped parameter marker.	The data type of the other operand
All operands of a BETWEEN predicate	Error
Either 1st and 2nd, or 1st and 3rd operands of a BETWEEN predicate	Same as that of the only non-parameter marker.
Remaining BETWEEN situations (i.e. one untyped parameter marker only)	Result of applying the “Rules for Result Data Types” on page 82 on all operands that are other than untyped parameter markers.
All operands of an IN predicate	Error
Both the 1st and 2nd operands of an IN predicate.	Result of applying the Rules for Result Data Types on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers.
The 1st operand of an IN predicate where the right hand side is a fullselect.	Data type of the selected column
Any or all operands of the IN list of the IN predicate	Data type of the 1st operand (left hand side)

## PREPARE

Table 22 (Page 4 of 5). Untyped Parameter Marker Usage

Untyped Parameter Marker Location	Data Type
The 1st operand and zero or more operands in the IN list excluding the 1st operand of the IN list	Result of applying the Rules for Result Data Types on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers.
All three operands of the LIKE predicate.	Match expression (operand 1) and pattern expression (operand 2) are VARCHAR(4000). Escape expression (operand 3) is VARCHAR(2).
The match expression of the LIKE predicate when either the pattern expression or the escape expression is other than an untyped parameter marker.	Either VARCHAR(4000) or VARGRAPHIC(2000) depending on the data type of the first operand that is not an untyped parameter marker.
The pattern expression of the LIKE predicate when either the match expression or the escape expression is other than an untyped parameter marker.	Either VARCHAR(4000) or VARGRAPHIC(2000) depending on the data type of the first operand that is not an untyped parameter marker. If the data type of the match expression is BLOB, the data type of the pattern expression is assumed to be BLOB(4000).
The escape expression of the LIKE predicate when either the match expression or the pattern expression is other than an untyped parameter marker.	Either VARCHAR(2) or VARGRAPHIC(1) depending on the data type of the first operand that is not an untyped parameter marker. If the data type of the match expression or pattern expression is BLOB, the data type of the escape expression is assumed to be BLOB(1).
Operand of the NULL predicate	error
Functions	
All operands of COALESCE (also called VALUE) or NULLIF	Error
.Any operand of COALESCE where at least one operand is other than an untyped parameter marker.	Result of applying the "Rules for Result Data Types" on page 82 on all operands that are other than untyped parameter markers.
An operand of NULLIF where the other operand is other than an untyped parameter marker.	The data type of the other operand
POSSTR (both operands)	Both operands are VARCHAR(4000).
POSSTR (one operand where the other operand is a character data type).	VARCHAR(4000).
POSSTR (one operand where the other operand is a graphic data type).	VARGRAPHIC(2000).
POSSTR (the search-string operand when the other operand is a BLOB).	BLOB(4000).
SUBSTR (1st operand)	VARCHAR(4000)

Table 22 (Page 5 of 5). Untyped Parameter Marker Usage

Untyped Parameter Marker Location	Data Type
SUBSTR (2nd and 3rd operands)	INTEGER
The 1st operand of the TRANSLATE scalar function.	Error
The 2nd and 3rd operands of the TRANSLATE scalar function.	VARCHAR(4000) if the first operand is a character type. VARGRAPHIC(2000) if the first operand is a graphic type.
The 4th operand of the TRANSLATE scalar function.	VARCHAR(1) if the first operand is a character type. VARGRAPHIC(1) if the first operand is a graphic type.
The 2nd operand of the TIMESTAMP scalar function.	TIME
Unary minus	DOUBLE PRECISION
Unary plus	DOUBLE PRECISION
All other operands of all other scalar functions including user-defined functions.	Error
Operand of a column function	Error

## Notes

- When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, the error condition is reported in the SQLCA. Any subsequent EXECUTE or OPEN statement that references this statement will also receive the same error (due to an implicit prepare done by the system) unless the error has been corrected.
- Prepared statements can be referred to in the following kinds of statements, with the restrictions shown:

In...	The prepared statement ...
DECLARE CURSOR	must be SELECT
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.
- Statement caching affects repeated preparations. See the “Notes” on page 627 for information.
- See the *Embedded SQL Programming Guide* for examples of dynamic SQL statements in the supported host languages.

## PREPARE

### Examples

*Example 1:* Prepare and execute a non-select-statement in a COBOL program. Assume the statement is contained in a host variable HOLDER and that the program will place a statement string into the host variable based on some instructions from the user. The statement to be prepared does not have any parameter markers.

```
EXEC SQL PREPARE STMT_NAME FROM :HOLDER  
END-EXEC.
```

```
EXEC SQL EXECUTE STMT_NAME  
END-EXEC.
```

*Example 2:* Prepare and execute a non-select-statement as in example 1, except code it for a C program. Also assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL PREPARE STMT_NAME FROM :holder;  
EXEC SQL EXECUTE STMT_NAME USING DESCRIPTOR :insert_da;
```

Assume that the following statement is to be prepared:

```
INSERT INTO DEPT VALUES(?, ?, ?, ?)
```

The columns in the DEPT table are defined as follows:

```
DEPT_NO  CHAR(3) NOT NULL, -- department number  
DEPTNAME VARCHAR(29), -- department name  
MGRNO    CHAR(6), -- manager number  
ADMRDEPT CHAR(3) -- admin department number
```

To insert department number G01 named COMPLAINTS, which has no manager and reports to department A00, the structure INSERT\_DA should have the following values before issuing the EXECUTE statement.

# PREPARE

SQLDAID	192	
SQLDABC	4	
SQLN	4	
SQLD		
SQLTYPE	452	
SQLLEN	3	
SQLDATA		→ G01
SQLIND		
SQLNAME		
SQLTYPE	449	
SQLLEN	29	
SQLDATA		→ COMPLAINTS
SQLIND		→ 0
SQLNAME		
SQLTYPE	453	
SQLLEN	6	
SQLDATA		
SQLIND		→ -1
SQLNAME		
SQLTYPE	453	
SQLLEN	3	
SQLDATA		→ A00
SQLIND		→ 0
SQLNAME		

## REFRESH TABLE

---

### REFRESH TABLE

The REFRESH TABLE statement refreshes the data in a summary table.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CONTROL privilege on the table.

#### Syntax

►► REFRESH TABLE *table-name* ◀◀

#### Description

*table-name*

Names a table.

The name, including the implicit or explicit schema, must identify a table that already exists at the current server. The table must allow the REFRESH TABLE statement (SQLSTATE 42809). This includes summary tables defined with:

- REFRESH IMMEDIATE
- REFRESH DEFERRED



## RELEASE

The RELEASE statement places one or more connections in the release-pending state.

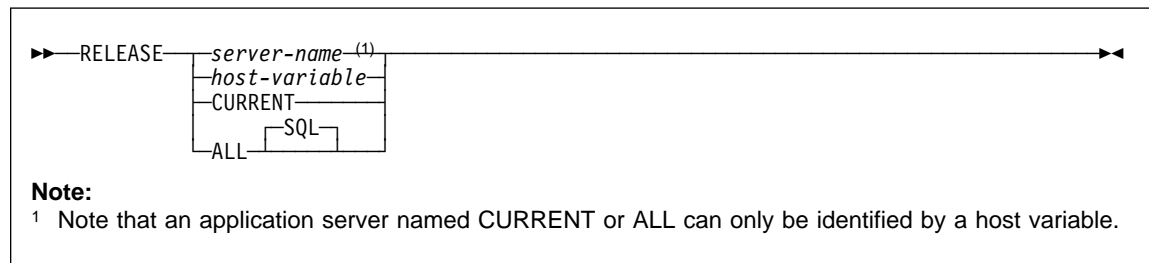
### Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

None Required.

### Syntax



### Description

#### *server-name* or *host-variable*

Identifies the application server by the specified *server-name* or a *host-variable* which contains the *server-name*.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The specified database-alias or the database-alias contained in the host variable must identify an existing connection of the application process. If the database-alias does not identify an existing connection, an error (SQLSTATE 08003) is raised.

#### **CURRENT**

Identifies the current connection of the application process. The application process must be in the connected state. If not, an error (SQLSTATE 08003) is raised.

#### **ALL**

Identifies all existing connections of the application process. This form of the RELEASE statement places all existing connections of the application process in

## RELEASE

the release-pending state. All connections will therefore be destroyed during the next commit operation. An error or warning does not occur if no connections exist when the statement is executed. The optional keyword SQL is included to be compatible with DB2/MVS SQL syntax.

### Notes

- If the RELEASE statement is successful, the identified connection is placed in the release-pending state and will therefore be destroyed 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.
- If the current connection is in the release-pending state when a commit operation is performed, the destruction of that connection places the application process into the unconnected state. In this case, the next executed SQL statement should be CONNECT or SET CONNECTION.
- Type 1 CONNECT semantics do not preclude the use of RELEASE. The connection will be disconnected when the unit of work is committed.
- RELEASE does not close cursors, does not release any resources, and does not prevent further use of the connection.
- A rollback operation 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 destroyed as soon as possible.
- Connections can also be destroyed during a commit operation because the option DISCONNECT(AUTOMATIC) or the option DISCONNECT(CONDITIONAL) is in effect. See "Options that Govern Distributed Unit of Work Semantics" on page 33 for information about the specification of the DISCONNECT option.

### Examples

*Example 1:* The SQL connection to IBMSTHDB is no longer needed by the application. The following statement will cause it to be destroyed during the next commit operation:

```
EXEC SQL RELEASE IBMSTHDB;
```

*Example 2:* The current connection is no longer needed by the application. The following statement will cause it to be destroyed during the next commit operation:

```
EXEC SQL RELEASE CURRENT;
```

*Example 3:* If an application has no need to access the databases after a commit but will continue to run for a while, then it is better not to tie up those connections unnecessarily. The following statement can be executed before the commit to ensure all connections will be destroyed at the commit:

```
EXEC SQL RELEASE ALL;
```

## RENAME TABLE

The RENAME TABLE statement renames an existing table.

### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

The privileges held by the authorization ID of the statement must include either SYSADM or DBADM authority or CONTROL privilege.

### Syntax

```

▶▶—RENAME—TABLE—source-table-name—TO—target-identifier—▶▶

```

### Description

#### *source-table-name*

Names the existing table that is to be renamed. The name, including the schema name, must identify a table that already exists in the database (SQLSTATE 42704). It can be an alias identifying the table. It must not be the name of a catalog table (SQLSTATE 42832), a summary table, a typed table (SQLSTATE 42997) or an object of other than table or alias (SQLSTATE 42809).

#### *target-identifier*

Specifies the new name for the table without a schema name. The schema name of the *source-table-name* is used to qualify the new name for the table. The qualified name must *not* identify a table, view, or alias that already exists in the database (SQLSTATE 42710).

### Rules

The source table must not:

- be referenced in any existing view definitions or summary table definitions
- be referenced in any triggered SQL statements in existing triggers or be the subject table of an existing trigger
- have any check constraints
- be a parent or dependent table in any referential integrity constraints
- be the scope of any existing reference column.

An error (SQLSTATE 42986) is returned if the source table violates one or more of these conditions.

## RENAME TABLE

### Notes

- Catalog entries are updated to reflect the new table name.
- *All* authorizations associated with the source table name are *transferred* to the new table name (the authorization catalog tables are updated appropriately).
- Indexes defined over the source table are *transferred* to the new table (the index catalog tables are updated appropriately).
- Any packages that are dependent on the source table are invalidated.
- If an alias is used for the *source-table-name*, it must resolve to a table name. The table is renamed within the schema of this table. The alias is not changed by the RENAME statement and continues to refer to the old table name.
- A table with primary key or unique constraints may be renamed if none of the primary key or unique constraints are referenced by any foreign key.

### Example

Change the name of the EMP table to EMPLOYEE.

```
RENAME TABLE EMP TO EMPLOYEE
```

## REVOKE (Database Authorities)

### REVOKE (Database Authorities)

This form of the REVOKE statement revokes authorities that apply to the entire database.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

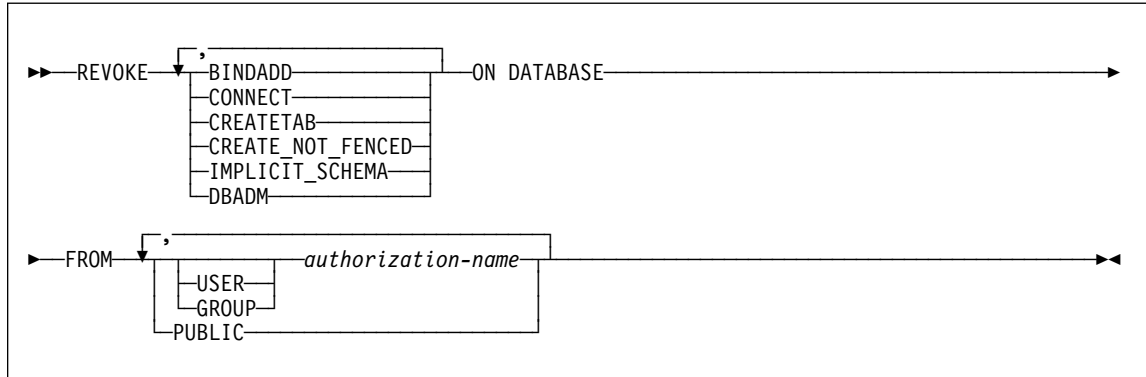
#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- DBADM authority
- SYSADM authority.

To revoke DBADM authority, SYSADM authority is required.

#### Syntax



#### Description

##### BINDADD

Revokes the authority to create packages. The creator of a package automatically has the CONTROL privilege on that package and retains this privilege even if his BINDADD authority is subsequently revoked.

The BINDADD authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority.

##### CONNECT

Revokes the authority to access the database.

Revoking the CONNECT authority from a user does not affect any privileges that were granted to that user on objects in the database. If the user is subsequently

## REVOKE (Database Authorities)

granted the CONNECT authority again, all previously held privileges are still valid (assuming they were not explicitly revoked).

The CONNECT authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

### CREATETAB

Revokes the authority to create tables. The creator of a table automatically has the CONTROL privilege on that table, and retains this privilege even if his CREATETAB authority is subsequently revoked.

The CREATETAB authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

### CREATE\_NOT\_FENCED

Revokes the authority to register functions that execute in the database manager's process. However, once a function has been registered as not fenced, it continues to run in this manner even if CREATE\_NOT\_FENCED is subsequently revoked from the authorization ID that registered the function.

The CREATE\_NOT\_FENCED authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

### IMPLICIT\_SCHEMA

Revokes the authority to implicitly create a schema. It does not affect the ability to create objects in existing schemas or to process a CREATE SCHEMA statement.

### DBADM

Revokes the DBADM authority.

DBADM authority cannot be revoked from PUBLIC (because it cannot be granted to PUBLIC).

**Revoking DBADM authority does not automatically revoke any privileges that were held by the authorization-name on objects in the database, nor does it revoke BINDADD, CONNECT, CREATETAB, IMPLICIT\_SCHEMA, or CREATE\_NOT\_FENCED authority.**

### FROM

Indicates from whom the authorities are revoked.

### USER

Specifies that the *authorization-name* identifies a user.

### GROUP

Specifies that the *authorization-name* identifies a group name

*authorization-name*,...

Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the authorities from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

## REVOKE (Database Authorities)

### **PUBLIC**

Revokes the authorities from PUBLIC.

### Rules

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.DBAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, then an error is raised (SQLSTATE 56092).

### Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have a higher level authority such as DBADM.

### Examples

*Example 1:* Given that USER6 is only a user and not a group, revoke the privilege to create tables from the user USER6.

```
REVOKE CREATETAB ON DATABASE FROM USER6
```

*Example 2:* Revoke BINDADD authority on the database from a group named D024. There are two rows in the SYSCAT.DBAUTH catalog view for this grantee; one with a GRANTEETYPE of U and one with a GRANTEETYPE of G.

```
REVOKE BINDADD ON DATABASE FROM GROUP D024
```

In this case, the GROUP keyword must be specified; otherwise an error will occur (SQLSTATE 56092).

## REVOKE (Index Privileges)

---

### REVOKE (Index Privileges)

This form of the REVOKE statement revokes the CONTROL privilege on an index.

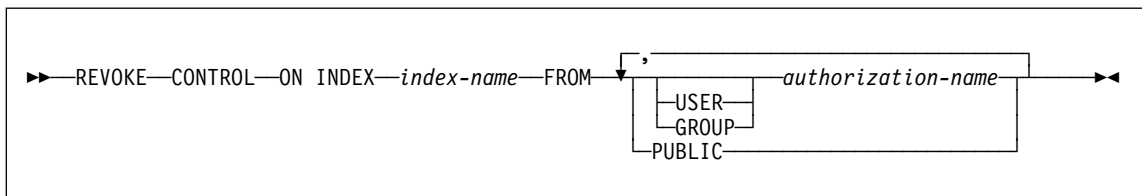
#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The authorization ID of the statement must hold either SYSADM or DBADM authority (SQLSTATE 42501).

#### Syntax



#### Description

##### CONTROL

Revokes the privilege to drop the index. This is the CONTROL privilege for indexes, which is automatically granted to creators of indexes.

##### ON INDEX *index-name*

Specifies the name of the index on which the CONTROL privilege is to be revoked.

##### FROM

Indicates from whom the privileges are revoked.

##### USER

Specifies that the *authorization-name* identifies a user.

##### GROUP

Specifies that the *authorization-name* identifies a group name

##### *authorization-name*,...

Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

##### PUBLIC

Revokes the privileges from PUBLIC.



## REVOKE (Index Privileges)

### Rules

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.INDEXAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, then an error is raised (SQLSTATE 56092).

### Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have authorities such as ALTERIN on the schema of an index.

### Examples

*Example 1:* Given that USER4 is only a user and not a group, revoke the privilege to drop an index DEPTIDX from the user USER4.

```
REVOKE CONTROL ON INDEX DEPTIDX FROM USER4
```

*Example 2:* Revoke the privilege to drop an index LUNCHITEMS from the user CHEF and the group WAITERS.

```
REVOKE CONTROL ON INDEX LUNCHITEMS  
FROM USER CHEF, GROUP WAITERS
```

## REVOKE (Package Privileges)

---

### REVOKE (Package Privileges)

This form of the REVOKE statement revokes CONTROL, BIND, and EXECUTE privileges against a package.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

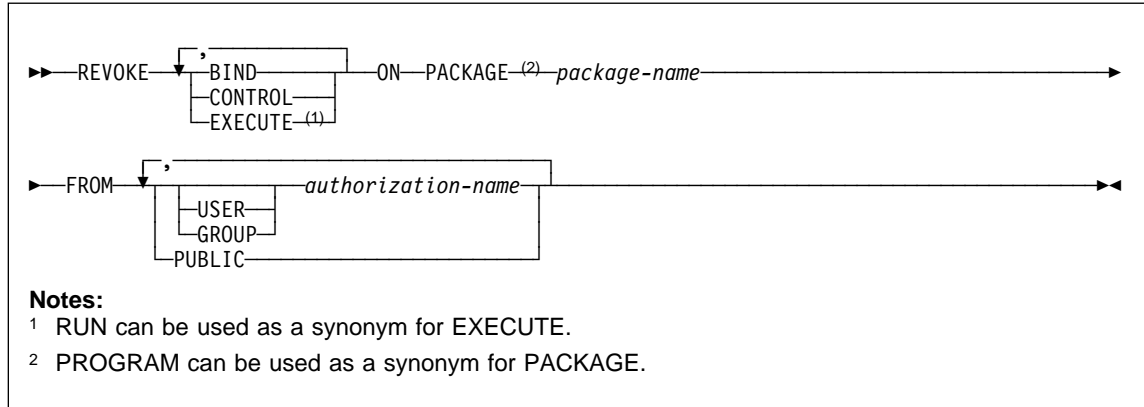
#### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- CONTROL privilege on the referenced package
- SYSADM or DBADM authority.

To revoke the CONTROL privilege, SYSADM or DBADM authority are required.

#### Syntax



#### Description

##### **BIND**

Revokes the privilege to execute BIND or REBIND on the referenced package.

The BIND privileges cannot be revoked from an *authorization-name* that holds CONTROL privilege on the package without also revoking the CONTROL privilege.

##### **CONTROL**

Revokes the privilege to drop the package and to extend package privileges to other users.

Revoking CONTROL does not revoke the other package privileges.

## REVOKE (Package Privileges)

### EXECUTE

Revokes the privilege to execute the package.

The EXECUTE privilege cannot be revoked from an *authorization-name* that holds CONTROL privilege on the package without also revoking the CONTROL privilege.

### ON PACKAGE *package-name*

Specifies the package on which privileges are revoked.

### FROM

Indicates from whom the privileges are revoked.

### USER

Specifies that the *authorization-name* identifies a user.

### GROUP

Specifies that the *authorization-name* identifies a group name.

*authorization-name,...*

Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

### PUBLIC

Revokes the privileges from PUBLIC.

## Rules

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.PACKAGEAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, then an error is raised (SQLSTATE 56092).

## Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have privileges such as ALTERIN on the schema of a package.

## Examples

*Example 1:* Revoke the EXECUTE privilege on package CORPDATA.PKGA from PUBLIC.

```
REVOKE EXECUTE
ON PACKAGE CORPDATA.PKGA
FROM PUBLIC
```

## REVOKE (Package Privileges)

*Example 2:* Revoke CONTROL authority on the RRSP\_PKG package for the user FRANK and for PUBLIC.

```
REVOKE CONTROL
ON PACKAGE RRSP_PKG
FROM USER FRANK, PUBLIC
```

## REVOKE (Schema Privileges)

---

### REVOKE (Schema Privileges)

This form of the REVOKE statement revokes the privileges on a schema.

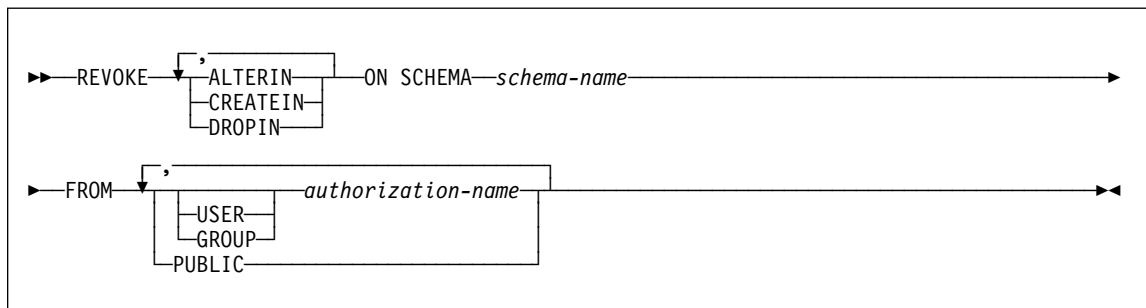
#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

The authorization ID of the statement must hold either SYSADM or DBADM authority (SQLSTATE 42501).

#### Syntax



#### Description

##### ALTERIN

Revokes the privilege to alter or comment on objects in the schema.

##### CREATEIN

Revokes the privilege to create objects in the schema.

##### DROPIN

Revokes the privilege to drop objects in the schema.

##### ON SCHEMA *schema-name*

Specifies the name of the schema on which privileges are to be revoked.

##### FROM

Indicates from whom the privileges are revoked.

##### USER

Specifies that the *authorization-name* identifies a user.

##### GROUP

Specifies that the *authorization-name* identifies a group name

## REVOKE (Schema Privileges)

*authorization-name*,...

Lists one or more authorization IDs.

The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

### **PUBLIC**

Revokes the privileges from PUBLIC.

## Rules

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.SCHEMAAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, then an error is raised (SQLSTATE 56092).

## Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have a higher level authority such as DBADM.

## Examples

*Example 1:* Given that USER4 is only a user and not a group, revoke the privilege to create objects in schema DEPTIDX from the user USER4.

```
REVOKE CREATEIN ON SCHEMA DEPTIDX FROM USER4
```

*Example 2:* Revoke the privilege to drop objects in schema LUNCH from the user CHEF and the group WAITERS.

```
REVOKE DROPIN ON SCHEMA LUNCH  
FROM USER CHEF, GROUP WAITERS
```

## REVOKE (Table or View Privileges)

### REVOKE (Table or View Privileges)

This form of the REVOKE statement revokes privileges on a table or view.

#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

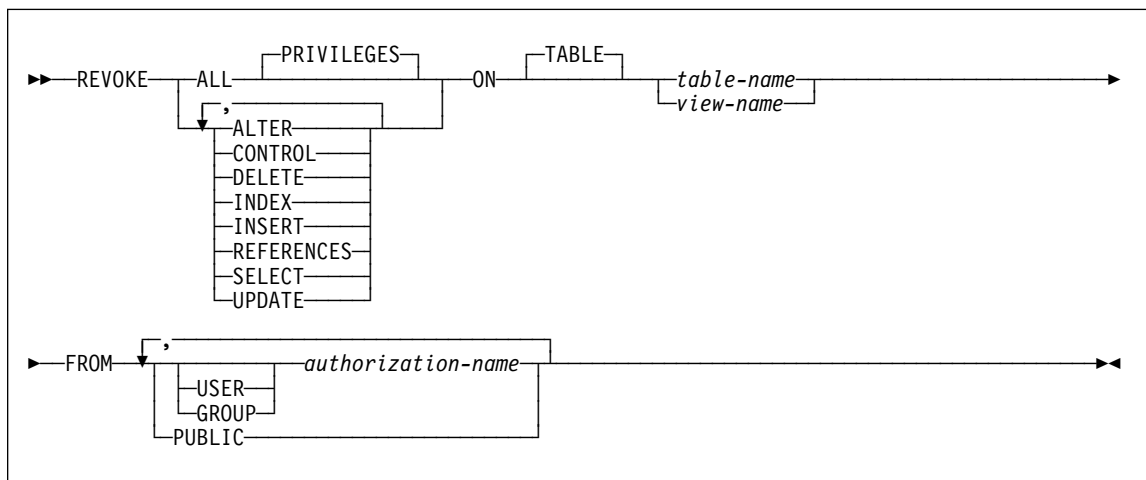
The privileges held by the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority
- CONTROL privilege on the referenced table or view.

To revoke the CONTROL privilege, either SYSADM or DBADM authority is required.

To revoke the privileges on catalog tables and views, either SYSADM or DBADM authority is required.

#### Syntax



#### Description

##### **ALL or ALL PRIVILEGES**

Revokes all privileges held by an authorization-name for the specified tables or views.

If ALL is not used, one or more of the keywords listed below must be used. Each keyword revokes the privilege described, but only as it applies to the tables or

## REVOKE (Table or View Privileges)

views named in the ON clause. The same keyword must not be specified more than once.

### ALTER

Revokes the privilege to add columns to the base table definition, create or drop a primary key or unique constraint on the table, create or drop a foreign key on the table, add/change a comment on the table, create or drop a check constraint, or create a trigger .

### CONTROL

Revokes the ability to drop the table or view, and the ability to execute the RUNSTATS utility on the table and indexes.

Revoking CONTROL privilege from an *authorization-name* does not revoke other privileges granted to the user on that object.

### DELETE

Revokes the privilege to delete rows from the table or updatable view.

### INDEX

Revokes the privilege to create an index on the table. The creator of an index automatically has the CONTROL privilege over the index (authorizing the creator to drop the index), and retains this privilege even if the INDEX privilege is revoked.

### INSERT

Revokes the privilege to insert rows into the table or updatable view and the privilege to run the IMPORT utility.

### REFERENCES

Revokes the privilege to create or drop a foreign key referencing the table as the parent. Any column level REFERENCES privileges are also revoked.

### SELECT

Revokes the privilege to retrieve rows from the table or view, create a view on a table, and run the EXPORT utility. Revoking SELECT privilege may cause some views to be marked inoperative. For information on inoperative views, see “Notes” on page 589.

### UPDATE

Revokes the privilege to update rows in the table or updatable view. Any column level UPDATE privileges are also revoked.

### ON TABLE *table-name* or *view-name*

Specifies the table or view on which privileges are to be revoked.

### FROM

Indicates from whom the privileges are revoked.

### USER

Specifies that the *authorization-name* identifies a user.

### GROUP

Specifies that the *authorization-name* identifies a group name.



## REVOKE (Table or View Privileges)

*authorization-name*,...

Lists one or more authorization IDs.

The ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

### **PUBLIC**

Revokes the privileges from PUBLIC.

## Rules

- If neither USER nor GROUP is specified, then:
  - If all rows for the grantee in the SYSCAT.TABAUTH and SYSCAT.COLAUTH catalog views have a GRANTEETYPE of U, then USER will be assumed.
  - If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  - If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  - If DCE authentication is used, then an error is raised (SQLSTATE 56092).

## Notes

- If a privilege is revoked from the *authorization-name* used to create a view (this is called the view's DEFINER in SYSCAT.VIEWS), that privilege is also revoked from any dependent views.
- If the DEFINER of the view loses a SELECT privilege on some object on which the view definition depends (or an object upon which the view definition depends is dropped (or made inoperative in the case of another view)), then the view will be made inoperative (see the "Notes" section in "CREATE VIEW" on page 582 for information on inoperative views).

However, if a DBADM or SYSADM explicitly revokes all privileges on the view from the DEFINER, then the record of the DEFINER will not appear in SYSCAT.TABAUTH but nothing will happen to the view - it remains operative.

- Privileges on inoperative views cannot be revoked.
- All packages dependent upon an object for which a privilege is revoked are marked invalid. A package remains invalid until a bind or rebind operation on the application is successfully executed, or the application is executed and the database manager successfully rebinds the application (using information stored in the catalogs). Packages marked invalid due to a revoke may be successfully rebound without any additional grants.

For example, if a package owned by USER1 contains a SELECT from table T1 and the SELECT privilege for table T1 is revoked from USER1, then the package will be marked invalid. If SELECT authority is re-granted, or if the user holds DBADM authority, the package is successfully rebound when executed.

- Packages, triggers or views that include the use of OUTER(Z) in the FROM clause, are dependent on having SELECT privilege on every subtable or subview of Z. Similarly, packages, triggers, or views that include the use of Deref(Y) where Y is

## REVOKE (Table or View Privileges)

a reference type with a target table or view *Z*, are dependent on having SELECT privilege on every subtable or subview of *Z*. If one of these SELECT privileges is revoked, such packages are invalidated and such triggers or views are made inoperative.

- Table or view privileges cannot be revoked from an *authorization-name* with CONTROL on the object without also revoking the CONTROL privilege (SQLSTATE 42504).
- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have privileges such as ALTERIN on the schema of a table or a view.
- If the DEFINER of the summary table loses a SELECT privilege on a table on which the summary table definition depends, (or a table upon which the summary table definition depends is dropped), then the summary table will be made inoperative (see the “Notes” on page 550 for information on inoperative summary tables).

However, if a DBADM or SYSADM explicitly revokes all privileges on the summary table from the DEFINER, then the record in SYSTABAUTH for the DEFINER will be deleted, but nothing will happen to the summary table - it remains operative.

**Note:** “Rules” on page 618 lists the dependencies that objects such as tables and views can have on one another.

## Examples

*Example 1:* Revoke SELECT privilege on table EMPLOYEE from user ENGLES. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is U.

```
REVOKE SELECT
ON TABLE EMPLOYEE
FROM ENGLES
```

*Example 2:* Revoke update privileges on table EMPLOYEE previously granted to all local users. Note that grants to specific users are not affected.

```
REVOKE UPDATE
ON EMPLOYEE
FROM PUBLIC
```

*Example 3:* Revoke all privileges on table EMPLOYEE from users PELLOW and MLI and from group PLANNERS.

```
REVOKE ALL
ON EMPLOYEE
FROM USER PELLOW, USER MLI, GROUP PLANNERS
```

*Example 4:* Revoke SELECT privilege on table CORPDATA.EMPLOYEE from a user named JOHN. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is U.

## REVOKE (Table or View Privileges)

```
REVOKE SELECT
ON CORPDATA.EMPLOYEE FROM JOHN
```

or

```
REVOKE SELECT
ON CORPDATA.EMPLOYEE FROM USER JOHN
```

Note that an attempt to revoke the privilege from GROUP JOHN would result in an error, since the privilege was not previously granted to GROUP JOHN.

*Example 5:* Revoke SELECT privilege on table CORPDATA.EMPLOYEE from a group named JOHN. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is G.

```
REVOKE SELECT
ON CORPDATA.EMPLOYEE FROM JOHN
```

or

```
REVOKE SELECT
ON CORPDATA.EMPLOYEE FROM GROUP JOHN
```

## ROLLBACK

---

### ROLLBACK

The ROLLBACK statement is used to terminate a unit of work and back out the database changes that were made by that unit of work.

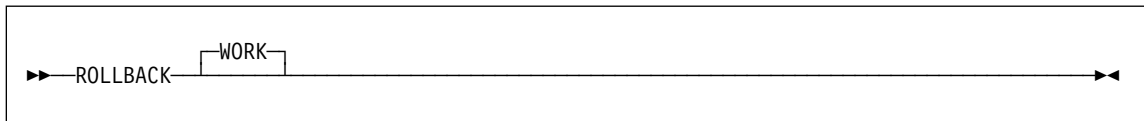
#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

None required.

#### Syntax



#### Description

The unit of work in which the ROLLBACK statement is executed is terminated and a new unit of work is initiated. All changes made to the database during the unit of work are backed out.

The following statements, however, are not under transaction control and changes made by them are independent of issuing the ROLLBACK statement:

- SET CONNECTION,
- SET CURRENT DEGREE,
- SET CURRENT EXPLAIN MODE,
- SET CURRENT EXPLAIN SNAPSHOT,
- SET CURRENT PACKAGESET,
- SET CURRENT QUERY OPTIMIZATION,
- SET CURRENT REFRESH AGE,
- SET EVENT MONITOR STATE,
- SET PATH,
- SET SCHEMA.

#### Notes

- All locks held by the unit of work are released. All open cursors are closed. All LOB locators are freed.
- Executing a ROLLBACK statement does not affect either the SET statements that change special register values or the RELEASE statement.

## ROLLBACK

- The termination of a unit of work is an implicit rollback if the program terminates abnormally due to a program check.
- Statement caching is affected by the rollback operation. See the “Notes” on page 627 for information.

### Example

Delete the alterations made since the last commit point or rollback.

**ROLLBACK WORK**

## SELECT

---

### SELECT

The SELECT statement is a form of query. It can be embedded in an application program or issued interactively. For detailed information, see “select-statement” on page 355.

## SELECT INTO

The SELECT INTO statement produces a result table consisting of 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 and '02000' to SQLSTATE and does not assign values to the host variables. If more than one row satisfies the search condition, statement processing is terminated, and an error occurs (SQLSTATE 21000).

### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared. The statement is not supported in REXX.

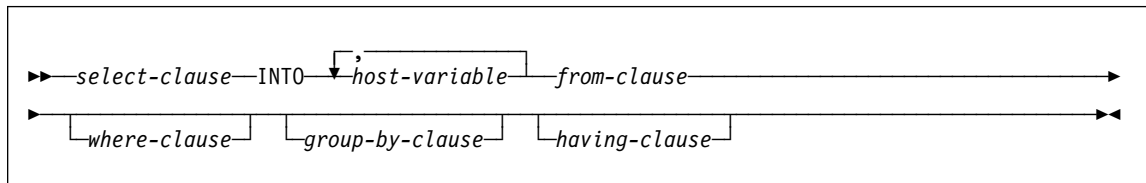
### Authorization

The privileges held by the authorization ID of the statement must include, for each table or view referenced in the SELECT INTO statement, at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

GROUP privileges are not checked for static SELECT INTO statements.

### Syntax



### Description

See Chapter 5, “Queries” on page 315 for a description of the *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, and *having-clause*.

#### INTO

Introduces a list of host variables.

#### *host-variable*

Identifies a variable that is described in the program under the rules for declaring 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 Appendix B, “SQL Communication Area (SQLCA)” on page 759.)

## SELECT INTO

Each assignment to a variable is made according to the rules described in “Assignments and Comparisons” on page 70. Assignments are made in sequence through the list.

If an error occurs, no value is assigned to any host variable.

### Examples

*Example 1:* This C example puts the maximum salary in EMP into the host variable MAXSALARY.

```
EXEC SQL SELECT MAX(SALARY)
INTO :MAXSALRY
FROM EMP;
```

*Example 2:* This C example puts the row for employee 528671, from EMP, into host variables.

```
EXEC SQL SELECT * INTO :h1, :h2, :h3, :h4
FROM EMP
WHERE EMPNO = '528671';
```



---

## SET CONNECTION

The SET CONNECTION statement changes the state of a connection from dormant to current, making the specified location the current server. It is not under transaction control.

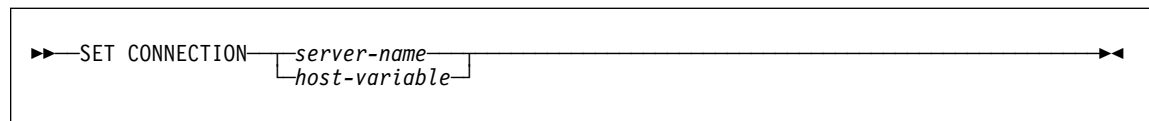
### Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

None Required.

### Syntax



### Description

*server-name* or *host-variable*

Identifies the application server by the specified *server-name* or a *host-variable* which contains the server-name.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-justified and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The *server-name* or the *host-variable* must identify an existing connection of the application process. If they do not identify an existing connection, an error (SQLSTATE 08003) is raised.

If SET CONNECTION is to the current connection, the states of all connections of the application process are unchanged.

#### **Successful Connection**

If the SET CONNECTION statement executes successfully:

- No connection is made. The CURRENT SERVER special register is updated with the specified *server-name*.
- The previously current connection, if any, is placed into the dormant state (assuming a different *server-name* is specified).

## SET CONNECTION

- The CURRENT SERVER special register and the SQLCA are updated in the same way as documented under Type 1 CONNECT; details on page 434.

### *Unsuccessful Connection*

If the SET CONNECTION statement fails:

- No matter what the reason for failure, the connection state of the application process and the states of its connections are unchanged.
- As with an unsuccessful Type 1 CONNECT, the SQLERRP field of the SQLCA is set to the name of the module that detected the error.

## Notes

- The use of type 1 CONNECT statements does not preclude the use of SET CONNECTION, but the statement will always fail (SQLSTATE 08003), unless the SET CONNECTION statement specifies the current connection, because dormant connections cannot exist.
- The SQLRULES(DB2) connection option (see “Options that Govern Distributed Unit of Work Semantics” on page 33) does not preclude the use of SET CONNECTION, but the statement is unnecessary because type 2 CONNECT statements can be used instead.
- When a connection is used, made dormant, and then restored to the current state in the same unit of work, that connection reflects its last use by the application process with regard to the status of locks, cursors, and prepared statements.

## Examples

Execute SQL statements at IBMSTHDB, execute SQL statements at IBMTOKDB, and then execute more SQL statements at IBMSTHDB.

```
EXEC SQL CONNECT TO IBMSTHDB;  
/* Execute statements referencing objects at IBMSTHDB */  
  
EXEC SQL CONNECT TO IBMTOKDB;  
/* Execute statements referencing objects at IBMTOKDB */  
  
EXEC SQL SET CONNECTION IBMSTHDB;  
/* Execute statements referencing objects at IBMSTHDB */
```

Note that the first CONNECT statement creates the IBMSTHDB connection, the second CONNECT statement places it in the dormant state, and the SET CONNECTION statement returns it to the current state.

---

## SET CONSTRAINTS

The SET CONSTRAINTS statement is used to do one of the following:

- Turn off check constraint and referential constraint checking for one or more tables. If the table is a summary table with REFRESH IMMEDIATE, the immediate refreshing of the data is turned off. Note that this places the table(s) into a *check pending state* where only limited access by a restricted set of statements and commands is allowed. Primary key and unique constraints continue to be checked.
- Both turn the constraint checking back on for one or more tables and to carry out all the deferred checking. If the table is a summary table, the data is refreshed as necessary and, when defined with the REFRESH IMMEDIATE attribute, immediate refreshing of the data is turned on.
- Turn on referential constraints and/or check constraints for one or more tables without first carrying out any deferred constraint checking. If the table is a summary table defined with the REFRESH IMMEDIATE attribute, immediate refreshing of the data is turned on.
- Place the table into Check pending state if the table is already in DataLink Reconcile Pending (DRP) or DataLink Reconcile Not Possible (DRNP) state. If a table is not in either of those states, then unconditionally set the table in DRP state and Check-pending state.

The SET CONSTRAINTS statement is under transaction control.

### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

The privileges required to execute SET CONSTRAINTS depend on the use of the statement, as outlined below:

1. Turn off constraint checking.

The privileges of the authorization ID of the statement must include at least one of the following:

- CONTROL privilege on the tables and all their dependents and descendants in referential integrity constraints.
- SYSADM or DBADM authority.

2. Both turn on constraints and carry out checking.

The privileges of the authorization ID of the statement must include at least one of the following:

- SYSADM or DBADM authority.

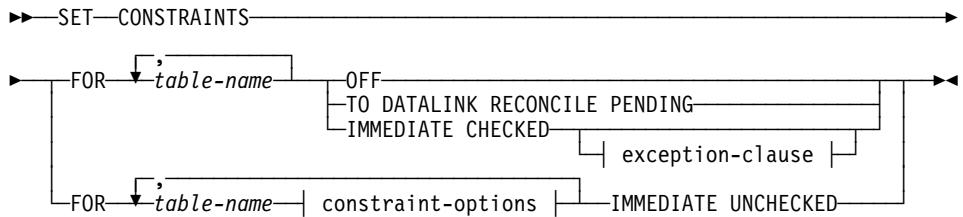
## SET CONSTRAINTS

- CONTROL privilege on the tables that are being checked **and** if exceptions are being posted to one or more tables, INSERT privilege on the exception tables.
3. Turn on constraints without first carrying out checking.

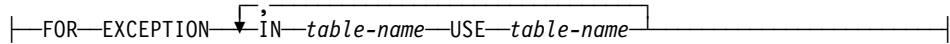
The authorization ID of the statement must have at least one of the following:

- SYSADM or DBADM authority
- CONTROL privilege on the tables that are being checked.

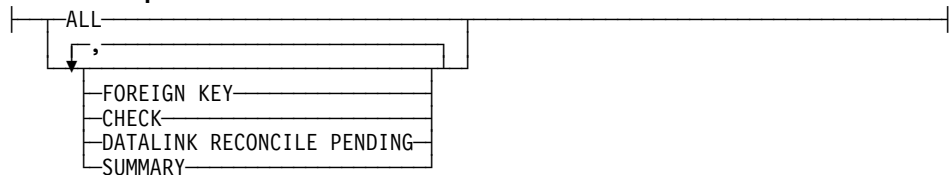
### Syntax



#### exception-clause:



#### constraint-options:



### Description

#### *table-name*

Identifies a base table for which constraint checking is to be turned either on or off or a summary table for which immediate refreshing is to be turned either on or off. It must be a table described in the catalog and must not be a view, catalog table or typed table. Only one summary table may be specified in the statement (SQLSTATE 42997).

#### OFF

Specifies that the tables are to have their foreign key constraints and check constraints turned off and are, therefore, to be placed into the check pending state. If it is a summary table, then immediate refreshing is turned off (if applicable) and the summary table is placed into check pending state.

Note that it is possible that a table may already be in the check pending state with only one type of constraint checking turned off; in such a situation the other type of constraint checking will also be turned off.

## SET CONSTRAINTS

If any table in the list is a parent table, the check pending state for foreign key constraints is extended to all dependent and descendent tables.

If any table in the list is an underlying table of a summary table, the check pending state is extended to such summary tables.

Only very limited activity is allowed on a table that is in the check pending state. "Notes" on page 713 lists the restrictions.

### TO DATALINK RECONCILE PENDING

Specifies that the tables are to have DATALINK integrity constraint checking turned off and the tables placed in check pending state. If the table is already in DataLink Reconcile Not Possible (DRNP) state, it remains in this state with check pending. Otherwise, the table is set to DataLink Reconcile Pending (DRP) state.

Dependent and descendent table are not affected when this option is specified.

### IMMEDIATE CHECKED

Specifies that the table is to have its constraints turned on and that the constraint checking that was deferred is to be carried out. This is done in accordance with the information set in the STATUS and CONST\_CHECKED columns of the SYSCAT.TABLES catalog. That is:

- The value in STATUS must be C (the table is in the check pending state) or an error (SQLSTATE 51027) is returned.
- The value in CONST\_CHECKED indicates which constraints are to be checked.

If it is a summary table, then the data is checked against the query and refreshed as necessary.

DATALINK values are not checked, even when the table is in DRP or DRNP state. The RECONCILE command or API should be used to perform the reconciliation of DATALINK values. The table will be taken out of check pending state but continue to have the DRP or DRNP flag set. This makes the table usable while the reconciliation of DATALINK values can be deferred to another time.

*exception-clause*

### FOR EXCEPTION

Indicates that any row that is in violation of a foreign key constraint or a check constraint will be copied to an exception table and deleted from the original table. See Appendix N, "Exception Tables" on page 967 for more information on these user-defined tables. Even if errors are detected the constraints are turned back on again and the table is taken out of the check pending state. A warning (SQLSTATE 01603) is issued to indicate that one or more rows have been moved to the exception tables.

If the FOR EXCEPTION clause is not specified and any constraints are violated, then only the first violation detected is returned to the user (SQLSTATE 23514). In the case of a violation in any table, all the tables are left in the check pending state, as they were before the execution of the statement. This

## SET CONSTRAINTS

clause cannot be specified if the *table-name* is a summary table (SQLSTATE 42997).

### **IN** *table-name*

Specifies the table from which rows that violate constraints are to be copied. There must be one exception table specified for each table being checked.

### **USE** *table-name*

Specifies the exception table into which error rows are to be copied.

### *constraint-options*

Used to define the constraint options that are set to IMMEDIATE UNCHECKED.

### **ALL**

This indicates that both foreign key constraints and check constraints are to be turned on.

### **FOREIGN KEY**

This indicates that foreign key constraints are to be turned on.

### **CHECK**

This indicates that check constraints are to be turned on.

### **DATALINK RECONCILE PENDING**

This indicates that DATALINK integrity constraints are to be turned on.

### **SUMMARY**

This indicates that the summary table should be refreshed and if the table has the REFRESH IMMEDIATE attribute, immediate refreshing should be turned on.

### **IMMEDIATE UNCHECKED**

Specifies one of the following:

- The table is to have its constraints turned on (and, thus, are to be taken out of the check pending state) without having the table checked for constraint violations or the summary table is to have immediate refreshing turned on and be taken out of check pending state.

This is specified for a given table either by specifying ALL, or by specifying CHECK when only check constraints are off for that table, or by specifying FOREIGN KEY when only foreign key constraints are off for that table, or by specifying DATALINK RECONCILE PENDING when only DATALINK integrity constraints are off for that table or by specifying SUMMARY when only summary table query checking is off for that summary table.

- The table is to have one type of constraint turned on but is to be left in the check pending state.

This is specified for a given table by specifying only CHECK, FOREIGN KEY, SUMMARY or DATALINK RECONCILE PENDING when any of those types of constraints are off for that table.

## SET CONSTRAINTS

The state change is not extended to any tables not explicitly included in the list.

If the parent of a dependent table is in the check pending state, the foreign key constraints of a dependent table cannot be marked to bypass checking (the check constraints checking can be bypassed).

The implications with respect to data integrity should be considered before using this option. See “Notes.”

### Notes

- Effects on tables in the check pending state:
  - Use of SELECT, INSERT, UPDATE, or DELETE is disallowed on a table that is either:
    - in the check pending state itself
    - or requires access to another table that is in the check pending state.

For example, a DELETE of a row in a parent table that cascades to a dependent table that is in the check pending state is not allowed.

  - New constraints added to a table are normally enforced immediately. However, if the table is in check pending state the checking of any new constraints is deferred until the table is taken out of the check pending state.
  - The CREATE INDEX statement cannot reference any tables that are in the check pending state. Similarly, ALTER TABLE to add a primary key or unique constraint cannot reference any tables that are in the check pending state.
  - The utilities EXPORT, IMPORT, REORG, and REORGCHK are not allowed to operate on a table in the check pending state. Note that the IMPORT utility differs from the LOAD utility in that it always checks the constraints immediately.
  - The utilities LOAD, BACKUP, RESTORE, ROLLFORWARD, UPDATE STATISTICS, RUNSTATS, LIST HISTORY, and ROLLFORWARD are allowed on a table in the check pending state.
  - The statements ALTER TABLE, COMMENT ON, DROP TABLE, CREATE ALIAS, CREATE TRIGGER, CREATE VIEW, GRANT, REVOKE, and SET CONSTRAINTS can reference a table that is in the check pending state.
  - Packages, views and any other objects that depend on a table that is in the check pending state will return an error when the table is accessed at run time.

The removal of violating rows by the SET CONSTRAINTS statement is not a delete event. Therefore, triggers are never activated by a SET CONSTRAINTS statement.

- Warning about the use of the IMMEDIATE UNCHECKED clause:

This clause is intended to be used by utility programs and its use by application programs is not recommended.

## SET CONSTRAINTS

The fact that constraints were turned on without doing deferred checking will be recorded in the catalog (the value in the CONST\_CHECKED column in the SYSCAT.TABLES view will be set to 'U'). This indicates that the user has assumed responsibility for data integrity with respect to the specific constraints. This value remains until either:

- The table is put back into the check pending state (by referencing the table in a SET CONSTRAINTS statement with the OFF clause).
  - All unchecked constraints for the table are dropped.
  - A REFRESH TABLE statement is issued for a summary table.
- A table that is in DataLink Reconcile Not Possible (DRNP) state requires corrective action to be taken (possibly outside of the database). Once corrective action is completed, the table is taken out of DRNP state using the IMMEDIATE UNCHECKED option. The RECONCILE command or API should then be used to check the DATALINK integrity constraints. For more details refer on removing a table from DataLink Reconcile Not Possible state refer to *DataLinks Administration and Reference*.
  - While constraints are being checked an exclusive lock is held on each table specified in the SET CONSTRAINTS invocation.
  - A shared lock on each table that is not listed in the SET CONSTRAINTS invocation but is a parent table of one of the dependent tables being checked.
  - If an error occurs during constraint checking, all the effects of the checking including deleting from the original and inserting into the exception tables will be rolled back.

### Example

*Example 1:* The following is an example of a query that gives us information about the check pending state of tables. SUBSTR is used to extract the first 2 bytes of the CONST\_CHECKED column of SYSCAT.TABLES. The first byte represents foreign key constraints, and the second byte represents check constraints.

```
SELECT TABNAME,  
       SUBSTR( CONST_CHECKED, 1, 1 ) AS FK_CHECKED,  
       SUBSTR( CONST_CHECKED, 2, 1 ) AS CC_CHECKED  
FROM SYSCAT.TABLES  
WHERE STATUS = 'C';
```

*Example 2:* Set tables T1 and T2 in the check pending state:

```
SET CONSTRAINTS FOR T1, T2 OFF;
```

*Example 3:* Check the constraints for T1 and get the first violation only:

```
SET CONSTRAINTS FOR T1 IMMEDIATE CHECKED
```

*Example 4:* Check the constraints for T1 and T2 and put the violating rows into exception tables E1 and E2:

```
SET CONSTRAINTS FOR T1, T2 IMMEDIATE CHECKED  
FOR EXCEPTION IN T1 USE E1,  
IN T2 USE E2;
```



## SET CONSTRAINTS

*Example 5:* Enable FOREIGN KEY constraint checking in T1 and CHECK constraint checking in T2 to be bypassed with the IMMEDIATE CHECKED option:

```
SET CONSTRAINTS FOR T1 FOREIGN KEY,  
T2 CHECK IMMEDIATE UNCHECKED;
```

*Example 6:* Add a check constraint and a foreign key to the EMP\_ACT table, using two ALTER TABLE statements. To perform constraint checking in a single pass of the table, constraint checking is turned off before the ALTER statements and checked after execution.

```
SET CONSTRAINTS FOR EMP_ACT OFF;  
ALTER TABLE EMP_ACT ADD CHECK (EMSTDATE <= EMENDATE);  
ALTER TABLE EMP_ACT ADD FOREIGN KEY (EMPNO) REFERENCES EMPLOYEE;  
SET CONSTRAINTS FOR EMP_ACT IMMEDIATE CHECKED;
```

## SET CURRENT DEGREE

---

### SET CURRENT DEGREE

The SET CURRENT DEGREE statement assigns a value to the CURRENT DEGREE special register. This statement is not under transaction control.

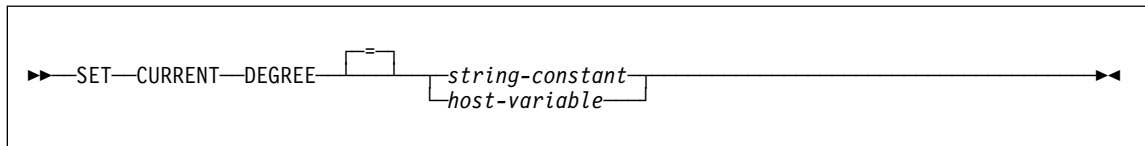
#### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

#### Authorization

No authorization is required to execute this statement.

#### Syntax



#### Description

The value of CURRENT DEGREE is replaced by the value of the string constant or host variable. The value must be a character string that is not longer than 5 bytes. The value must be the character string representation of an integer between 1 and 32767 inclusive or 'ANY'.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intra-partition parallelism .

If the value of CURRENT DEGREE is a number when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism with the specified degree.

If the value of CURRENT DEGREE is 'ANY' when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism using a degree determined by the database manager.

##### *host-variable*

The *host-variable* must be of data type CHAR or VARCHAR and the length must not exceed 5. If a longer field is provided, an error will be returned (SQLSTATE 42815). If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If a *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

## SET CURRENT DEGREE

*string-constant*

The *string-constant* length must not exceed 5.

### Notes

The degree of intra-partition parallelism for static SQL statements can be controlled using the DEGREE option of the PREP or BIND command. Refer to the *Command Reference* for details on these commands.

The actual runtime degree of intra-partition parallelism will be the lower of:

- Maximum query degree (max\_querydegree) configuration parameter
- Application runtime degree
- SQL statement compilation degree

The intra\_parallel database manager configuration must be on to use intra-partition parallelism . If it is set to off, the value of this register will be ignored and the statement will not use intra-partition parallelism for the purpose of optimization (SQLSTATE 01623).

Some SQL statements cannot use intra-partition parallelism . See the *Administration Guide* for a description of degree of intra-partition parallelism and a list of restrictions.

### Example

*Example 1:* The following statement sets the CURRENT DEGREE to inhibit intra-partition parallelism .

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

*Example 2:* The following statement sets the CURRENT DEGREE to allow intra-partition parallelism .

```
SET CURRENT DEGREE = 'ANY'
```

## SET CURRENT EXPLAIN MODE

---

### SET CURRENT EXPLAIN MODE

The SET CURRENT EXPLAIN MODE statement changes the value of the CURRENT EXPLAIN MODE special register. It is not under transaction control.

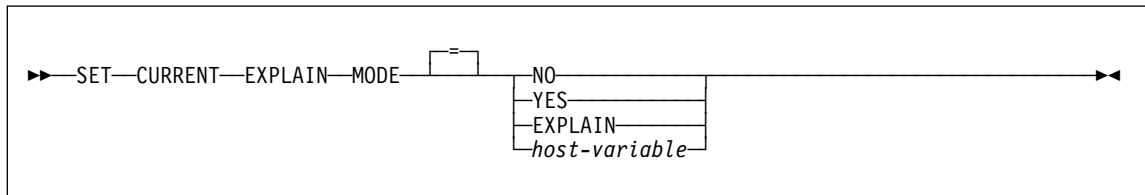
#### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

#### Authorization

No special authorization is required to execute this statement.

#### Syntax



#### Description

##### NO

Disables the Explain facility. No Explain information is captured. NO is the initial value of the special register.

##### YES

Enables the Explain facility and causes Explain information to be inserted into the Explain tables for eligible dynamic SQL statements. All dynamic SQL statements are compiled and executed normally.

##### EXPLAIN

Enables the Explain facility and causes Explain information to be captured for any eligible dynamic SQL statement that is prepared. However, dynamic statements are not executed.

##### *host-variable*

The *host-variable* must be of data type CHAR or VARCHAR and the length must not exceed 8. If a longer field is provided, an error will be returned (SQLSTATE 42815). The value specified must be NO, YES, or EXPLAIN. If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If a *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

## SET CURRENT EXPLAIN MODE

### Notes

Explain information for static SQL statements can be captured by using the EXPLAIN option of the PREP or BIND command. If the ALL value of the EXPLAIN option is specified, and the CURRENT EXPLAIN MODE register value is NO, explain information will be captured for dynamic SQL statements at runtime. If the value of the CURRENT EXPLAIN MODE register is not NO, then the value of the EXPLAIN bind option is ignored. For more information on the interaction between the EXPLAIN option and the CURRENT EXPLAIN MODE special register, see Table 109 on page 958.

If the Explain facility is activated, the current authorization ID must have INSERT privilege for the Explain tables or an error (SQLSTATE 42501) is raised.

For further information, see the *Administration Guide*.

### Example

*Example 1:* The following statement sets the CURRENT EXPLAIN MODE special register, so that Explain information will be captured for any subsequent eligible dynamic SQL statements and the statement will not be executed.

```
SET CURRENT EXPLAIN MODE = EXPLAIN
```

## SET CURRENT EXPLAIN SNAPSHOT

---

### SET CURRENT EXPLAIN SNAPSHOT

The SET CURRENT EXPLAIN SNAPSHOT statement changes the value of the CURRENT EXPLAIN SNAPSHOT special register. It is not under transaction control.

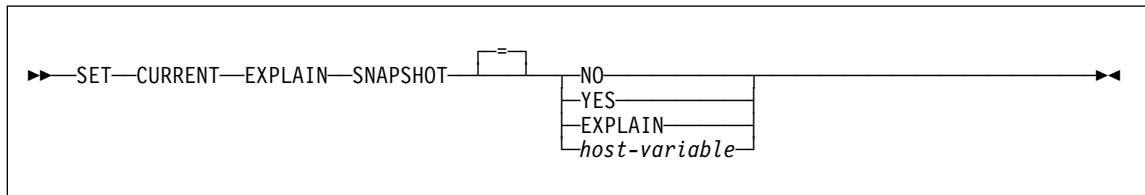
#### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

#### Authorization

No authorization is required to execute this statement.

#### Syntax



#### Description

##### NO

Disables the Explain snapshot facility. No snapshot is taken. NO is the initial value of the special register.

##### YES

Enables the Explain snapshot facility, creating a snapshot of the internal representation for each eligible dynamic SQL statement. This information is inserted in the SNAPSHOT column of the EXPLAIN\_STATEMENT table (see Appendix K, “Explain Tables and Definitions” on page 935).

The EXPLAIN SNAPSHOT facility is intended for use with Visual Explain.

##### EXPLAIN

Enables the Explain snapshot facility, creating a snapshot of the internal representation for each eligible dynamic SQL statement that is prepared. However, dynamic statements are not executed.

##### *host-variable*

The *host-variable* must be of data type CHAR or VARCHAR and the length of its contents must not exceed 8. If a longer field is provided, an error will be returned (SQLSTATE 42815). The value contained in this register must be either NO, YES, or EXPLAIN. If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive.

## SET CURRENT EXPLAIN SNAPSHOT

If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

### Notes

Explain snapshots for static SQL statements can be captured by using the EXPLSNAP option of the PREP or BIND command. If the ALL value of the EXPLSNAP option is specified, and the CURRENT EXPLAIN SNAPSHOT register value is NO, Explain snapshots will be captured for dynamic SQL statements at runtime. If the value of the CURRENT EXPLAIN SNAPSHOT register is not NO, then the EXPLSNAP option is ignored. For more information on the interaction between the EXPLSNAP option and the CURRENT EXPLAIN SNAPSHOT special register, see Table 110 on page 959.

If the Explain snapshot facility is activated, the current authorization ID must have INSERT privilege for the Explain tables or an error (SQLSTATE 42501) is raised.

For further information, see the *Administration Guide*.

### Example

*Example 1:* The following statement sets the CURRENT EXPLAIN SNAPSHOT special register, so that an Explain snapshot will be taken for any subsequent eligible dynamic SQL statements and the statement will be executed.

```
SET CURRENT EXPLAIN SNAPSHOT = YES
```

*Example 2:* The following example retrieves the current value of the CURRENT EXPLAIN SNAPSHOT special register into the host variable called SNAP.

```
EXEC SQL VALUES (CURRENT EXPLAIN SNAPSHOT) INTO :SNAP;
```

## SET CURRENT PACKAGESET

---

### SET CURRENT PACKAGESET

The SET CURRENT PACKAGESET statement sets the schema name (collection identifier) that will be used to select the package to use for subsequent SQL statements. This statement is not under transaction control.

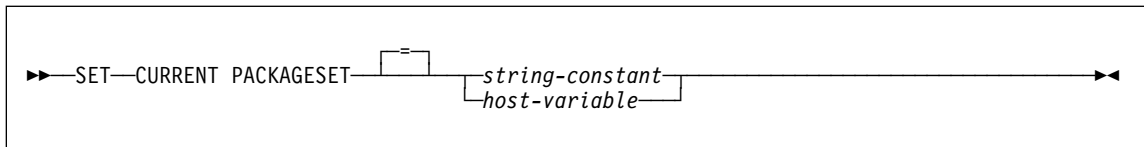
#### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared. This statement is not supported in REXX.

#### Authorization

None required.

#### Syntax



#### Description

##### *string-constant*

A character string constant with a maximum length of 8. If more than the maximum, it will be truncated at run time.

##### *host-variable*

A variable of type CHAR or VARCHAR with a maximum length of 8. It cannot be set to null. If more than the maximum, it will be truncated at run time.

#### Notes

- This statement allows an application to specify the schema name used when selecting a package for an executable SQL statement. The statement is processed at the client and does not flow to the application server.
- The COLLECTION bind option can be used to create a package with a specified schema name. See the *Command Reference* for details.
- Unlike DB2 for MVS/ESA, the SET CURRENT PACKAGESET statement is implemented without support for a special register called CURRENT PACKAGESET.

#### Example

Assume an application called TRYIT is precompiled by userid PRODUSA, making 'PRODUSA' the default schema name in the bind file. The application is then bound twice with different bind options. The following command line processor commands were used:



## SET CURRENT PACKAGESET

```
DB2 CONNECT TO SAMPLE USER PRODUSA
DB2 BIND TRYIT.BND DATETIME USA
DB2 CONNECT TO SAMPLE USER PRODEUR
DB2 BIND TRYIT.BND DATETIME EUR COLLECTION 'PRODEUR'
```

This creates two packages called TRYIT. The first bind command created the package in the schema named 'PRODUSA'. The second bind command created the package in the schema named 'PRODEUR' based on the COLLECTION option.

Assume the application TRYIT contains the following statements:

```
EXEC SQL CONNECT TO SAMPLE;
.
.
EXEC SQL SELECT HIREDATE INTO :HD FROM EMPLOYEE WHERE EMPNO='000010'; 1
.
.
EXEC SQL SET CURRENT PACKAGESET 'PRODEUR'; 2
.
.
EXEC SQL SELECT HIREDATE INTO :HD FROM EMPLOYEE WHERE EMPNO='000010'; 3
```

- 1** This statement will run using the PRODUSA.TRYIT package because it is the default package for the application. The date is therefore returned in USA format.
- 2** This statement sets the schema name to 'PRODEUR' for package selection.
- 3** This statement will run using the PRODEUR.TRYIT package as a result of the SET CURRENT PACKAGESET statement. The date is therefore returned in EUR format.

# SET CURRENT QUERY OPTIMIZATION

---

## SET CURRENT QUERY OPTIMIZATION

The SET CURRENT QUERY OPTIMIZATION statement assigns a value to the CURRENT QUERY OPTIMIZATION special register. The value specifies the current class of optimization techniques enabled when preparing dynamic SQL statements. It is not under transaction control.

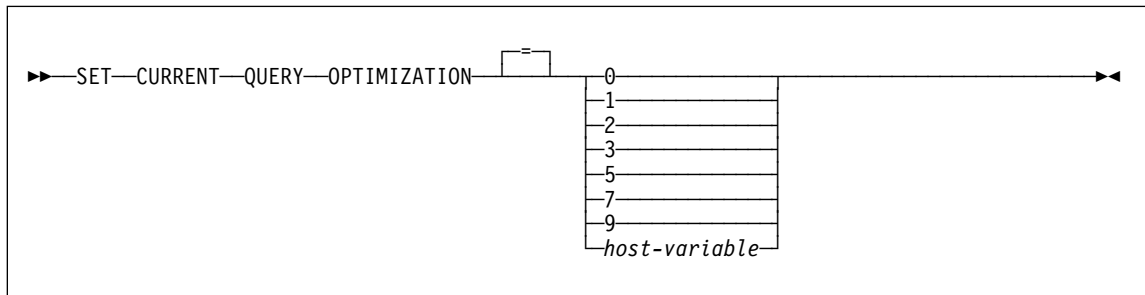
### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

No authorization is required to execute this statement.

### Syntax



### Description

#### *optimization-class*

*optimization-class* can be specified either as an integer constant or as the name of a host variable that will contain the appropriate value at run time. An overview of the classes follows (for details refer to the *Administration Guide* ).

- 0** Specifies that a minimal amount of optimization is performed to generate an access plan. This class is most suitable for simple dynamic SQL access to well-indexed tables.
- 1** Specifies that optimization roughly comparable to DB2 Version 1 is performed to generate an access plan.
- 2** Specifies a level of optimization higher than that of DB2 Version 1, but at significantly less optimization cost than levels 3 and above, especially for very complex queries.
- 3** Specifies that a moderate amount of optimization is performed to generate an access plan.

## SET CURRENT QUERY OPTIMIZATION

5	Specifies a significant amount of optimization is performed to generate an access plan. For complex dynamic SQL queries, heuristic rules are used to limit the amount of time spent selecting an access plan. Where possible, queries will use summary tables instead of the underlying base tables.
7	Specifies a significant amount of optimization is performed to generate an access plan. Similar to 5 but without the heuristic rules.
9	Specifies a maximal amount of optimization is performed to generate an access plan. This can greatly expand the number of possible access plans that are evaluated. This class should be used to determine if a better access plan can be generated for very complex and very long-running queries using large tables. Explain and performance measurements can be used to verify that a better plan has been generated.
<i>host-variable</i>	The data type is INTEGER. The value must be in the range 0 to 9 (SQLSTATE 42815) but should be 0, 1, 2, 3, 5, 7, or 9 (SQLSTATE 01608). If <i>host-variable</i> has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

### Notes

- When the CURRENT QUERY OPTIMIZATION register is set to a particular value, a set of query rewrite rules are enabled, and certain optimization variables take on particular values. This class of optimization techniques is then used during preparation of dynamic SQL statements.
- In general, changing the optimization class impacts the execution time of the application, the compilation time, and resources required. Most statements will be adequately optimized using the default query optimization class. Lower query optimization classes, especially classes 1 and 2, may be appropriate for dynamic SQL statements for which the resources consumed by the dynamic *PREPARE* are a significant portion of those required to execute the query. Higher optimization classes should be chosen only after considering the additional resources that may be consumed and verifying that a better access plan has been generated. For additional detail on the behavior associated with each query optimization class see *Administration Guide*.
- Query optimization classes must be in the range 0 to 9. Classes outside this range will return an error (SQLSTATE 42815). Unsupported classes within this range will return a warning (SQLSTATE 01608) and will be replaced with the next lowest query optimization class. For example, a query optimization class of 6 will be replaced by 5.
- Dynamically prepared statements use the class of optimization that was set by the most recently executed SET CURRENT QUERY OPTIMIZATION statement. In cases where a SET CURRENT QUERY OPTIMIZATION statement has not yet

## SET CURRENT QUERY OPTIMIZATION

been executed, the query optimization class is determined by the value of the database configuration parameter, `dft_queryopt`.

- Statically bound statements do not use the CURRENT QUERY OPTIMIZATION special register; therefore this statement has no effect on them. The QUERYOPT option is used during preprocessing or binding to specify the desired class of optimization for statically bound statements. If QUERYOPT is not specified then, the default value specified by the database configuration parameter, `dft_queryopt`, is used. Refer to the BIND command in the *Command Reference* for details.
- The results of executing the SET CURRENT QUERY OPTIMIZATION statement are not rolled back if the unit of work in which it is executed is rolled back.

### Examples

*Example 1:* This example shows how the highest degree of optimization can be selected.

```
SET CURRENT QUERY OPTIMIZATION 9
```

*Example 2:* The following example shows how the CURRENT QUERY OPTIMIZATION special register can be used within a query.

Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```
EXEC SQL DECLARE C1 CURSOR FOR
SELECT PKGNAME, PKGSHEMA FROM SYSCAT.PACKAGES
WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```

## SET CURRENT REFRESH AGE

### SET CURRENT REFRESH AGE

The SET CURRENT REFRESH AGE statement changes the value of the CURRENT REFRESH AGE special register. It is not under transaction control.

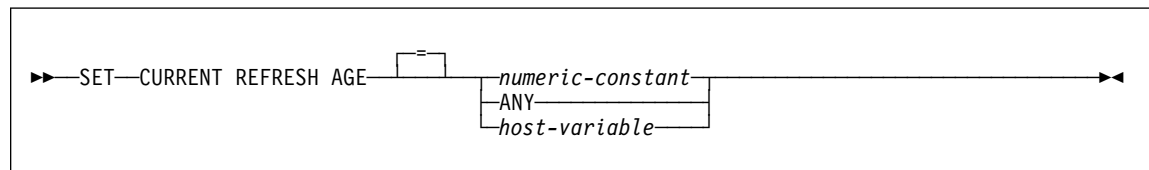
### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

### Authorization

No authorization is required to execute this statement.

### Syntax



### Description

#### *numeric-constant*

A DECIMAL(20,6) value representing a timestamp duration. The value must be 0 or 9999999999999999 (the microseconds portion of the value is ignored and can therefore be any value).

**0**

Indicates that only summary tables defined with REFRESH IMMEDIATE may be used to optimize the processing of a query.

**9999999999999999**

Indicates that any summary tables defined with REFRESH DEFERRED or REFRESH IMMEDIATE may be used to optimize the processing of a query. This value represents 9999 years, 99 months, 99 days, 99 hours, 99 minutes, and 99 seconds.

#### **ANY**

This is a shorthand for 9999999999999999.

#### *host-variable*

A variable of type DECIMAL(20,6) or other type that is assignable to DECIMAL(20,6). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815). The value of the host-variable must be 0 or 9999999999999999.000000.

## SET CURRENT REFRESH AGE

### Notes

- The initial value of the CURRENT REFRESH AGE special register is zero.
- Setting the CURRENT REFRESH AGE special register to a value other than zero should be done with caution. By allowing a summary table that may not represent the values of the underlying base table to be used to optimize the processing of the query, the result of the query may NOT accurately represent the data in the underlying table. This may be reasonable when you know the underlying data has not changed or you are willing to accept the degree of error in the results based on your knowledge of the data.
- The CURRENT REFRESH AGE value of 9999999999999999 cannot be used in timestamp arithmetic operations since the result would be outside the valid range of dates (SQLSTATE 22008).

### Examples

*Example 1:* The following statement sets the CURRENT REFRESH AGE special register.

```
SET CURRENT REFRESH AGE ANY
```

*Example 2:*

The following example retrieves the current value of the CURRENT REFRESH AGE special register into the host variable called CURMAXAGE.

```
EXEC SQL VALUES (CURRENT REFRESH AGE) INTO :CURMAXAGE;
```

The value would be 9999999999999999.000000, set by the previous example.

## SET EVENT MONITOR STATE

The SET EVENT MONITOR STATE statement activates or deactivates an event monitor. The current state of an event monitor (active or inactive) is determined by using the EVENT\_MON\_STATE built-in function. The SET EVENT MONITOR STATE statement is not under transaction control.

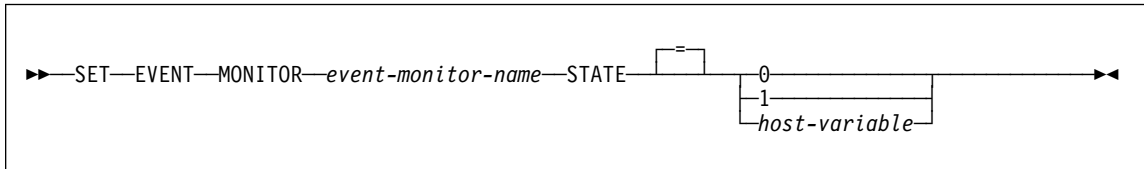
### Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

The authorization ID of the statement must hold either SYSADM or DBADM authority (SQLSTATE 42815).

### Syntax



### Description

#### *event-monitor-name*

Identifies the event monitor to activate or deactivate. The name must identify an event monitor that exists in the catalog (SQLSTATE 42704).

#### *new-state*

*new-state* can be specified either as an integer constant or as the name of a host variable that will contain the appropriate value at run time. The following may be specified:

- 0** Indicates that the specified event monitor should be deactivated.
- 1** Indicates that the specified event monitor should be activated. The event monitor should not already be active; otherwise a warning (SQLSTATE 01598) is issued.

#### *host-variable*

The data type is INTEGER. The value specified must be 0 or 1 (SQLSTATE 42815). If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

## SET EVENT MONITOR STATE

### Rules

- Although an unlimited number of event monitors may be defined, there is a limit of 32 event monitors that can be simultaneously active (SQLSTATE 54030).
- In order to activate an event monitor, the transaction in which the event monitor was created must have been committed (SQLSTATE 55033). This rule prevents (in one unit of work) creating an event monitor, activating the monitor, then rolling back the transaction.
- If the number or size of the event monitor files exceeds the values specified for MAXFILES or MAXFILESIZE on the CREATE EVENT MONITOR statement, an error (SQLSTATE 54031) is raised.
- If the target path of the event monitor (that was specified on the CREATE EVENT MONITOR statement) is already in use by another event monitor, an error (SQLSTATE 51026) is raised.

### Notes

- Activating an event monitor performs a reset of any counters associated with it.

### Example

The following example activates an event monitor called SMITHPAY.

```
SET EVENT MONITOR SMITHPAY STATE = 1
```



## SET PATH

The SET PATH statement changes the value of the CURRENT PATH special register. It is not under transaction control.

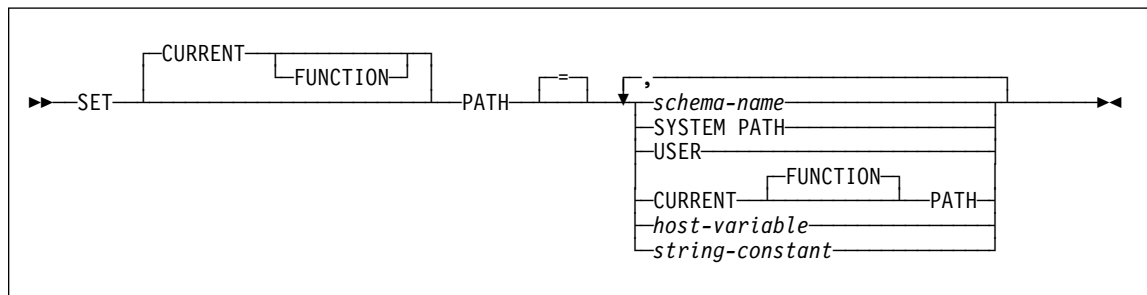
### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

### Authorization

No authorization is required to execute this statement.

### Syntax



### Description

#### *schema-name*

This one-part name identifies a schema that exists at the application server. It must be a short SQL identifier (SQLSTATE 42815). No validation that the schema exists is made at the time that the path is set. If a *schema-name* is, for example, misspelled, it will not be caught, and it could affect the way subsequent SQL operates.

#### **SYSTEM PATH**

This value is the same as specifying the schema names "SYSIBM", "SYSFUN".

#### **USER**

The value in the USER special register.

#### **CURRENT PATH**

The value of the CURRENT PATH before the execution of this statement. CURRENT FUNCTION PATH may also be specified.

#### *host-variable*

A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 8 (SQLSTATE 42815). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

## SET PATH

The characters of the *host-variable* must be left justified. When specifying the *schema-name* with a *host-variable*, all characters must be specified in the exact case intended as there is no conversion to uppercase characters.

### *string-constant*

A character string constant with a maximum length of 8.

## Rules

- A schema name cannot appear more than once in the function path (SQLSTATE 42732).
- The number of schemas that can be specified is limited by the total length of the CURRENT PATH special register. The special register string is built by taking each schema name specified and removing trailing blanks, delimiting with double quotes, doubling quotes within the schema name as necessary, and then separating each schema name by a comma. The length of the resulting string cannot exceed 254 bytes (SQLSTATE 42907).

## Notes

- The initial value of the CURRENT PATH special register is "SYSIBM","SYSFUN","X" where X is the value of the USER special register.
- The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed as the first schema (in this case, it is not included in the CURRENT PATH special register).
- The CURRENT PATH special register specifies the SQL path used to resolve user-defined data types, procedures and functions in dynamic SQL statements. The FUNCPATH bind option specifies the SQL path to be used for resolving user-defined data types and functions in static SQL statements. See the *Command Reference* for further information on the use of FUNCPATH option in BIND command.

## Example

*Example 1:* The following statement sets the CURRENT FUNCTION PATH special register.

```
SET PATH = FERMAT, "McDrw #8", SYSIBM
```

*Example 2:* The following example retrieves the current value of the CURRENT PATH special register into the host variable called CURPATH.

```
EXEC SQL VALUES (CURRENT PATH) INTO :CURPATH;
```

The value would be "FERMAT","McDrw #8","SYSIBM" if set by the previous example.

## SET SCHEMA

The SET SCHEMA statement changes the value of the CURRENT SCHEMA special register. It is not under transaction control.

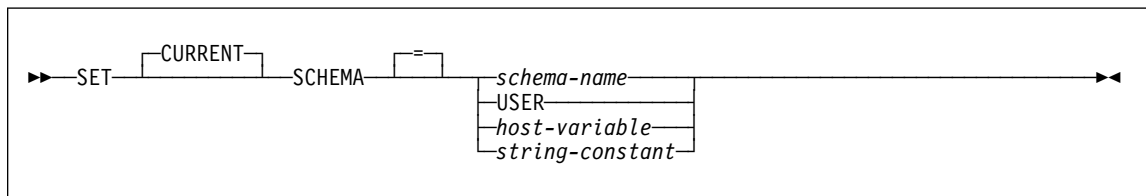
### Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

### Authorization

No authorization is required to execute this statement.

### Syntax



### Description

#### *schema-name*

This one-part name identifies a schema that exists at the application server. It must be a short SQL identifier (SQLSTATE 42815). No validation that the schema exists is made at the time that the schema is set. If a *schema-name* is misspelled, it will not be caught, and it could affect the way subsequent SQL operates.

#### **USER**

The value in the USER special register.

#### *host-variable*

A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 8 (SQLSTATE 42815). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

The characters of the *host-variable* must be left justified. When specifying the *schema-name* with a *host-variable*, all characters must be specified in the exact case intended as there is no conversion to uppercase characters.

#### *string-constant*

A character string constant with a maximum length of 8.

## SET SCHEMA

### Rules

- If the value specified does not conform to the rules for a *schema-name*, an error (SQLSTATE 3F000) is raised.
- The value of the CURRENT SCHEMA special register is used as the schema name in all dynamic SQL statements, with the exception of the CREATE SCHEMA statement, where an unqualified reference to a database object exists.
- The QUALIFIER bind option specifies the schema name for use as the qualifier for unqualified database object names in static SQL statements (see the *Command Reference* for further information on use of the QUALIFIER option).

### Notes

- The initial value of the CURRENT SCHEMA special register is equivalent to USER.
- Setting the CURRENT SCHEMA special register does not effect the CURRENT PATH special register. Hence, the CURRENT SCHEMA will not be included in the SQL path and functions, procedures and user-defined type resolution may not find these objects. To include the current schema value in the SQL path, whenever the SET SCHEMA statement is issued, also issue the SET PATH statement including the schema name from the SET SCHEMA statement.
- CURRENT SQLID is accepted as a synonym for CURRENT SCHEMA and the effect of a SET CURRENT SQLID statement will be identical to that of a SET CURRENT SCHEMA statement. No other effects, such as statement authorization changes, will occur.

### Examples

*Example 1:* The following statement sets the CURRENT SCHEMA special register.

```
SET SCHEMA RICK
```

*Example 2:* The following example retrieves the current value of the CURRENT SCHEMA special register into the host variable called CURSCHEMA.

```
EXEC SQL VALUES (CURRENT SCHEMA) INTO :CURSCHEMA;
```

The value would be RICK, set by the previous example.

## SET transition-variable

The SET transition-variable statement assigns values to new transition variables. It is under transaction control.

### Invocation

This statement can only be used as a triggered SQL statement in the triggered action of a BEFORE trigger whose granularity is FOR EACH ROW (see “CREATE TRIGGER” on page 568).

### Authorization

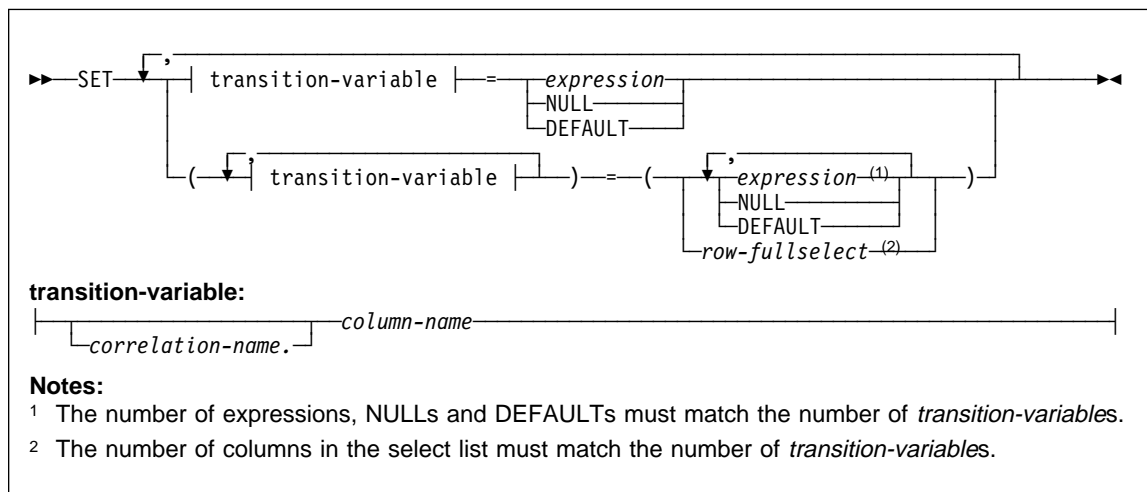
The privileges held by the authorization ID of the creator of the trigger must include at least one of the following:

- UPDATE of the columns referenced on the left hand side of the assignment and SELECT for any columns referenced on the right hand side.
- CONTROL privilege on the table (subject table of the trigger)
- SYSADM or DBADM authority.

To execute this statement with a *row-fullselect* as the right hand side of the assignment, the privileges held by the authorization ID of the creator of the trigger must also include at least one of the following for each table or view referenced:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM.

### Syntax



## SET transition-variable

### Description

#### **transition-variable**

Identifies a column in the set of affected rows for the trigger.

#### *correlation-name*

The correlation-name given for referencing the NEW transition variables. This correlation-name must match the correlation name specified following NEW in the REFERENCING clause of the CREATE TRIGGER.

If OLD is not specified in the REFERENCING clause, the correlation-name will default to the correlation-name specified following NEW. If both NEW and OLD are specified in the REFERENCING clause, then a correlation-name is required with each column-name (SQLSTATE 42702).

#### *column-name*

Identifies the column to be updated. The *column-name* must identify a column of the subject table of the trigger (SQLSTATE 42703). A column must not be specified more than once (SQLSTATE 42701).

#### *expression*

Indicates the new value of the column. The expression is any expression of the type described in “Expressions” on page 117. The expression can not include a column function except when it occurs within a scalar fullselect (SQLSTATE 42903). An *expression* may contain references to OLD and NEW transition variables and must be qualified by the *correlation-name* to specify which transition variable (SQLSTATE 42702).

#### **NULL**

Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502).

#### **DEFAULT**

Specifies that the default value should be used based on how the corresponding column is defined in the table. The value that is inserted depends on how the column was defined.

- If the column was defined using the WITH DEFAULT clause, then the value is set to the default defined for the column (see default-clause in “ALTER TABLE” on page 380).
- If the column was defined without specifying the WITH DEFAULT clause or the NOT NULL clause, then the value inserted is NULL.
- If the column was defined using the NOT NULL clause and the WITH DEFAULT clause was not used or DEFAULT NULL was used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

#### *row-fullselect*

A fullselect that returns a single row with the number of columns corresponding to the number of column-names specified for assignment. The values are assigned to each corresponding column-name. If the result of the row-fullselect is no rows, then null values are assigned. A *row-fullselect* may contain references to OLD and NEW transition variables which must be qualified by the *correlation-name* to specify

## SET transition-variable

which transition variable to use. (SQLSTATE 42702). An error is returned if there is more than one row in the result (SQLSTATE 21000).

### Rules

- The number of values to be assigned from expressions, NULLs and DEFAULTs or the *row-fullselect* must match the number of columns specified for assignment (SQLSTATE 42802).
- If the statement is used in a BEFORE UPDATE trigger, the *column-name* specified as a *transition-variable* cannot be a partitioning key column (SQLSTATE 42997).

### Notes

- If more than one assignment is included, all the *expressions* and *row-fullselects* are evaluated before the assignments are performed. Thus references to columns in an expression or row fullselect are always the value of the transition variable prior to any assignment in the single SET transition-variable statement.

### Examples

*Example 1:* Set the salary column of the row for which the trigger action is currently executing to 50000.

```
SET NEW_VAR.SALARY = 50000;  
or  
SET (NEW_VAR.SALARY) = (50000);
```

*Example 2:* Set the salary and the commission column of the row for which the trigger action is currently executing to 50000 and 8000 respectively.

```
SET NEW_VAR.SALARY = 50000, NEW_VAR.COMM = 8000;  
or  
SET (NEW_VAR.SALARY, NEW_VAR.COMM) = (50000, 8000);
```

*Example 3:* Set the salary and the commission column of the row for which the trigger action is currently executing to the average of the salary and of the commission of the employees of the updated row's department respectively.

```
SET (NEW_VAR.SALARY, NEW_VAR.COMM)  
= (SELECT AVG(SALARY), AVG(COMM)  
FROM EMPLOYEE E  
WHERE E.WORKDEPT = NEW_VAR.WORKDEPT);
```

*Example 4:* Set the salary and the commission column of the row for which the trigger action is currently executing to 10000 and the original value of salary respectively (i.e., before the SET statement was executed).

```
SET NEW_VAR.SALARY = 10000, NEW_VAR.COMM = NEW_VAR.SALARY;  
or  
SET (NEW_VAR.SALARY, NEW_VAR.COMM) = (10000, NEW_VAR.SALARY);
```

## SIGNAL SQLSTATE

---

### SIGNAL SQLSTATE

The SIGNAL SQLSTATE statement is used to signal an error. It causes an error to be returned with the specified SQLSTATE and the specified *diagnostic-string*.

#### Invocation

The SIGNAL SQLSTATE statement can only be used as a triggered SQL statement within a trigger.

#### Authorization

No authorization is required to execute this statement.

#### Syntax

```
►—SIGNAL—SQLSTATE—string-constant—(—diagnostic-string—)————►
```

#### Description

##### *string-constant*

The specified *string-constant* represents an SQLSTATE. It must be a character string constant with exactly 5 characters that follow the rules for application-defined SQLSTATEs as follows:

- 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' since these are not error classes.
- If the SQLSTATE class (first two characters) starts with the character '0' through '6' or 'A' through 'H', then 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', then the subclass (last three characters) can be any of '0' through '9' or 'A' through 'Z'.

If the SQLSTATE does not conform to these rules an error occurs (SQLSTATE 428B3).

##### *diagnostic-string*

An expression with a type of CHAR or VARCHAR that returns a character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.



### Example

Consider an order system that records orders in an ORDERS table (ORDERNO, CUSTNO, PARTNO, QUANTITY) only if there is sufficient stock in the PARTS tables.

```
CREATE TRIGGER check_avail
NO CASCADE BEFORE INSERT ON orders
REFERENCING NEW AS new_order
FOR EACH ROW MODE DB2SQL
WHEN (new_order.quantity > (SELECT on_hand FROM parts
                             WHERE new_order.partno=parts.partno))
BEGIN ATOMIC
  SIGNAL SQLSTATE '75001' ('Insufficient stock for order');
END
```

## UPDATE

---

### 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 its base table.

The forms of this statement are:

- 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

An UPDATE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- UPDATE privilege on the table or view where rows are to be updated
- UPDATE privilege on each of the columns to be updated.
- CONTROL privilege on the table or view where rows are to be updated
- SYSADM or DBADM authority.
- If a *row-fullselect* is included in the assignment, at least one of the following for each referenced table or view:
  - SELECT privilege
  - CONTROL privilege
  - SYSADM or DBADM authority.

For each table or view referenced by a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

When the package is precompiled with SQL92 rules<sup>85</sup> and the searched form of an UPDATE includes a reference to a column of the table or view in the right side of the *assignment-clause* or anywhere in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following:

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

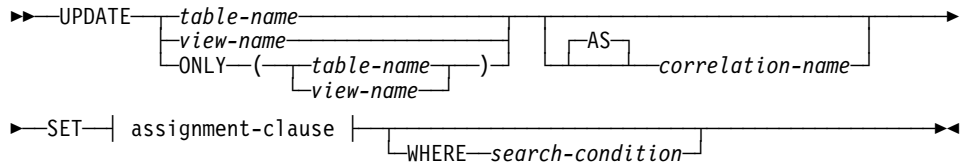
---

<sup>85</sup> The package used to process the statement is precompiled using option LANGLEVEL with value SQL92E or MIA.

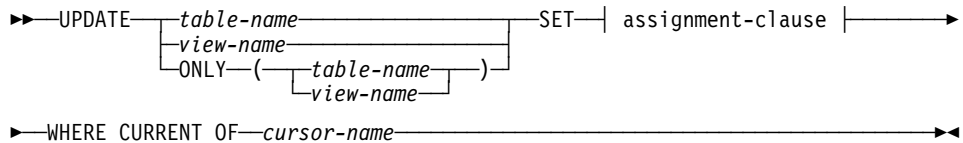
GROUP privileges are not checked for static UPDATE statements.

## Syntax

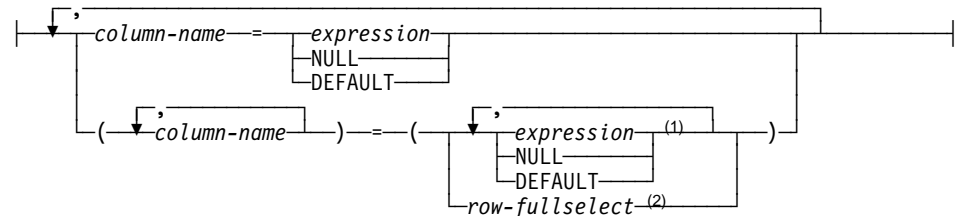
### Searched UPDATE:



### Positioned UPDATE:



### assignment-clause:



### Notes:

- 1 The number of expressions, NULLs and DEFAULTs must match the number of *column-names*.
- 2 The number of columns in the select list must match the number of *column-names*.

## Description

### *table-name* or *view-name*

Is the name of the table or view to be updated. The name must identify a table or view described in the catalog, but not a catalog table, a view of a catalog table (unless it is one of the updatable SYSSTAT views), a summary table or a read-only view. (For an explanation of read-only views, see "CREATE VIEW" on page 582. For an explanation of updatable catalog views, see Appendix D, "Catalog Views" on page 773.)

If *table-name* is a typed table, rows of the table or any of its proper subtables may get updated by the statement. Only the columns of the specified table may be set or referenced in the WHERE clause. For a positioned UPDATE, the associated cursor must also have specified the same table or view in the FROM clause without using ONLY.

## UPDATE

### **ONLY** (*table-name*)

Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be updated by the statement. For a positioned UPDATE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

### **ONLY** (*view-name*)

Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be updated by the statement. For a positioned UPDATE, the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

**AS** Optional keyword to introduce the *correlation-name*.

### *correlation-name*

May be used within search-condition to designate the table or view. (For an explanation of correlation-name, see “Correlation Names” on page 99.)

### **SET**

Introduces the assignment of values to column names.

### *assignment-clause*

### *column-name*

Identifies a column to be updated. The column-name must identify an updatable column of the specified table or view.<sup>86</sup> The object ID column of a typed table is not updatable (SQLSTATE 428DZ). A column must not be specified more than once (SQLSTATE 42701).

For a Positioned UPDATE:

- If the UPDATE clause was specified in the select-statement of the cursor, each column name in the assignment-clause must also appear in the UPDATE clause.
- If the UPDATE clause was not specified in the select-statement of the cursor and LANGLEVEL MIA or SQL92E was specified when the application was precompiled, the name of any updatable column may be specified.
- If the UPDATE clause was not specified in the select-statement of the cursor and LANGLEVEL SAA1 was specified either explicitly or by default when the application was precompiled, no columns may be updated.

---

<sup>86</sup> A column of a partitioning key is not updatable (SQLSTATE 42997). The row of data must be deleted and inserted to change columns in a partitioning key.

### *expression*

Indicates the new value of the column. The expression is any expression of the type described in “Expressions” on page 117. The expression can not include a column function except when it occurs within a scalar fullselect (SQLSTATE 42903).

An *expression* may contain references to columns of the target table of the UPDATE statement. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated.

### **NULL**

Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502).

### **DEFAULT**

Specifies that the default value should be used based on how the corresponding column is defined in the table. The value that is inserted depends on how the column was defined.

- If the column was defined using the WITH DEFAULT clause, then the value is set to the default defined for the column (see default-clause in “ALTER TABLE” on page 380).
- If the column was defined without specifying the WITH DEFAULT clause or the NOT NULL clause, then the value inserted is NULL.
- If the column was defined using the NOT NULL clause and the WITH DEFAULT clause was not used or DEFAULT NULL was used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

### *row-fullselect*

A fullselect that returns a single row with the number of columns corresponding to the number of column-names specified for assignment. The values are assigned to each corresponding column-name. If the result of the row-fullselect is no rows, then null values are assigned.

A *row-fullselect* may contain references to columns of the target table of the UPDATE statement. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated. An error is returned if there is more than one row in the result (SQLSTATE 21000).

### **WHERE**

Introduces a condition that indicates what rows are updated. You can omit the clause, give a search condition, or name a cursor. If the clause is omitted, all rows of the table or view are updated.

### *search-condition*

Is any search condition as described in Chapter 3, “Language Elements” on page 45. Each column-name in the search condition, other than in a subquery, must name a column of the table or view. When the search condition includes

## UPDATE

a subquery in which the same table is the base object of both the UPDATE and the subquery, the subquery is completely evaluated before any rows are 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 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 only once, whereas a subquery with a correlated reference may have to 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 595. The DECLARE CURSOR statement must precede the UPDATE statement in the program.

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 595.)

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

## Rules

- **Assignment:** Update values are assigned to columns under the assignment rules described in Chapter 3.
- **Validity:** 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.

If a view is used that is not defined using WITH CHECK OPTION, rows can be changed 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.

If a view is used that is defined using WITH CHECK OPTION, an updated row must conform to the definition of the view. For an explanation of the rules governing this situation, see “CREATE VIEW” on page 582.

- **Check Constraint:** Update value must satisfy the check-conditions of the check constraints defined on the table.

An UPDATE to a table with check constraints defined has the constraint conditions for each column updated evaluated once for each row that is updated. When processing an UPDATE statement, only the check constraints referring to the updated columns are checked.

- **Referential Integrity:** The value of the parent unique keys cannot be changed if the update rule is RESTRICT and there are one or more dependent rows. However, if the update rule is NO ACTION, parent unique keys can be updated as long as every child has a parent key by the time the update statement completes.

## UPDATE

A non-null update value of a foreign key must be equal to a value of the primary key of the parent table of the relationship.

### Notes

- If an update value violates any constraints, or if any other error occurs during the execution of the UPDATE statement, no rows are updated. The order in which multiple rows are updated is undefined.
- When an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows updated. The SQLERRD(5) field contains the number of rows inserted, deleted, or updated by all activated triggers. (For a description of the SQLCA, see Appendix B, “SQL Communication Area (SQLCA)” on page 759.)
- Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful UPDATE statement. Until the locks are released, the updated row can only be accessed by the application process that performed the update (except for applications using the Uncommitted Read isolation level). For further information on locking, see the descriptions of the COMMIT, ROLLBACK, and LOCK TABLE statements.

- If the URL value of a DATALINK column is updated, this is the same as deleting the old DATALINK value then inserting the new one. First, if the old value was linked to a file, that file is unlinked. Then, unless the linkage attributes of the DATALINK value are empty, the specified file is linked to that column.

The comment value of a DATALINK column can be updated without relinking the file by specifying an empty string as the URL path (for example, as the *data-location* argument of the DLVALUE scalar function or by specifying the new value to be the same as the old value).

If a DATALINK column is updated with a null, it is the same as deleting the existing DATALINK value.

An error may occur when attempting to update a DATALINK value if the file server of either the existing value or the new value is no longer registered with the database server (SQLSTATE 55022).

- When updating the column distribution statistics for a typed table, the subtable that first introduced the column must be specified.

### Examples

- *Example 1:* Change the job (JOB) of employee number (EMPNO) '000290' in the EMPLOYEE table to 'LABORER'.

```
UPDATE EMPLOYEE
SET JOB = 'LABORER'
WHERE EMPNO = '000290'
```

- *Example 2:* Increase the project staffing (PRSTAFF) by 1.5 for all projects that department (DEPTNO) 'D21' is responsible for in the PROJECT table.

## UPDATE

```
UPDATE PROJECT
SET PRSTAFF = PRSTAFF + 1.5
WHERE DEPTNO = 'D21'
```

- *Example 3:* All the employees except the manager of department (WORKDEPT) 'E21' have been temporarily reassigned. Indicate this by changing their job (JOB) to NULL and their pay (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.

```
UPDATE EMPLOYEE
SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

This statement could also be written as follows.

```
UPDATE EMPLOYEE
SET (JOB, SALARY, BONUS, COMM) = (NULL, 0, 0, 0)
WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

- *Example 4:* Update the salary and the commission column of the employee with employee number 000120 to the average of the salary and of the commission of the employees of the updated row's department respectively.

```
UPDATE EMPLOYEE EU
SET (EU.SALARY, EU.COMM)
=
(SELECT AVG(ES.SALARY), AVG(ES.COMM)
FROM EMPLOYEE ES
WHERE ES.WORKDEPT = EU.WORKDEPT)
WHERE EU.EMPNO = '000120'
```

- *Example 5:* In a C program display the rows from the EMPLOYEE table and then, if requested to do so, change the job (JOB) of certain employees to the new job keyed in.

```
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT *
        FROM EMPLOYEE
        FOR UPDATE OF JOB;

EXEC SQL OPEN C1;

EXEC SQL FETCH C1 INTO ... ;

if ( strcmp (change, "YES") == 0 )
    EXEC SQL UPDATE EMPLOYEE
        SET JOB = :newjob
        WHERE CURRENT OF C1;

EXEC SQL CLOSE C1;
```



---

**VALUES**

The `VALUES` statement is a form of query. It can be embedded in an application program or issued interactively. For detailed information, see “fullselect” on page 350.

## VALUES INTO

---

### VALUES INTO

The VALUES INTO statement produces a result table consisting of at most one row and assigns the values in that row to host variables.

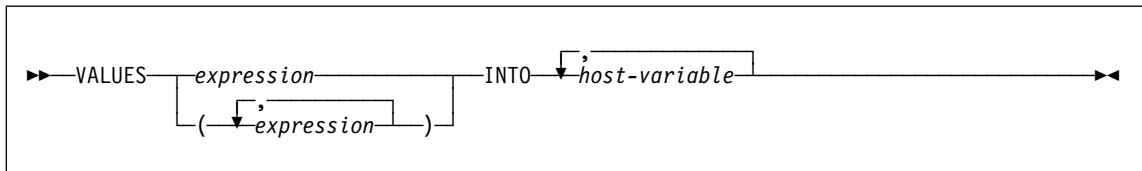
#### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared. The statement is not supported in REXX.

#### Authorization

None required.

#### Syntax



#### Description

##### VALUES

Introduces a single row consisting of one or more columns.

*expression*

An expression that defines a single value of a one column result table.

*(expression,...)*

One or more expressions that define the values for one or more columns of the result table.

##### INTO

Introduces a list of host variables.

*host-variable*

Identifies a variable that is described in the program under the rules for declaring 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 Appendix B, "SQL Communication Area (SQLCA)" on page 759.)

Each assignment to a variable is made according to the rules described in "Assignments and Comparisons" on page 70. Assignments are made in sequence through the list.

If an error occurs, no value is assigned to any host variable.

### Examples

*Example 1:* This C example retrieves the value of the CURRENT PATH special register into a host variable.

```
EXEC SQL VALUES(CURRENT PATH)  
       INTO :hv1;
```

*Example 2:* This C example retrieves a portion of a LOB field into a host variable, exploiting the LOB locator for deferred retrieval.

```
EXEC SQL VALUES (substr(:locator1,35))  
       INTO :details;
```

## WHENEVER

---

### WHENEVER

The **WHENEVER** statement specifies the action to be taken when a specified exception condition occurs.

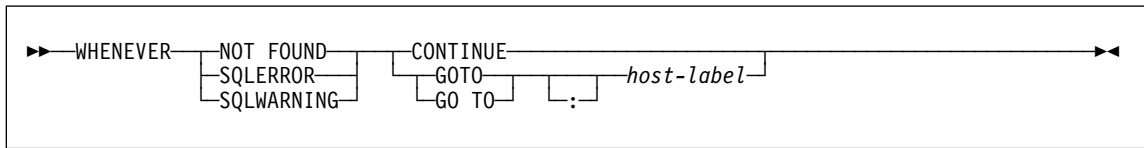
#### Invocation

This statement can only be embedded in an application program. It is not an executable statement. The statement is not supported in REXX.

#### Authorization

None required.

#### Syntax



#### Description

The **NOT FOUND**, **SQLERROR**, or **SQLWARNING** clause is used to identify the type of exception condition.

##### **NOT FOUND**

Identifies any condition that results in an **SQLCODE** of +100 or an **SQLSTATE** 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 **SQL** return code other than +100.

The **CONTINUE** or **GO TO** clause is used to specify what is to happen when the identified type of exception condition exists.

##### **CONTINUE**

Causes the next sequential instruction of the source program to be executed.

##### **GOTO** or **GO TO** *host-label*

Causes control to pass to 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.

## Notes

There are three types of **WHENEVER** statements:

```
WHENEVER NOT FOUND  
WHENEVER SQLERROR  
WHENEVER SQLWARNING
```

Every executable SQL statement in a 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 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.

## Example

In the following C example, if an error is produced, go to **HANDLERR**. If a warning code is produced, continue with the normal flow of the program. If no data is returned, go to **ENDDATA**.

```
EXEC SQL WHENEVER SQLERROR GOTO HANDLERR;  
EXEC SQL WHENEVER SQLWARNING CONTINUE;  
EXEC SQL WHENEVER NOT FOUND GO TO ENDDATA;
```

**WHENEVER**

## Appendix A. SQL Limits

The following tables describe certain SQL limits. Adhering to the most restrictive case can help the programmer design application programs that are easily portable.

Table 23. Identifier Length Limits

	Description	Limit in Bytes
1	Longest authorization name (can only be single-byte characters)	8
2	Longest constraint name	18
3	Longest correlation name	18
4	Longest cursor name	18
5	Longest external program name	8
6	Longest host identifier <sup>a</sup>	255
7	Longest schema name	8
8	Longest server (database alias) name	8
9	Longest statement name	18
10	Longest unqualified column name	18
11	Longest unqualified package name	8
12	Longest unqualified user-defined type, user-defined function, stored procedure, buffer pool, table space, nodegroup, table, view, alias, trigger or index name.	18

**Notes:**

- a** Individual host language compilers may have a more restrictive limit on variable names.

Table 24 (Page 1 of 2). Numeric Limits

	Description	Limit
1	Smallest INTEGER value	-2 147 483 648
2	Largest INTEGER value	+2 147 483 647
3	Smallest BIGINT value	-9 223 372 036 854 775 808
4	Largest BIGINT value	+9 223 372 036 854 775 807
5	Smallest SMALLINT value	-32 768
6	Largest SMALLINT value	+32 767
7	Largest decimal precision	31
8	Smallest DOUBLE value	-1.79769E+308
9	Largest DOUBLE value	+1.79769E+308
10	Smallest positive DOUBLE value	+2.225E-307
11	Largest negative DOUBLE value	-2.225E-307

## SQL Limits

Table 24 (Page 2 of 2). Numeric Limits

	Description	Limit
12	Smallest REAL value	-3.402E+38
13	Largest REAL value	+3.402E+38
14	Smallest positive REAL value	+1.175E-37
15	Largest negative REAL value	-1.175E-37

Table 25. String Limits

	Description	Limit
1	Maximum length of CHAR (in bytes)	254
2	Maximum length of VARCHAR (in bytes)	4 000
3	Maximum length of LONG VARCHAR (in bytes)	32 700
4	Maximum length of CLOB (in bytes)	2 147 483 647
5	Maximum length of GRAPHIC (in characters)	127
6	Maximum length of VARGRAPHIC (in characters)	2 000
7	Maximum length of LONG VARGRAPHIC (in characters)	16 350
8	Maximum length of DBCLOB (in characters)	1 073 741 823
9	Maximum length of BLOB (in bytes)	2 147 483 647
10	Maximum length of character constant	4 000
11	Maximum length of graphic constant	2 000
12	Maximum length of concatenated character string	2 147 483 647
13	Maximum length of concatenated graphic string	1 073 741 823
14	Maximum length of concatenated binary string	2 147 483 647
15	Maximum number of hex constant digits	4 000
16	Maximum size of a catalog comment (in bytes)	254

Table 26. Datetime Limits

	Description	Limit
1	Smallest DATE value	0001-01-01
2	Largest DATE value	9999-12-31
3	Smallest TIME value	00:00:00
4	Largest TIME value	24:00:00
5	Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
6	Largest TIMESTAMP value	9999-12-31-24.00.00.000000



## SQL Limits

Table 27 (Page 1 of 3). Database Manager Limits

	Description	Limit
1	Most columns in a table <sup>g</sup>	1 012
2	Most columns in a view <sup>a</sup>	5 000
3	Maximum length of a row including all overhead <sup>b</sup> g	8 101
4	Maximum size of a table per partition (in gigabytes) <sup>c</sup> g	128
5	Maximum size of an index per partition (in gigabytes)	64
6	Most rows in a table per partition	4 x 10 <sup>9</sup>
7	Longest index key including all overhead (in bytes)	255
8	Most columns in an index key	16
9	Most indexes on a table	32 767 or storage
10	Most tables referenced in an SQL statement or a view	storage
11	Most host variable declarations in a precompiled program <sup>c</sup>	storage
12	Most host variable references in an SQL statement	32 767
13	Longest host variable value used for insert or update (in bytes)	2 147 483 647
14	Longest SQL statement (in bytes)	32 765
15	Most elements in a select list <sup>g</sup>	1 012
16	Most predicates in a WHERE or HAVING clause	storage
17	Maximum number of columns in a GROUP BY clause <sup>g</sup>	1 012
18	Maximum total length of columns in a GROUP BY clause (in bytes) <sup>g</sup>	8 101
19	Maximum number of columns in an ORDER BY clause <sup>g</sup>	1 012
20	Maximum total length of columns in an ORDER BY clause (in bytes) <sup>g</sup>	8 101
21	Maximum size of an SQLDA (in bytes)	storage
22	Maximum number of prepared statements	storage
23	Most declared cursors in a program	storage
24	Maximum number of cursors opened at one time	storage
25	Most tables in an SMS table space	65 534
26	Maximum number of constraints on a table	storage
27	Maximum level of subquery nesting	storage

## SQL Limits

Table 27 (Page 2 of 3). Database Manager Limits

	Description	Limit
28	Maximum number of subqueries in a single statement	storage
29	Most values in an INSERT statement <sup>g</sup>	1 012
30	Most SET clauses in a single UPDATE statement <sup>g</sup>	1 012
31	Most columns in a UNIQUE constraint (supported via a UNIQUE index)	16
32	Maximum combined length of columns in a UNIQUE constraint (supported via a UNIQUE index) (in bytes)	255
33	Most referencing columns in a foreign key	16
34	Maximum combined length of referencing columns in a foreign key (in bytes)	255
35	Maximum length of a check constraint specification (in bytes)	32765
36	Maximum number of columns in a partitioning key <sup>e</sup>	500
37	Maximum number of rows changed in a unit of work	storage
38	Maximum number of packages	storage
39	Most constants in a statement	storage
40	Maximum concurrent users of server <sup>d</sup>	64 000
41	Maximum number of parameters in a stored procedure	32 767
42	Maximum number of parameters in a user defined function	90
43	Maximum run-time depth of cascading triggers	16
44	Maximum number of simultaneously active event monitors	32
45	Maximum size of a regular DMS table space (in gigabytes) <sup>c g</sup>	128
46	Maximum size of a long DMS table space (in terabytes) <sup>c</sup>	2
47	Maximum size of a temporary DMS table space (in terabytes) <sup>c</sup>	2
48	Maximum number of databases per instance concurrently in use	256
49	Maximum number of concurrent users per instance	64 000
50	Maximum number of concurrent applications per database	1 000

Table 27 (Page 3 of 3). Database Manager Limits

	Description	Limit
51	Maximum depth of cascaded triggers	16
52	Maximum partition number	999
53	Most table objects in DMS table space <sup>f g</sup>	13 305

**Notes:**

- a** This maximum can be achieved using a join in the CREATE VIEW statement. Selecting from such a view is subject to the limit of most elements in a select list.
- b** The actual data for BLOB, CLOB, LONG VARCHAR, DBCLOB, and LONG VARGRAPHIC columns is not included in this count. However information about the location of that data does take up some space in the row.
- c** The numbers shown are architectural limits and approximations. The practical limits may be less.
- d** The actual value will be the value of the MAXAGENTS configuration parameter. See the *Administration Guide* for information on MAXAGENTS.
- e** This is an architectural limit. The limit on the most columns in an index key should be used as a practical limit.
- f** Table objects include data, indexes, LONG VARCHAR/VARGRAPHIC columns, and LOB columns. Table objects that are in the same table space as the table data do not count extra toward the limit. However, each table object that is in a different table space than the table data does contribute one toward the limit for each table object type per table in the table space in which the table object resides.
- g** For page size specific values please refer to Table 28 on page 757.

Table 28 (Page 1 of 2). Database Manager Page Size Specific Limits

	Description	4K page size limit	8K page size limit
1	Most columns in a table	500	1 012
3	Maximum length of a row including all overhead <sup>b</sup>	4 005	8 101
4	Maximum size of a table per partition (in gigabytes) <sup>c</sup>	64	128
15	Most elements in a select list	500	1 012
17	Maximum number of columns in a GROUP BY clause	500	1 012
18	Maximum total length of columns in a GROUP BY clause (in bytes)	4 005	8 101
19	Maximum number of columns in an ORDER BY clause	500	1 012
20	Maximum total length of columns in an ORDER BY clause (in bytes)	4 005	8 101
29	Most values in an INSERT statement	500	1 012
30	Most SET clauses in a single UPDATE statement	500	1 012

## SQL Limits

Table 28 (Page 2 of 2). Database Manager Page Size Specific Limits

	Description	4K page size limit	8K page size limit
45	Maximum size of a regular DMS table space (in gigabytes) <sup>c</sup>	64	128
53	Most table objects in DMS table space <sup>f</sup>	6 648	13 305

## Appendix B. SQL Communication Area (SQLCA)

An SQLCA is a collection of variables that is updated at the end of the execution of every SQL statement. A program that contains executable SQL statements (except for DECLARE, INCLUDE, and WHENEVER) and is precompiled with option LANGLEVEL SAA1 (the default) or MIA must provide exactly one SQLCA, though more than one SQLCA is possible by having one SQLCA per thread in a multi-threaded application.

When a program is precompiled with option LANGLEVEL SQL92E, an SQLCODE or SQLSTATE variable may be declared in the SQL declare section or an SQLCODE variable can be declared somewhere in the program.

An SQLCA should not be provided when using LANGLEVEL SQL92E (see the *Road Map to DB2 Programming* for information on declaring SQLSTATE or SQLCODE variables in specific programming languages). The SQL INCLUDE statement can be used to provide the declaration of the SQLCA in all languages but REXX. The SQLCA is automatically provided in REXX (see the *Road Map to DB2 Programming* for information on declaring the SQLCA in REXX).

### Viewing the SQLCA Interactively

To display the SQLCA after each command you use in the command line processor, use the command **db2 -a**. The SQLCA is then provided as part of the output for subsequent commands. The SQLCA is also dumped in the db2diag.log file.

### SQLCA Field Descriptions

Table 29 (Page 1 of 4). Fields of SQLCA

Name <sup>87</sup>	Data Type	Field values								
sqlcaid	CHAR(8)	An "eye catcher" for storage dumps containing 'SQLCA'.								
sqlcab	INTEGER	Contains the length of the SQLCA, 136.								
sqlcode	INTEGER	Contains the SQL return code. For specific meanings of SQL return codes, see the message section of the <i>Messages Reference</i> . <table border="1" data-bbox="628 1397 1290 1534"> <thead> <tr> <th>Code</th> <th>Means</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Successful execution (although one or more SQLWARN indicators may be set).</td> </tr> <tr> <td>positive</td> <td>Successful execution, but with a warning condition.</td> </tr> <tr> <td>negative</td> <td>Error condition.</td> </tr> </tbody> </table>	Code	Means	0	Successful execution (although one or more SQLWARN indicators may be set).	positive	Successful execution, but with a warning condition.	negative	Error condition.
Code	Means									
0	Successful execution (although one or more SQLWARN indicators may be set).									
positive	Successful execution, but with a warning condition.									
negative	Error condition.									
sqlerrml	SMALLINT	Length indicator for <i>sqlerrmc</i> , in the range 0 through 70. 0 means that the value of <i>sqlerrmc</i> is not relevant.								

<sup>87</sup> The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

## SQLCA

Table 29 (Page 2 of 4). Fields of SQLCA

Name <sup>87</sup>	Data Type	Field values
sqlerrmc	VARCHAR (70)	<p>Contains one or more tokens, separated by X'FF', that are substituted for variables in the descriptions of error conditions.</p> <p>This field is also used when a successful connection is completed.</p> <p>When a NOT ATOMIC compound SQL statement is issued, it may contain information on up to 7 errors.</p> <p>For specific meanings of SQL return codes, see the message section of the <i>Messages Reference</i>.</p>
sqlerrp	CHAR(8)	<p>Begins with a three-letter identifier indicating the product, followed by five digits indicating the version, release, and modification level of the product. For example, SQL05000 means DB2 Universal Database versions for Version 5 Release 0 Modification level 0.</p> <p>If SQLCODE indicates an error condition, then this field identifies the module that returned the error.</p> <p>This field is also used when a successful connection is completed.</p>
sqlerrd	ARRAY	Six INTEGER variables that provide diagnostic information. These values are generally empty if there are no errors, except for sqlerrd(6) from a partitioned database.
sqlerrd(1)	INTEGER	If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the database code page from the application code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. <sup>a</sup>
sqlerrd(2)	INTEGER	If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. <sup>a</sup> If the SQLCA results from a NOT ATOMIC compound SQL statement that encountered one or more errors, the value is set to the number of statements that failed.
sqlerrd(3)	INTEGER	If PREPARE is invoked and successful, contains an estimate of the number of rows that will be returned. After INSERT, UPDATE, and DELETE, contains the actual number of rows affected. If compound SQL is invoked, contains an accumulation of all sub-statement rows. If CONNECT is invoked, contains 1 if the database can be updated; 2 if the database is read only.
sqlerrd(4)	INTEGER	If PREPARE is invoked and successful, contains a relative cost estimate of the resources required to process the statement. If compound SQL is invoked, contains a count of the number of successful sub-statements. If CONNECT is invoked, contains 0 for a one-phase commit from a down-level client; 1 for a one-phase commit; 2 for a one-phase, read-only commit; and 3 for a two-phase commit.

Table 29 (Page 3 of 4). Fields of SQLCA

Name <sup>87</sup>	Data Type	Field values
sqlerrd(5)	INTEGER	<p>Contains the total number of rows deleted, inserted, or updated as a result of both:</p> <ul style="list-style-type: none"> <li>• The enforcement of constraints after a successful delete operation</li> <li>• The processing of triggered SQL statements from activated triggers.</li> </ul> <p>If compound SQL is invoked, contains an accumulation of the number of such rows for all substatements. In some cases when an error is encountered, this field contains a negative value that is an internal error pointer. If CONNECT is invoked, contains an authentication type value of 0 for a server authentication; 1 for client authentication; 2 for authentication using DB2 Connect; 3 for DCE security services authentication; 255 for unspecified authentication.</p>
sqlerrd(6)	INTEGER	<p>For a partitioned database, contains the partition number of the partition that encountered the error or warning. If no errors or warnings were encountered, this field contains the partition number of the coordinator node. The number in this field is the same as that specified for the partition in the db2nodes.cfg file.</p>
sqlwarn	Array	<p>A set of warning indicators, each containing a blank or W. If compound SQL is invoked, contains an accumulation of the warning indicators set for all substatements.</p>
sqlwarn0	CHAR(1)	<p>Blank if all other indicators are blank; contains W if at least one other indicator is not blank.</p>
sqlwarn1	CHAR(1)	<p>Contains W if the value of a string column was truncated when assigned to a host variable. Contains N if the null terminator was truncated.</p>
sqlwarn2	CHAR(1)	<p>Contains W if null values were eliminated from the argument of a function. <sup>b</sup></p>
sqlwarn3	CHAR(1)	<p>Contains W if the number of columns is not equal to the number of host variables.</p>
sqlwarn4	CHAR(1)	<p>Contains W if a prepared UPDATE or DELETE statement does not include a WHERE clause.</p>
sqlwarn5	CHAR(1)	<p>Reserved for future use.</p>
sqlwarn6	CHAR(1)	<p>Contains W if the result of a date calculation was adjusted to avoid an impossible date.</p>
sqlwarn7	CHAR(1)	<p>Reserved for future use.</p>
sqlwarn8	CHAR(1)	<p>Contains W if a character that could not be converted was replaced with a substitution character.</p>
sqlwarn9	CHAR(1)	<p>Contains W if arithmetic expressions with errors were ignored during column function processing.</p>
sqlwarn10	CHAR(1)	<p>Contains W if there was a conversion error when converting a character data value in one of the fields in the SQLCA.</p>
sqlstate	CHAR(5)	<p>A return code that indicates the outcome of the most recently executed SQL statement.</p>

## SQLCA

Table 29 (Page 4 of 4). Fields of SQLCA

Name <sup>87</sup>	Data Type	Field values
<b>Note:</b>		
a		See the “Character Conversion Expansion Factor” section of the “Programming in Complex Environments” chapter in the <i>Embedded SQL Programming Guide</i> for details.
b		Some functions may not set SQLWARN2 to W even though null values were eliminated because the result was not dependent on the elimination of null values.

---

### Order of Error Reporting

The order of error reporting is as follows:

1. Severe error conditions are always reported. When a severe error is reported, there are no additions to the SQLCA.
2. If no severe error occurs, a deadlock error takes precedence over other errors.
3. For all other errors, the SQLCA for the first negative SQL code is returned.
4. If no negative SQL codes are detected, the SQLCA for the first warning (that is, positive SQL code) is returned.

For DB2 Extended Enterprise Edition, the exception to this rule occurs if a data manipulation operation is issued on a table that is empty on one partition, but has data on other nodes. The SQLCODE +100 is only returned to the application if agents from all partitions return SQL0100W, either because the table is empty on all partitions or there are no rows that satisfy the WHERE clause in an UPDATE statement.

---

### DB2 Extended Enterprise Edition Usage of the SQLCA

In DB2 Universal Database Extended Enterprise Edition, one SQL statement may be executed by a number of agents on different partitions, and each agent may return a different SQLCA for different errors or warnings. The coordinator agent also has its own SQLCA.

To provide a consistent view for applications, all SQLCA values are merged into one structure and SQLCA fields indicate global counts. For example:

- For all errors and warnings, the *sqlwarn* field contains the warning flags received from all agents.
- Values in the *sqlerrd* fields indicating row counts are accumulations from all agents.

Note that SQLSTATE 09000 may not be returned in all cases of an error occurring while processing a triggered SQL statement.



---

## Appendix C. Appendix C. SQL Descriptor Area (SQLDA)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement. The SQLDA variables are options that can be used by the PREPARE, OPEN, FETCH, EXECUTE, and CALL statements. An SQLDA communicates with dynamic SQL; it can be used in a DESCRIBE statement, modified with the addresses of host variables, and then reused in a FETCH statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C, REXX, FORTRAN, and COBOL. In REXX, the SQLDA is somewhat different than in the other languages; for information on the use of SQLDAs in REXX see the *Embedded SQL Programming Guide*

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In OPEN, EXECUTE, FETCH, and CALL, an SQLDA describes host variables.

In DESCRIBE and PREPARE, if any one of the columns being described is either a LOB type,<sup>88</sup> or a distinct type, the number of SQLVAR entries for the entire SQLDA will be doubled. For example:

When describing a table with 3 VARCHAR columns and 1 INTEGER column, there will be 4 SQLVAR entries

When describing a table with 2 VARCHAR columns, 1 CLOB column, and 1 integer column, there will be 8 SQLVAR entries

In EXECUTE, FETCH, OPEN, and CALL, if any one of the variables being described is a LOB type,<sup>88</sup> the number of SQLVAR entries for the entire SQLDA needs to be doubled.<sup>89</sup>

---

### Field Descriptions

An SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR. In OPEN, FETCH, EXECUTE, and CALL each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, each occurrence of SQLVAR describes a column of a result table. There are two types of SQLVAR entries:

1. **Base SQLVARs:** These entries are always present. They contain the base information about the column or host variable such as data type code, length attribute, column name, host variable address, and indicator variable address.
2. **Secondary SQLVARs:** These entries are only present if the number of SQLVAR entries is doubled as per the rules outlined above. For distinct types they contain

---

<sup>88</sup> LOB locators and file reference variables do not require doubled SQLDAs.

<sup>89</sup> Distinct types are not relevant in these cases since a host variable cannot be a distinct type.

## SQLDA

the distinct type name. For LOBs, they contain the length attribute of the host variable and a pointer to the buffer that contains the actual length.<sup>90</sup> If locators or file reference variables are used to represent LOBs, these entries are not necessary.

In SQLDAs that contain both types of entries, the base SQLVARs are in a block before the block of secondary SQLVARs. In each, the number of entries is equal to value in SQLD (even though many of the secondary SQLVAR entries may be unused).

The circumstances under which the SQLVAR entries are set by DESCRIBE is detailed in “Effect of DESCRIBE on the SQLDA” on page 767.

### Fields in the SQLDA Header

Table 30. Fields in the SQLDA Header

C Name	SQL Data Type	Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, EXECUTE, and CALL (set by the application prior to executing the statement)
sqldaaid	CHAR(8)	The seventh byte of this field is a flag byte named SQLDOUBLED. The database manager sets SQLDOUBLED to the character '2' if two SQLVAR entries have been created for each column; otherwise it is set to a blank (X'20' in ASCII, X'40' in EBCDIC). See “Effect of DESCRIBE on the SQLDA” on page 767 for details on when SQLDOUBLED is set.	The seventh byte of this field is used when the number of SQLVARs is doubled. It is named SQLDOUBLED. If any of the host variables being described is a BLOB, CLOB, or DBCLOB, the seventh byte must be set to the character '2'; otherwise it can be set to any character but we advise the use of a blank.  When used with the CALL statement and one or more SQLVARs define of data field as FOR BIT DATA, the sixth byte must be set to the '+' character; otherwise it can be set to any character but we advise the use of blank.
sqldabc	INTEGER	Length of the SQLDA, equal to SQLN*44+16.	Length of the SQLDA, >= to SQLN*44+16.
sqln	SMALLINT	Unchanged by the database manager. Must be set to a value greater than or equal to zero before the DESCRIBE statement is executed. Indicates the total number of occurrences of SQLVAR.	Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero.
sqld	SMALLINT	Set by the database manager to the number of columns in the result table (or to zero if the statement being described is not a select-statement).	The number of host variables described by occurrences of SQLVAR.

<sup>90</sup> The distinct type and LOB information does not overlap, so distinct types can be based on LOBs without forcing the number of SQLVAR entries on a DESCRIBE to be tripled.

## Fields in an Occurrence of a Base SQLVAR

Table 31 (Page 1 of 2). Fields in a Base SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, EXECUTE, and CALL
sqltype	SMALLINT	<p>Indicates the data type of the column and whether it can contain nulls. Table 33 on page 769 lists the allowable values and their meanings.</p> <p>Note that for a distinct type, the data type of the base type is placed into this field. There is no indication in the Base SQLVAR that it is part of the description of a distinct type.</p>	<p>Same for host variable. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string.</p>
sqlen	SMALLINT	<p>The length attribute of the column. For datetime columns, the length of the string representation of the values. See Table 33 on page 769.</p> <p>Note that the value is set to 0 for large object strings (even for those whose length attribute is small enough to fit into a two byte integer).</p>	<p>The length attribute of the host variable. See Table 33 on page 769.</p> <p>Note that the value is ignored by the database manager for CLOB, DBCLOB, and BLOB columns. The len.sqllonglen field in the Secondary SQLVAR is used instead.</p>
sqldata	pointer	<p>For character-string SQLVARs, sqldata contains 0 if the column is defined with the FOR BIT DATA attribute. If the column does not have the FOR BIT DATA attribute, the value depends on the encoding of the data. For single-byte SBCS encoded data, sqldata contains the SBCS code page. For mixed DBCS encoded data, sqldata contains the SBCS code page associated with the composite DBCS code page. For Japanese or Traditional-Chinese EUC encoded data, sqldata contains the composite EUC code page.</p> <p>For all other column types, sqldata is undefined.</p>	<p>Contains the address of the host variable (where the fetched data will be stored).</p>
sqlind	pointer	<p>For character-string SQLVARs, sqlind contains 0 except for mixed DBCS encoded data when sqlind contains the DBCS code page associated with the composite DBCS code page.</p> <p>For all other column types, sqlind is undefined.</p>	<p>Contains the address of an associated indicator variable, if there is one; otherwise, not used.</p>

## SQLDA

Table 31 (Page 2 of 2). Fields in a Base SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, EXECUTE, and CALL
sqlname	VARCHAR (30)	<p>Contains the unqualified name of the column.</p> <p>For columns that have a system generated name (the result column was not directly derived from a single column and did not specify a name using the AS clause), the thirtieth byte is set to X'FF'. For column names specified by the AS clause, this byte is X'00'.</p>	<p>When used with the CALL statement to access a DRDA application server, sqlname can be set to indicate a FOR BIT DATA string as follows:</p> <ul style="list-style-type: none"> <li>the length of sqlname is 8</li> <li>the first four bytes of sqlname are X'00000000'</li> <li>the remaining four bytes of sqlname are reserved (and currently ignored).</li> </ul> <p>In addition, the sqltype must indicate a CHAR, VARCHAR or LONG VARCHAR and the sixth byte of the sqlda field is set to the '+' character.</p> <p>This technique can also be used with OPEN and EXECUTE when using DB2 Connect to access the server.</p>

## Fields in an Occurrence of a Secondary SQLVAR

Table 32 (Page 1 of 2). Fields in a Secondary SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, EXECUTE, and CALL
len.sqlonglen	INTEGER	The length attribute of a BLOB, CLOB, or DBCLOB column.	The length attribute of a BLOB, CLOB, or DBCLOB host variable. The database manager ignores the SQLLEN field in the Base SQLVAR for the data types. The length attribute stores the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB.
reserve2	CHAR(3)	Not used.	Not used.
sqlflag4	CHAR(1)	The value is X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. Otherwise, the value is X'00'.	Set to X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. Otherwise, the value is X'00'.

Table 32 (Page 2 of 2). Fields in a Secondary SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, EXECUTE, and CALL
sqldatalen	pointer	Not used.	Used for BLOB, CLOB, and DBCLOB host variables only.  If this field is NULL, then the actual length (in characters) should be stored in the 4 bytes immediately before the start of the data and SQLDATA should point to the first byte of the field length.  If this field is not NULL, it contains a pointer to a 4 byte long buffer that contains the actual length <i>in bytes</i> (even for DBCLOB) of the data in the buffer pointed to from the SQLDATA field in the matching Base SQLVAR.  Note that, whether or not this field is used, the len.sqllonglen field must be set.
sqldatatype_name	VARCHAR(27)	For a distinct type column, the database manager sets this to the fully qualified distinct type name. <sup>1</sup> For a reference type, the database manager sets this to the fully qualified type name of the target type of the reference.	Not used.
reserved	CHAR(3)	Not used.	Not used.

**Note:**

1. The first 8 bytes contain the schema name of the type (extended to the right with spaces, if necessary). Byte 9 contains a dot (.). Bytes 10 to 27 contain the low order portion of the type name which is *not* extended to the right with spaces.

Note that, although the prime purpose of this field is for the name of user-defined types, the field is also set for IBM predefined data types. In this case, the schema name is SYSIBM and the low order portion of the name is the name stored in TYPENAME column of the DATATYPES catalog view. For example:

type name	length	sqldatatype_name
A.B	10	A .B
INTEGER	16	SYSIBM .INTEGER
"Frank's".SMINT	13	Frank's .SMINT
MY."type "	15	MY .type

**Effect of DESCRIBE on the SQLDA**

For a DESCRIBE or PREPARE INTO statement, the database manager always sets SQLD to the number of columns in the result set.

The SQLVARs in the SQLDA are set in the following cases:

- SQLN >= SQLD and no column is either a LOB or distinct type

## SQLDA

The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.

- SQLN  $\geq$  2\*SQLD and at least one column is a LOB or distinct type

Two times SQLD SQLVAR entries are set and SQLDOUBLED is set to '2'.

- SQLD  $\leq$  SQLN < 2\*SQLD and at least one column is a distinct type but there are no LOB columns

The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- SQLN < SQLD and no column is either a LOB or distinct type

No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.

Allocate SQLD SQLVARs for a successful DESCRIBE.

- SQLN < SQLD and at least one column is a distinct type but there are no LOB columns

No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.

Allocate 2\*SQLD SQLVARs for a successful DESCRIBE including the names of the distinct types.

- SQLN < 2\*SQLD and at least one column is a LOB

No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).

Allocate 2\*SQLD SQLVARs for a successful DESCRIBE.

References in the above lists to LOB columns include distinct type columns whose source type is a LOB type.

The SQLWARN option of the BIND or PREP command is used to control whether the DESCRIBE (or PREPARE INTO) will return the warning SQLCODEs +236, +237, +239. It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 is always returned when there are LOB columns in the select list and there are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB column in the result set.

---

## SQLTYPE and SQLLEN

Table 33 on page 769 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.

Table 33 (Page 1 of 2). SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, EXECUTE, and CALL

SQLTYPE	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
	Column Data Type	SQLLEN	Host Variable Data Type	SQLLEN
384/385	date	10	fixed-length character string representation of a date	length attribute of the host variable
388/389	time	8	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
396/397	DATALINK	length attribute of the column	DATALINK	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 *	BLOB	Not used. *
408/409	CLOB	0 *	CLOB	Not used. *
412/413	DBCLOB	0 *	DBCLOB	Not used. *
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	8 for double precision, 4 for single precision	floating point	8 for double precision, 4 for single precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2

## SQLDA

Table 33 (Page 2 of 2). *SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, EXECUTE, and CALL*

SQLTYPE	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, EXECUTE, and CALL	
	Column Data Type	SQLLEN	Host Variable Data Type	SQLLEN
492/493	big integer	8	big integer	8
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
804/805	Not applicable	Not applicable	BLOB file reference variable.	267
808/809	Not applicable	Not applicable	CLOB file reference variable.	267
812/813	Not applicable	Not applicable	DBCLOB file reference variable.	267
960/961	Not applicable	Not applicable	BLOB locator	4
964/965	Not applicable	Not applicable	CLOB locator	4
968/969	Not applicable	Not applicable	DBCLOB locator	4

**Note:**

\* The len.sqlonglen field in the secondary SQLVAR contains the length attribute of the column.

### Packed Decimal Numbers

Packed decimal numbers are stored in a variation of Binary Coded Decimal (BCD) notation. In BCD, each nybble (four bits) represents one decimal digit. For example, 0001 0111 1001 represents 179. Therefore, read a packed decimal value nybble by nybble. Store the value in bytes and then read those bytes in hexadecimal representation to return to decimal. For example, 0001 0111 1001 becomes 00000001 01111001 in binary representation. By reading this number as hexadecimal, it becomes 0179.

The decimal point is determined by the scale. In the case of a DEC(12,5) column, for example, the rightmost 5 digits are to the right of the decimal point.

Sign is indicated by a nybble to the right of the nybbles representing the digits. A positive or negative sign is indicated as follows:

Table 34. *Values for Sign Indicator of a Packed Decimal Number*

Sign	Representation		
	Binary	Decimal	Hexadecimal
Positive (+)	1100	12	C
Negative (-)	1101	13	D

In summary:



1. To store any value, allocate  $p/2+1$  bytes, where  $p$  is precision.
2. Assign the nybbles from left to right to represent the value. If a number has an even precision, a leading zero nybble is added. This assignment includes leading (insignificant) and trailing (significant) zero digits.
3. The sign nybble will be the second nybble of the last byte.

There is an alternative way to perform packed decimal conversions, see "CHAR" on page 195.

For example:

Column	Value	Nybbles in Hexadecimal Grouped by Bytes
DEC(8,3)	6574.23	00 65 74 23 0C
DEC(6,2)	-334.02	00 33 40 2D
DEC(7,5)	5.2323	05 23 23 0C
DEC(5,2)	-23.5	02 35 0D

## UNRECOGNIZED AND UNSUPPORTED SQLTYPES

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as at the receiver of the data. This is particularly important as new data types are added to the product.

New data types may or may not be supported by the sender or receiver of the data and may or may not even be recognized by the sender or receiver of the data. Depending on the situation, the new data type may be returned, or a compatible data type agreed upon by both the sender and receiver of the data may be returned or an error may result.

When the sender and receiver agree to use a compatible data type, the following indicates the mapping that will take place. This mapping will take place when at least one of the sender or the receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Data Type	Compatible Data Type
BIGINT	DECIMAL(19, 0)
ROWID	VARCHAR(40) FOR BIT DATA

Note that no indication is given in the SQLDA that the data type is substituted.

## SQLLEN Field for Decimal

The SQLLEN field contains the precision (first byte) and scale (second byte) of the decimal column. If writing a portable application, the precision and scale bytes should be set individually, versus setting them together as a short integer. This will avoid integer byte reversal problems.

For example, in C:

## SQLDA

```
((char *)&(sqlda->sqlvar[i].sqlen))[0] = precision;  
((char *)&(sqlda->sqlvar[i].sqlen))[1] = scale;
```

In REXX, the scale and precision fields are referenced as follows:

```
sqlen.scale  
sqlen.precision
```

## Appendix D. Catalog Views

The database manager creates and maintains two sets of system catalog views. This appendix contains a description of each system catalog view, including column names and data types. All the system catalog views are created when a database is created with the CREATE DATABASE command. The catalog views cannot be explicitly created or dropped. The system catalog views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the system catalog views is available through normal SQL query facilities. The system catalog views cannot be modified using normal SQL data manipulation commands with the exception of some specific updatable catalog views.

The catalog views are supported in addition to the catalog base tables from Version 1. The views are within the SYSCAT schema and SELECT privilege on all views is granted to PUBLIC by default. Application programs should be written to these views rather than the base catalog tables.<sup>91</sup> A second set of views formed from a subset of those within the SYSCAT schema, contain statistical information used by the optimizer. The views within the SYSSTAT schema contain some updatable columns.

The catalog views are designed to use more consistent conventions than the underlying catalog base tables. Columns have consistent names based on the type of objects that they describe:

Described Object	Column Names
Table	TABSCHEMA, TABNAME
Index	INDSCHEMA, INDNAME
View	VIEWSCHEMA, VIEWNAME
Constraint	CONSTSCHEMA, CONSTNAME
Trigger	TRIGSCHEMA, TRIGNAME
Package	PKGSCHEMA, PKGNAME
Type	TYPESCHEMA, TYPENAME, TYPEID
Function	FUNCSCHEMA, FUNCNAME, FUNCID
Column	COLNAME
Schema	SCHEMANAME
Table Space	TBSPACE
Nodegroup	NGNAME
Buffer pool	BPNAME
Event Monitor	EVMONNAME
Creation Timestamp	CREATE_TIME

Objects associated with typed tables (columns, indexes, ...) that are part of a hierarchy appear in the SYSCAT and SYSSTAT views only at the level they are introduced, not for every subtable in the hierarchy.

<sup>91</sup> Most existing applications using the base tables, however, will continue to run.

## Catalog Views

---

### Updatable Catalog Views

The updatable views contain statistical information used by the optimizer. Some columns in these views may be changed to investigate the performance of hypothetical databases. An object (table, column, function, or index) will appear in the updatable catalog view for a given user only if that user created the object, holds CONTROL privilege on the object, or holds explicit DBADM privilege. These views are found in the SYSSTAT schema. They are defined on top of the SYSCAT views.

Before changing any statistics for the first time, it is advised to issue the RUNSTATS command so that all statistics will reflect the current state.

---

### “Roadmap” to Catalog Views

Description	Catalog View	Page
authorities on database	SYSCAT.DBAUTH	786
Buffer pool configuration on nodegroup	SYSCAT.BUFFERPOOLS	776
Buffer pool size on node	SYSCAT.BUFFERPOOLNODES	777
check constraints	SYSCAT.CHECKS	778
column privileges	SYSCAT.COLAUTH	779
columns	SYSCAT.COLUMNS	782
columns referenced by check constraints	SYSCAT.COLCHECKS	780
columns used in keys	SYSCAT.KEYCOLUSE	797
constraint dependencies	SYSCAT.CONSTDEP	784
datatypes	SYSCAT.DATATYPES	785
event monitor definitions	SYSCAT.EVENTMONITORS	787
events currently monitored	SYSCAT.EVENTS	788
function parameters	SYSCAT.FUNCPARMS	789
index privileges	SYSCAT.INDEXAUTH	793
indexes	SYSCAT.INDEXES	794
detailed column statistics	SYSCAT.COLDIST	781
nodegroup definitions	SYSCAT.NODEGROUPS	799
nodegroup nodes	SYSCAT.NODEGROUPDEF	798
partitioning maps	SYSCAT.PARTITIONMAPS	805
package dependencies	SYSCAT.PACKAGEDEP	801
package privileges	SYSCAT.PACKAGEAUTH	800
packages	SYSCAT.PACKAGES	802
stored procedures	SYSCAT.PROCEDURES	806
procedure parameters	SYSCAT.PROCPARMS	807
referential constraints	SYSCAT.REFERENCES	808

## Catalog Views

Description	Catalog View	Page
schema privileges	SYSCAT.SCHEMAAUTH	809
schemas	SYSCAT.SCHEMATA	810
statements in packages	SYSCAT.STATEMENTS	811
table constraints	SYSCAT.TABCONST	814
table privileges	SYSCAT.TABAUTH	812
tables	SYSCAT.TABLES	815
table spaces	SYSCAT.TABLESPACES	818
trigger dependencies	SYSCAT.TRIGDEP	819
triggers	SYSCAT.TRIGGERS	820
user-defined functions	SYSCAT.FUNCTIONS	790
view dependencies	SYSCAT.VIEWDEP	821
views	SYSCAT.TABLES SYSCAT.VIEWS	815 822

---

### “Roadmap” to Updatable Catalog Views

Description	Catalog View	Page
columns	SYSSTAT.COLUMNNS	824
indexes	SYSSTAT.INDEXES	827
detailed column statistics	SYSSTAT.COLDIST	823
tables	SYSSTAT.TABLES	830
user-defined functions	SYSSTAT.FUNCTIONS	825

## SYSCAT.BUFFERPOOLS

---

### SYSCAT.BUFFERPOOLS

Contains a row for every buffer pool in every nodegroup.

*Table 35. SYSCAT.BUFFERPOOLS Catalog View*

Column Name	Data Type	Nullable	Description
BPNAME	VARCHAR(18)		Name of buffer pool
BUFFERPOOLID	INTEGER		Internal buffer pool identifier
NGNAME	VARCHAR(18)	Yes	Nodegroup name (NULL if the buffer pool exists on all nodes in the database)
NPAGES	INTEGER		Number of pages in the buffer pool
PAGESIZE	INTEGER		Pagesize for this buffer pool
ESTORE	CHAR(1)		N=This buffer pool does not use extended storage Y=This buffer pool uses extended storage

## SYSCAT.BUFFERPOOLNODES

---

### SYSCAT.BUFFERPOOLNODES

Contains a row for each node in the buffer pool for which the size of the buffer pool on the node is different from the default size in SYSCAT.BUFFERPOOLS column NPAGES.

*Table 36. SYSCAT.BUFFERPOOLNODES Catalog View*

Column Name	Data Type	Nullable	Description
BUFFERPOOLID	INTEGER		Internal buffer pool identifier
NODENUM	SMALLINT		Node Number
NPAGES	INTEGER		Number of pages in this buffer pool on this node

## SYSCAT.CHECKS

---

### SYSCAT.CHECKS

Contains one row for each CHECK constraint.

*Table 37. SYSCAT.CHECKS Catalog View*

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the check constraint (unique within a table.)
DEFINER	CHAR(8)		Authorization ID under which the check constraint was defined.
TABSCHEMA	CHAR(8)		Qualified name of the table to which this constraint applies.
TABNAME	VARCHAR(18)		
CREATE_TIME	TIMESTAMP		The time at which the constraint was defined. Used in resolving functions that are used in this constraint. No functions will be chosen that were created after the definition of the constraint.
FUNC_PATH	VARCHAR(254)		The current SQL path that was used when the constraint was created.
TEXT	CLOB(32K)		The text of the CHECK clause.



**SYSCAT.COLAUTH**

Contains one or more rows for each user or group who is granted a column level privilege, indicating the type of privilege and whether or not it is grantable.

*Table 38. SYSCAT.COLAUTH Catalog View*

Column Name	Data Type	Nullable	Description
GRANTOR	CHAR(8)		Authorization ID of the user who granted the privileges or SYSIBM.
GRANTEE	CHAR(8)		Authorization ID of the user or group who holds the privileges.
GRANTEEType	CHAR(1)		U=Grantee is an individual user G=Grantee is a group
TABSCHEMA	CHAR(8)		Qualified name of the table or view.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Name of the column to which this privilege applies.
COLNO	SMALLINT		Number of this column in the table or view.
PRIVTYPE	CHAR(1)		Indicates the type of privilege held on the table or view: U=update privilege. R=reference privilege.
GRANTABLE	CHAR(1)		Indicates if the privilege is grantable. G=grantable. N=not grantable.

## SYSCAT.COLCHECKS

---

### SYSCAT.COLCHECKS

Each row represents some column that is referenced by a CHECK constraint.

*Table 39. SYSCAT.COLCHECKS Catalog View*

<b>Column Name</b>	<b>Data Type</b>	<b>Nullable</b>	<b>Description</b>
CONSTNAME	VARCHAR(18)		Name of the check constraint. (Unique within a table. May be system generated.)
TABSCHEMA	CHAR(8)		Qualified name of table containing referenced column.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Name of column.

**SYSCAT.COLDIST**

Contains detailed column statistics for use by the optimizer. Each row describes the Nth-most-frequent value of some column.

*Table 40. SYSCAT.COLDIST Catalog View*

<b>Column Name</b>	<b>Data Type</b>	<b>Nullable</b>	<b>Description</b>
TABSCHEMA	CHAR(8)		Qualified name of the table to which this entry applies.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Name of the column to which this entry applies.
TYPE	CHAR(1)		F=Frequency (most frequent value) Q=Quantile value
SEQNO	SMALLINT		If TYPE=F, then N in this column identifies the Nth most frequent value. If TYPE=Q, then N in this column identifies the Nth quantile value.
COLVALUE	VARCHAR(33)	Yes	The data value, as a character literal or a null value.
VALCOUNT	INTEGER		If TYPE=F, then VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE=Q, then VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT	INTEGER	Yes	If TYPE=Q, this column records the number of distinct values that are less than or equal to COLVALUE (null if unavailable).

## SYSCAT.COLUMNS

---

### SYSCAT.COLUMNS

Contains one row for each column that is defined for a table or view. All of the catalog views have entries in the SYSCAT.COLUMNS table.

Table 41 (Page 1 of 2). SYSCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
TABSHEMA	CHAR(8)		Qualified name of the table or view that contains the column.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Column name.
COLNO	SMALLINT		Numerical place of column in table or view, beginning at zero.
TYPESHEMA	CHAR(8)		Contains the qualified name of the type, if the data type of the column is distinct. Otherwise TYPESHEMA contains the value SYSIBM and TYPENAME contains the data type of the column (in long form, for example, CHARACTER). If FLOAT or FLOAT( <i>n</i> ) with <i>n</i> greater than 24 is specified, TYPENAME is renamed to DOUBLE. If FLOAT( <i>n</i> ) with <i>n</i> less than 25 is specified, TYPENAME is renamed to REAL. Also, NUMERIC is renamed to DECIMAL.
TYPENAME	VARCHAR(18)		
LENGTH	INTEGER		Maximum length of data. 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields.
SCALE	SMALLINT		Scale for DECIMAL fields; 0 if not DECIMAL.
DEFAULT	VARCHAR(254)	Yes	<p>Default value for the column of a table expressed as a constant, special register, or cast-function appropriate for the data type of the column. May also be the keyword NULL.</p> <p>Values may be converted from what was specified as a default value. For example, date and time constants are presented in ISO format and cast-function names are qualified with schema name and the identifiers are delimited (see Note 3).</p> <p>Null value if a DEFAULT clause was not specified or the column is a view column.</p>
NULLS	CHAR(1)		<p>Y=Column is nullable. N=Column is not nullable.</p> <p>The value can be N for a view column that is derived from an expression or function. Nevertheless, such a column allows nulls when the statement using the view is processed with warnings for arithmetic errors.</p> <p>See Note 1.</p>
CODEPAGE	SMALLINT		Code page of the column. For character-string columns not defined with the FOR BIT DATA attribute, the value is the database code page. For graphic-string columns, the value is the DBCS code page implied by the (composite) database code page. Otherwise, the value is 0.

## SYSCAT.COLUMNS

Table 41 (Page 2 of 2). SYSCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
LOGGED	CHAR(1)		Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise).  Y=Column is logged. N=Column is not logged.
COMPACT	CHAR(1)		Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise).  Y=Column is compacted in storage. N=Column is not compacted.
COLCARD	INTEGER		Number of distinct values in the column; -1 if statistics are not gathered.
HIGH2KEY	VARCHAR(33)		Second highest value of the column. This field is empty if statistics are not gathered. See Note 2.
LOW2KEY	VARCHAR(33)		Second lowest value of the column. Empty if statistics not gathered. See Note 2.
AVGCOLLEN	INTEGER		Average column length. -1 if a long field or LOB, or statistics have not been collected.
KEYSEQ	SMALLINT	Yes	The column's numerical position within the table's primary key. This field is null or 0 if the column is not part of the primary key.
PARTKEYSEQ	SMALLINT	Yes	The column's numerical position within the table's partitioning key. This field is null or 0 if the column is not part of the partitioning key.
NQUANTILES	SMALLINT		Number of quantile values recorded in SYSCAT.SYSCOLDIST for this column; -1 if no statistics.
NMOSTFREQ	SMALLINT		Number of most-frequent values recorded in SYSCAT.COLDIST for this column; -1 if statistics not gathered.
REMARKS	VARCHAR(254)	Yes	User-supplied comment.

**Note:**

1. Starting with Version 2, value D (indicating not null with a default) is no longer used. Instead, use of WITH DEFAULT is indicated by a non-null value in the DEFAULT column.
2. Starting with Version 2, representation of numeric data has been changed to character literals. The size has been enlarged from 16 to 33 bytes.
3. For Version 2.1.0, cast-function names were not delimited and may still appear this way in the DEFAULT column. Also, some view columns included default values which will still appear in the DEFAULT column.

## SYSCAT.CONSTDEP

---

### SYSCAT.CONSTDEP

Contains a row for every dependency of a constraint on some other object.

*Table 42. SYSCAT.CONSTDEP Catalog View*

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint.
TABSCHEMA	CHAR(8)		Qualified name of the table to which the constraint applies.
TABNAME	VARCHAR(18)		
BTYPE	CHAR(1)		Type of object that the constraint depends on. Possible values: <ul style="list-style-type: none"><li>• F=function instance</li><li>• I=index instance</li><li>• R=structured type</li></ul>
BSCHEMA	CHAR(8)		Qualified name of object that the constraint depends on.
BNAME	VARCHAR(18)		

**SYSCAT.DATATYPES**

Contains a row for every data type, including built-in and user-defined types.

Table 43. SYSCAT.DATATYPES Catalog View

Column Name	Data Type	Nullable	Description
TYPESHEMA	CHAR(8)		Qualified name of the data type (for built-in types, TYPESHEMA is SYSIBM).
TYPENAME	VARCHAR(18)		
DEFINER	CHAR(8)		Authorization ID under which type was created.
SOURCESHEMA	CHAR(8)	Yes	Qualified name of the source type for distinct types.
SOURCENAME	VARCHAR(18)	Yes	Qualified name of the builtin type used as the reference type that is used as the representation for references to structured types. Null for other types.
METATYPE	CHAR(1)		<ul style="list-style-type: none"> <li>• S=System predefined type</li> <li>• T=Distinct type</li> <li>• R=structured type</li> </ul>
TYPEID	SMALLINT		The system generated internal identifier of the data type.
SOURCETYPEID	SMALLINT	Yes	Internal type ID of source type (null for built-in types). For user-defined structured types, this is the internal type ID of the reference representation type.
LENGTH	INTEGER		Maximum length of the type. 0 for system predefined parameterized types (for example, DECIMAL and VARCHAR). For user-defined structured types, this indicates the length of the reference representation type.
SCALE	SMALLINT		Scale for distinct types or reference representation types based on the system predefined DECIMAL type. 0 for all other types (including DECIMAL itself).
CODEPAGE	SMALLINT		Code page for character and graphic distinct types or reference representation types; 0 otherwise.
CREATE_TIME	TIMESTAMP		Creation time of the data type.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

## SYSCAT.DBAUTH

---

### SYSCAT.DBAUTH

Records the database authorities held by users.

Table 44. SYSCAT.DBAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	CHAR(8)		SYSIBM or authorization ID of the user who granted the privileges.
GRANTEE	CHAR(8)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U=Grantee is an individual user G=Grantee is a group
DBADMAUTH	CHAR(1)		Whether grantee holds DBADM authority over the database:  Y=Authority is held N=Authority is not held
CREATETABAUTH	CHAR(1)		Whether grantee can create tables in the database (CREATETAB):  Y=Privilege is held N=Privilege is not held
BINDADDAUTH	CHAR(1)		Whether grantee can create new packages in the database (BINDADD):  Y=Privilege is held N=Privilege is not held
CONNECTAUTH	CHAR(1)		Whether grantee can connect to the database (CONNECT):  Y=Privilege is held N=Privilege is not held
NOFENCEAUTH	CHAR(1)		Whether grantee holds privilege to create non-fenced functions.  Y=Privilege is held N=Privilege is not held
IMPLSCHEMAAUTH	CHAR(1)		Whether grantee can implicitly create schemas in the database (IMPLICIT_SCHEMA):  Y=Privilege is held N=Privilege is not held



**SYSCAT.EVENTMONITORS**

Contains a row for every event monitor that has been defined.

Table 45. SYSCAT.EVENTMONITORS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR(18)		Name of event monitor.
DEFINER	CHAR(8)		Authorization ID of definer of event monitor.
TARGET_TYPE	CHAR(1)		The type of the target to which event data is written. Values: F=File P=Pipe
TARGET	VARCHAR(246)		Name of the target to which event data is written. Absolute pathname of file, or absolute name of pipe.
MAXFILES	INTEGER	Yes	Maximum number of event files that this event monitor permits in an event path. Null if there is no maximum, or if the target-type is not FILE.
MAXFILESIZE	INTEGER	Yes	Maximum size (in 4K pages) that each event file can reach before the event monitor creates a new file. Null if there is no maximum, or if the target-type is not FILE.
BUFFERSIZE	INTEGER	Yes	Size of buffers (in 4K pages) used by event monitors with file targets; otherwise null.
IO_MODE	CHAR(1)	Yes	Mode of file I/O. B=Blocked N=Not blocked. Null if target-type is not FILE.
WRITE_MODE	CHAR(1)	Yes	Indicates how this monitor handles existing event data when the monitor is activated. Values: A=Append R=Replace Null if target-type is not FILE.
AUTOSTART	CHAR(1)		The event monitor will be activated automatically when the database starts. Y=Yes N=No
NODENUM	SMALLINT		The number of the partition (or node) on which the event monitor runs and logs events
MONSCOPE	CHAR(1)		Monitoring scope: L=Local G=Global
REMARKS	VARCHAR(254)	Yes	Reserved for future use.

## SYSCAT.EVENTS

---

### SYSCAT.EVENTS

Contains a row for every event that is being monitored. An event monitor, in general, monitors multiple events.

*Table 46. SYSCAT.EVENTS Catalog View*

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR(18)		Name of event monitor that is monitoring this event.
TYPE	VARCHAR(18)		Type of event being monitored. Possible values: DATABASE CONNECTIONS TABLES STATEMENTS TRANSACTIONS DEADLOCKS TABLESPACES
FILTER	CLOB(32K)	Yes	The full text of the WHERE-clause that applies to this event.

**SYSCAT.FUNCPARMS**

Contains a row for every parameter or result of a function defined in SYSCAT.FUNCTIONS.

*Table 47. SYSCAT.FUNCPARMS Catalog View*

Column Name	Data Type	Nullable	Description
FUNCSHEMA	CHAR(8)		Qualified function name.
FUNCNAME	VARCHAR(18)		
SPECIFICNAME	VARCHAR(18)		The name of the function instance (may be system-generated).
ROWTYPE	CHAR(1)		P=parameter R=result before casting C=result after casting
ORDINAL	SMALLINT		If ROWTYPE=P, the parameter's numerical position within the function signature. Otherwise 0.
PARAMNAME	VARCHAR(18)		Name of parameter or result column, or null if no name exists.
TYPESHEMA	CHAR(8)		Qualified name of data type of parameter or result.
TYPENAME	VARCHAR(18)		
LENGTH	INTEGER		Length of parameter or result. 0 if parameter or result is a distinct type. See Note 1.
SCALE	SMALLINT		Scale of parameter or result. 0 if parameter or result is a distinct type. See Note 1.
CODEPAGE	SMALLINT		Code page of parameter. 0 denotes either not applicable or a column for character data declared with the FOR BIT DATA attribute.
CAST_FUNCID	INTEGER	Yes	Internal function ID.
AS_LOCATOR	CHAR(1)		Y=Parameter or result is passed in the form of a locator N=Not passed in the form of a locator.

**Note:**

1. LENGTH and SCALE are set to 0 for sourced functions (functions defined with a reference to another function) because they inherit the length and scale of parameters from their source.

## SYSCAT.FUNCTIONS

### SYSCAT.FUNCTIONS

Contains a row for each user-defined function (scalar, table or sourced). Does not include built-in functions.

Table 48 (Page 1 of 3). SYSCAT.FUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCSHEMA	CHAR(8)		Qualified function name.
FUNCNAME	VARCHAR(18)		
SPECIFICNAME	VARCHAR(18)		The name of the function instance (may be system-generated).
DEFINER	CHAR(8)		Authorization ID of function definer.
FUNCID	INTEGER		Internally-assigned function ID.
RETURN_TYPE	SMALLINT		Internal type code of return type of function.
ORIGIN	CHAR(1)		B=Built-in E=User-defined, external U=User-defined, based on a source S=System-generated
TYPE	CHAR(1)		S=Scalar function C=Column function T=Table function
PARAM_COUNT	SMALLINT		Number of function parameters.
PARAM_SIGNATURE	VARCHAR(180) FOR BIT DATA		Concatenation of up to 90 parameter types, in internal format. Zero length if function takes no parameters.
CREATE_TIME	TIMESTAMP		Timestamp of function creation. Set to 0 for Version 1 functions.
VARIANT	CHAR(1)		Y=Variant (results may differ) N=Invariant (results are consistent) Blank if ORIGIN is not E
SIDE_EFFECTS	CHAR(1)		E=Function has external side-effects (number of invocations is important) N=No side-effects Blank if ORIGIN is not E
FENCED	CHAR(1)		Y=Fenced N=Not fenced Blank if ORIGIN is not E
NULLCALL	CHAR(1)		Y=Nullcall N=No nullcall (function result is implicitly null if operand(s) are null). Blank if ORIGIN is not E.
CAST_FUNCTION	CHAR(1)		Y=This is a cast function N=This is not a cast function
ASSIGN_FUNCTION	CHAR(1)		Y=Implicit assignment function N=Not an assignment function

## SYSCAT.FUNCTIONS

Table 48 (Page 2 of 3). SYSCAT.FUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
SCRATCHPAD	CHAR(1)		Y=This function has a scratch pad N=This function does not have a scratch pad Blank if ORIGIN is not E
FINAL_CALL	CHAR(1)		Y=Final call is made to this function at run time end-of-statement. N=No final call is made. Blank if ORIGIN is not E
PARALLELIZABLE	CHAR(1)		Y=Function can be executed in parallel N=Function cannot be executed in parallel Blank if ORIGIN is not E
CONTAINS_SQL	CHAR(1)		Indicates wheter an external function contains SQL.  N=Function does not contain SQL statements. R=Contains read-only SQL statements. M=Contains SQL statements that modify data. Blank if ORIGIN is not E
DBINFO	CHAR(1)		Indicates whether a DBINFO parameter is passed to an external function.  Y=DBINFO is passed. N=DBINFO is not passed. Blank if ORIGIN is not E
RESULT_COLS	SMALLINT		For a table function (TYPE=T) contains the number of columns in the result table; otherwise contains 1.
LANGUAGE	CHAR(8)		Implementation language of function body. Possible values are C, JAVA or OLE. Blank if ORIGIN is not E.
IMPLEMENTATION	VARCHAR(254)	Yes	If ORIGIN=E, identifies the path/module/function that implements this function. If ORIGIN=U and the source function is built-in, this column contains the name and signature of the source function. Null otherwise.
PARAM_STYLE	CHAR(8)		Indicates the parameter style declared in the CREATE FUNCTION statement. Values:  DB2SQL DB2GENRL
SOURCE_SCHEMA	CHAR(8)	Yes	If ORIGIN=U and the source function is a user-defined function, contains the qualified name of the source function. If ORIGIN=U and the source function is built-in, SOURCE_SCHEMA is 'SYSIBM' and SOURCE_SPECIFIC is 'N/A for built-in'. Null if ORIGIN is not U.
SOURCE_SPECIFIC	VARCHAR(18)	Yes	
IOS_PER_INVOC	DOUBLE		Estimated number of I/Os per invocation; -1 if not known (0 default).

## SYSCAT.FUNCTIONS

Table 48 (Page 3 of 3). SYSCAT.FUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
INSTS_PER_INVOC	DOUBLE		Estimated number of instructions per invocation; -1 if not known (450 default).
IOS_PER_ARGBYTE	DOUBLE		Estimated number of I/O's per input argument byte; -1 if not known (0 default).
INSTS_PER_ARGBYTE	DOUBLE		Estimated number of instructions per input argument byte; -1 if not known (0 default).
PERCENT_ARGBYTES	SMALLINT		Estimated average percent of input argument bytes that the function will actually read; -1 if not known (100 default).
INITIAL_IOS	DOUBLE		Estimated number of I/O's performed the first/last time the function is invoked; -1 if not known (0 default).
INITIAL_INSTS	DOUBLE		Estimated number of instructions executed the first/last time the function is invoked; -1 if not known (0 default).
CARDINALITY	INTEGER	Yes	The predicted cardinality of a table function. -1 if not known or if function is not a table function.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

**SYSCAT.INDEXAUTH**

Contains a row for every privilege held on an index.

*Table 49. SYSCAT.INDEXAUTH Catalog View*

Column Name	Data Type	Nullable	Description
GRANTOR	CHAR(8)		Authorization ID of the user who granted the privileges.
GRANTEE	CHAR(8)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U=Grantee is an individual user G=Grantee is a group
INDSCHEMA	CHAR(8)		Name of the index.
INDNAME	VARCHAR(18)		
CONTROLAUTH	CHAR(1)		Whether grantee holds CONTROL privilege over the index: Y=Privilege is held N=Privilege is not held

## SYSCAT.INDEXES

---

### SYSCAT.INDEXES

Contains one row for each index that is defined for a table.

Table 50 (Page 1 of 3). SYSCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	CHAR(8)		Name of the index.
INDNAME	VARCHAR(18)		
DEFINER	CHAR(8)		User who created the index.
TABSCHEMA	CHAR(8)		Qualified name of the table on which the index is defined.
TABNAME	VARCHAR(18)		
COLNAMES	VARCHAR(320)		List of column names, each preceded by + or – to indicate ascending or descending order respectively.
UNIQUERULE	CHAR(1)		Unique rule: <ul style="list-style-type: none"><li>• D=duplicates allowed</li><li>• P=primary index.</li><li>• U=unique entries only allowed</li></ul>
MADE_UNIQUE	CHAR(1)		<ul style="list-style-type: none"><li>• Y=Index was originally non-unique but was converted to a unique index to support a unique or primary key constraint. If the constraint is dropped, the index will revert to non-unique.</li><li>• N=Index remains as it was created.</li></ul>
COLCOUNT	SMALLINT		Number of columns in the key plus the number of include columns if any.
UNIQUE_COLCOUNT	SMALLINT		The number of columns required for a unique key. Always <=COLCOUNT. < COLCOUNT only if there a include columns. -1 if index has no unique key (permits duplicates)
INDEXTYPE	CHAR(4)		Type of index. <ul style="list-style-type: none"><li>• CLUS =Clustering</li><li>• REG =Regular</li></ul>
PCTFREE	SMALLINT		Percentage of each index leaf page to be reserved during initial building of the index. This space is available for future inserts after the index is built.
IID	SMALLINT		Internal index ID.
NLEAF	INTEGER		Number of leaf pages; <ul style="list-style-type: none"><li>• -1 if statistics are not gathered.</li></ul>
NLEVELS	SMALLINT		Number of index levels; -1 if statistics are not gathered.



## SYSCAT.INDEXES

Table 50 (Page 2 of 3). SYSCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
FIRSTKEYCARD	INTEGER		Number of distinct first key values; -1 if statistics are not gathered.
FIRST2KEYCARD	INTEGER		Number of distinct keys using the first two columns of the index (-1 if no statistics or inapplicable)
FIRST3KEYCARD	INTEGER		Number of distinct keys using the first three columns of the index (-1 if no statistics or inapplicable)
FIRST4KEYCARD	INTEGER		Number of distinct keys using the first four columns of the index (-1 if no statistics or inapplicable)
FULLKEYCARD	INTEGER		Number of distinct full key values; -1 if statistics are not gathered.
CLUSTERRATIO	SMALLINT		Degree of data clustering with the index; -1 if statistics are not gathered or if detailed index statistics are gathered (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR	DOUBLE		Finer measurement of degree of clustering, or -1 if detailed index statistics have not been gathered.
SEQUENTIAL_PAGES	INTEGER		Number of leaf pages located on disk in index key order with few or no large gaps between them. (-1 if no statistics are available.)
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100, -1 if no statistics are available.)
USER_DEFINED	SMALLINT		1 if this index was defined by a user and has not been dropped; otherwise 0.
SYSTEM_REQUIRED	SMALLINT		<ul style="list-style-type: none"> <li>• 1 if this index is required for primary key or unique key constraint, OR if this is the index on the object identifier (OID) column of a typed table.</li> <li>• 2 if this index is required for primary key or unique key constraint, AND this is the index on the object identifier (OID) column of a typed table.</li> <li>• 0 otherwise.</li> </ul>
CREATE_TIME	TIMESTAMP		Time when the index was created.
STATS_TIME	TIMESTAMP	Yes	Last time when any change was made to recorded statistics for this index. Null if no statistics available.

## SYSCAT.INDEXES

Table 50 (Page 3 of 3). SYSCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
PAGE_FETCH_PAIRS	VARCHAR(254)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available.)
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.
TEXT	CLOB(32K)	Yes	Reserved for future use.

---

**SYSCAT.KEYCOLUSE**

Lists all columns that participate in a key defined by a unique, primary key, or foreign key constraint.

*Table 51. SYSCAT.KEYCOLUSE Catalog View*

<b>Column Name</b>	<b>Data Type</b>	<b>Nullable</b>	<b>Description</b>
CONSTNAME	VARCHAR(18)		Name of the constraint (unique within a table).
TABSCHEMA	CHAR(8)		Qualified name of the table containing the column.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the key (initial position=1).

## SYSCAT.NODEGROUPDEF

---

### SYSCAT.NODEGROUPDEF

Contains a row for each partition that is contained in a nodegroup.

*Table 52. SYSCAT.NODEGROUPDEF Catalog View*

Column Name	Data Type	Nullable	Description
NGNAME	VARCHAR(18)		The name of the nodegroup that contains the partition (or node) .
NODENUM	SMALLINT		The partition (or node) number of a partition contained in the nodegroup. A valid partition number is between 0 and 999 inclusive.
IN_USE	CHAR(1)		Status of the partition (or node) . A The newly added partition is not in the partitioning map but the containers for the table spaces in the nodegroup are created. The partition is added to the partitioning map when a Redistribute Nodegroup operation is successfully completed. D The partition will be dropped when a Redistribute Nodegroup operation is completed. T The newly added partition is not in the partitioning map and it was added using the WITHOUT TABLESPACES clause. Containers must be specifically added to the table spaces for the nodegroup. Y The partition is in the partitioning map.

## SYSCAT.NODEGROUPS

---

### SYSCAT.NODEGROUPS

Contains a row for each nodegroup.

*Table 53. SYSCAT.NODEGROUPS Catalog View*

Column Name	Data Type	Nullable	Description
NGNAME	VARCHAR(18)		Name of the nodegroup.
DEFINER	CHAR(8)		Authorization ID of the nodegroup definer.
PMAP_ID	SMALLINT		Identifier of the partitioning map in SYSCAT.PARTITIONMAPS.
REBALANCE_PMAP_ID	SMALLINT		Identifier of the partitioning map currently being used for redistribution. Value is -1 if redistribution is currently not in progress.
CREATE_TIME	TIMESTAMP		Creation time of nodegroup.
REMARKS	VARCHAR(254)	Yes	User-provided comment.

## SYSCAT.PACKAGEAUTH

---

### SYSCAT.PACKAGEAUTH

Contains a row for every privilege held on a package.

Table 54. SYSCAT.PACKAGEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	CHAR(8)		Authorization ID of the user who granted the privileges.
GRANTEE	CHAR(8)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U=Grantee is an individual user G=Grantee is a group
PKGSHEMA	CHAR(8)		Name of the package on which the privileges are held.
PKGNAME	CHAR(8)		
CONTROLAUTH	CHAR(1)		Indicates whether grantee holds CONTROL privilege on the package: Y=Privilege is held. N=Privilege is not held.
BINDAUTH	CHAR(1)		Indicates whether grantee holds BIND privilege on the package: Y=Privilege is held. N=Privilege is not held.
EXECUTEAUTH	CHAR(1)		Indicates whether grantee holds EXECUTE privilege on the package: Y=Privilege is held. N=Privilege is not held.

**SYSCAT.PACKAGEDEP**

Contains a row for each dependency that packages have on indexes, tables, views, functions, aliases, types, and hierarchies.

Table 55. SYSCAT.PACKAGEDEP Catalog View

Column Name	Data Type	Nullable	Description
PKGSHEMA	CHAR(8)		Name of the package.
PKGNAME	CHAR(8)		
BINDER	CHAR(8)	Yes	Binder of the package.
BTYPE	CHAR(1)		Type of object BNAME: <ul style="list-style-type: none"> <li>• A=alias</li> <li>• F=function-instance</li> <li>• H=table or view hierarchy</li> <li>• I=index</li> <li>• R=structured type</li> <li>• S=summary table</li> <li>• T=table</li> <li>• V=view</li> </ul>
BSCHEMA	CHAR(8)		Qualified name of an object on which the package is dependent.
BNAME	VARCHAR(18)		
TBAUTH	SMALLINT	Yes	If BTYPE is T(table) or V(view), encodes the privileges that are required by this package (Select, Insert, Delete, Update).

**Note:**

1. When a depended-on function-instance is dropped, the package is placed into an "inoperative" state from which it must be explicitly rebound. When any other depended-on object is dropped, the package is placed into an "invalid" state from which the system will attempt to rebound it automatically when a package is first referenced.

## SYSCAT.PACKAGES

---

### SYSCAT.PACKAGES

Contains a row for each package that has been created by binding an application program.

Table 56 (Page 1 of 3). SYSCAT.PACKAGES Catalog View

Column Name	Data Type	Nullable	Description
PKGSHEMA	CHAR(8)		Name of the package.
PKGNAME	CHAR(8)		
BOUNDBY	CHAR(8)		Authorization ID of the binder of the package.
DEFINER	CHAR(8)		Userid under which package was bound.
DEFAULT_SCHEMA	CHAR(8)		Default schema name used for unqualified names in static SQL statements.
VALID	CHAR(1)		Y=Valid N=Not valid X=Package is inoperative because some function instance that it depends on has been dropped. Explicit rebind is needed. See Note 1 on "SYSCAT.PACKAGEDEP" on page 801
UNIQUE_ID	CHAR(8)		Internal date and time information indicating when the package was first created.
TOTAL_SECT	SMALLINT		Total number of sections in the package.
FORMAT	CHAR(1)		Date and time format associated with the package:  0=Format associated with country code of the database 1=USA date and time 2=EUR date, EUR time 3=ISO date, ISO time. 4=JIS date, JIS time. 5=LOCAL date, LOCAL time.
ISOLATION	CHAR(2)	Yes	Isolation level:  RR=Repeatable read RS=Read stability CS=Cursor stability UR=Uncommitted read.
BLOCKING	CHAR(1)	Yes	Cursor blocking option:  N=No blocking U=Block unambiguous cursors B=Block all cursors
INSERT_BUF	CHAR(1)		Insert option used during bind:  Y=Inserts are buffered N=Inserts are not buffered



## SYSCAT.PACKAGES

Table 56 (Page 2 of 3). SYSCAT.PACKAGES Catalog View

Column Name	Data Type	Nullable	Description
LANG_LEVEL	CHAR(1)	Yes	LANGLEVEL value used during BIND: 0=SAA1 1=SQL92E or MIA
FUNC_PATH	VARCHAR(254)		The SQL path used by the last BIND command for this package. This is used as the default path for REBIND. SYSIBM for pre-Version 2 packages.
QUERYOPT	INTEGER		Optimization class under which this package was bound. Used for rebind. The classes are: 0, 1, 3, 5 and 9. See .
EXPLAIN_LEVEL	CHAR(1)		Indicates whether Explain was requested using the EXPLAIN or EXPLSNAP bind option.  Blank=No Explain requested P=Plan Selection level
EXPLAIN_MODE	CHAR(1)		Value of EXPLAIN bind option:  Y=Yes (static) N=No A=All (static and dynamic)
EXPLAIN_SNAPSHOT	CHAR(1)		Value of EXPLSNAP bind option:  Y=Yes (static) N=No A=All (static and dynamic)
SQLWARN	CHAR(1)		Are positive SQLCODES resulting from dynamic SQL statements returned to the application?  Y=Yes N=No, they are suppressed
SQLMATHWARN	CHAR(1)		Value of database configuration parameter DFT_SQLMATHWARN at time of bind. Are arithmetic errors and retrieval conversion errors in static SQL statements handled as nulls with a warning?  Y=Yes N=No, they are suppressed
EXPLICIT_BIND_TIME	TIMESTAMP		The time at which this package was last explicitly bound or rebound. When the package is implicitly rebound, no function instance will be selected that was created later than this time.
LAST_BIND_TIME	TIMESTAMP		Time at which the package last explicitly or implicitly bound or rebound.
CODEPAGE	SMALLINT		Application codepage at bind time (-1 if not known).

## SYSCAT.PACKAGES

Table 56 (Page 3 of 3). SYSCAT.PACKAGES Catalog View

Column Name	Data Type	Nullable	Description
DEGREE	CHAR(5)		Indicates the limit on intra-partition parallelism (as a bind option) when package was bound.  1 = No intra-partition parallelism. 2 - 32767 = Degree of intra-partition parallelism. ANY = Degree was determined by the database manager.
MULTINODE_PLANS	CHAR(1)		Y =Package was bound in a multiple partition environment. N =Package was bound in a single partition environment.
INTRA_PARALLEL	CHAR(1)		Indicates the use of intra-partition parallelism by static SQL statements within the package.  Y = one or more static SQL statement in package uses intra-partition parallelism. N = no static SQL statement in package uses intra-partition parallelism. F = one or more static SQL statement in package can use intra-partition parallelism; this parallelism has been disabled for use on a system that is not configured for intra-partition parallelism.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

## SYSCAT.PARTITIONMAPS

---

### SYSCAT.PARTITIONMAPS

Contains a row for each partitioning map that is used to distribute the rows of tables among the partitions in a nodegroup, based on hashing the tables partitioning key.

*Table 57. SYSCAT.PARTITIONMAPS Catalog View*

Column Name	Data Type	Nullable	Description
PMAP_ID	SMALLINT		Identifier of the partitioning map.
PARTITIONMAP	LONG VARCHAR FOR BIT DATA		The actual partitioning map, a vector of 4096 two-byte integers for a multiple node nodegroup. For a single node nodegroup, there is one entry denoting the partition (or node) number of the single node.

## SYSCAT.PROCEDURES

---

### SYSCAT.PROCEDURES

Contains a row for each stored procedure that is created.

Table 58. SYSCAT.PROCEDURES Catalog View

Column Name	Data Type	Nullable	Description
PROCSHEMA	CHAR(8)		Qualified procedure name.
PROCNAME	VARCHAR(18)		
SPECIFICNAME	VARCHAR(18)		The name of the procedure instance (may be system generated).
PROCEDURE_ID	INTEGER		Internal ID of stored procedure.
DEFINER	CHAR(8)		Authorization of the procedure definer.
PARAM_COUNT	SMALLINT		Number of procedure parameters.
PARAM_SIGNATURE	VARCHAR(180) FOR BIT DATA		Concatenation of up to 90 parameter types, in internal format. Zero length if procedure takes no parameters.
ORIGIN	CHAR(1)		Always 'E' = User defined, external
CREATE_TIME	TIMESTAMP		Timestamp of procedure registration.
DETERMINISTIC	CHAR(1)		Y=Results are deterministic. N=Results are not deterministic.
FENCED	CHAR(1)		Y=Fenced N=Not Fenced
NULLCALL	CHAR(1)		Always Y=NULLCALL
LANGUAGE	CHAR(8)		Implementation language of procedure body. Possible values are C and JAVA.
IMPLEMENTATION	VARCHAR(254)	Yes	Identifies the path/module/function or class/method that implements the procedure.
PARAM_STYLE	CHAR(8)		DB2DARI=Language is C DB2GENRL=Language is Java
RESULT_SETS	SMALLINT		Estimated upper limit of returned result sets.
REMARKS	VARCHAR(254)	Yes	User supplied comment, or null.

**SYSCAT.PROCPARMS**

Contains a row for each parameter of a stored procedure.

*Table 59. SYSCAT.PROCPARMS Catalog View*

Column Name	Data Type	Nullable	Description
PROCSHEMA	CHAR(8)		Qualified procedure name.
PROCNAME	VARCHAR(18)		
SPECIFICNAME	VARCHAR(18)		The name of the procedure instance (may be system generated).
ORDINAL	SMALLINT		The parameter's numerical position within the procedure signature.
PARAMNAME	VARCHAR(18)		Parameter name.
TYPESHEMA	CHAR(8)		Qualified name of data type of the parameter.
TYPENAME	VARCHAR(18)		
LENGTH	INTEGER		Length of the parameter.
SCALE	SMALLINT		Scale of the parameter.
CODEPAGE	SMALLINT		Code page of parameter. 0 denotes either not applicable or a parameter for character data declared with the FOR BIT DATA attribute.
PARAM_MODE	VARCHAR(5)		IN=Input, OUT=Output, INOUT=Input/output
AS_LOCATOR	CHAR(1)		Always 'N'

## SYSCAT.REFERENCES

---

### SYSCAT.REFERENCES

Contains a row for each defined referential constraint.

Table 60. SYSCAT.REFERENCES Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of constraint.
TABSCHEMA	CHAR(8)		Qualified name of the constraint.
TABNAME	VARCHAR(18)		
DEFINER	CHAR(8)		User who created the constraint.
REFKEYNAME	VARCHAR(18)		Name of parent key.
REFTABSCHEMA	CHAR(8)		Name of the parent table.
REFTABNAME	VARCHAR(18)		
COLCOUNT	SMALLINT		Number of columns in the foreign key.
DELETERULE	CHAR(1)		Delete rule: A=NO ACTION C=CASCADE N=SET NULL R=RESTRICT
UPDATERULE	CHAR(1)		Update rule: A=NO ACTION R=RESTRICT
CREATE_TIME	TIMESTAMP		The timestamp when the referential constraint was defined.
FK_COLNAMES	VARCHAR(320)		List of foreign key column names.
PK_COLNAMES	VARCHAR(320)		List of parent key column names.

**Note:**

1. The SYSCAT.REFERENCES view is based on the SYSIBM.SYSRELS table from Version 1.

**SYSCAT.SCHEMAAUTH**

Contains one or more rows for each user or group who is granted a privilege on a particular schema in the database. All schema privileges for a single schema granted by a specific grantor to a specific grantee appear in a single row.

*Table 61. SYSCAT.SCHEMAAUTH Catalog View*

<b>Column Name</b>	<b>Data Type</b>	<b>Nullable</b>	<b>Description</b>
GRANTOR	CHAR(8)		Authorization ID of the user who granted the privileges or SYSIBM.
GRANTEE	CHAR(8)		Authorization ID of the user or group who holds the privileges.
GRANTEEType	CHAR(1)		U=Grantee is an individual user G=Grantee is a group
SCHEMANAME	CHAR(8)		Name of the schema.
ALTERINAUTH	CHAR(1)		Indicates whether grantee holds ALTERIN privilege on the schema:  Y=Privilege is held G=Privilege is held and grantable N=Privilege is not held.
CREATEINAUTH	CHAR(1)		Indicates whether grantee holds CREATEIN privilege on the schema:  Y=Privilege is held G=Privilege is held and grantable N=Privilege is not held.
DROPINAUTH	CHAR(1)		Indicates whether grantee holds DROPIN privilege on the schema:  Y=Privilege is held G=Privilege is held and grantable N=Privilege is not held.

## SYSCAT.SCHEMATA

---

### SYSCAT.SCHEMATA

Contains a row for each schema.

*Table 62. SYSCAT.SCHEMATA Catalog View*

Column Name	Data Type	Nullable	Description
SCHEMANAME	CHAR(8)		Name of the schema.
OWNER	CHAR(8)		Authorization id of the schema. The value for implicitly created schemas is SYSIBM.
DEFINER	CHAR(8)		User who created the schema.
CREATE_TIME	TIMESTAMP		Timestamp indicating when the object was created.
REMARKS	VARCHAR(254)	Yes	User-provided comment.



## SYSCAT.STATEMENTS

---

### SYSCAT.STATEMENTS

Contains one or more rows for each SQL statement in each package in the database.

*Table 63. SYSCAT.STATEMENTS Catalog View*

Column Name	Data Type	Nullable	Description
PKGSHEMA	CHAR(8)		Name of the package.
PKGNAME	CHAR(8)		
STMTNO	SMALLINT		Line number of the SQL statement in the source module of the application program.
SECTNO	SMALLINT		Number of the package section containing the SQL statement.
SEQNO	SMALLINT		Sequence number of this row; the first portion of the SQL text is stored on row one, and successive rows have increasing values for SEQNO.
TEXT	VARCHAR (3600)		Text or portion of the text of the SQL statement.

## SYSCAT.TABAUTH

---

### SYSCAT.TABAUTH

Contains one or more rows for each user or group who is granted a privilege on a particular table or view in the database. All the table privileges for a single table or view granted by a specific grantor to a specific grantee appear in a single row.

Table 64 (Page 1 of 2). SYSCAT.TABAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	CHAR(8)		Authorization ID of the user who granted the privileges or SYSIBM.
GRANTEE	CHAR(8)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U=Grantee is an individual user G=Grantee is a group
TABSHEMA	CHAR(8)		Qualified name of the table or view.
TABNAME	VARCHAR(18)		
CONTROLAUTH	CHAR(1)		Indicates whether grantee holds CONTROL privilege on the table or view:  Y=Privilege is held. N=Privilege is not held.
ALTERAUTH	CHAR(1)		Indicates whether grantee holds ALTER privilege on the table:  Y=Privilege is held. N=Privilege is not held. G=Privilege is held and grantable.
DELETEAUTH	CHAR(1)		Indicates whether grantee holds DELETE privilege on the table or view:  Y=Privilege is held. N=Privilege is not held. G=Privilege is held and grantable.
INDEXAUTH	CHAR(1)		Indicates whether grantee holds INDEX privilege on the table:  Y=Privilege is held. N=Privilege is not held. G=Privilege is held and grantable.
INSERTAUTH	CHAR(1)		Indicates whether grantee holds INSERT privilege on the table or view:  Y=Privilege is held. N=Privilege is not held. G=Privilege is held and grantable.
SELECTAUTH	CHAR(1)		Indicates whether grantee holds SELECT privilege on the table or view:  Y=Privilege is held. N=Privilege is not held. G=Privilege is held and grantable.

## SYSCAT.TABAUTH

Table 64 (Page 2 of 2). SYSCAT.TABAUTH Catalog View

Column Name	Data Type	Nullable	Description
REFAUTH	CHAR(1)		Indicates whether grantee holds REFERENCE privilege on the table or view:  Y=Privilege is held. N=Privilege is not held. G=Privilege is held and grantable.
UPDATEAUTH	CHAR(1)		Indicates whether grantee holds UPDATE privilege on the table or view:  Y=Privilege is held. N=Privilege is not held. G=Privilege is held and grantable.

## SYSCAT.TABCONST

---

### SYSCAT.TABCONST

Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY.

*Table 65. SYSCAT.TABCONST Catalog View*

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint (unique within a table).
TABSCHEMA	CHAR(8)		Qualified name of the table to which this constraint applies.
TABNAME	VARCHAR(18)		
DEFINER	CHAR(8)		Authorization ID under which the constraint was defined.
TYPE	CHAR(1)		Indicates the constraint type: K=CHECK P=PRIMARY KEY F=FOREIGN KEY U=UNIQUE
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

---

**SYSCAT.TABLES**

Contains one row for each table, view, or alias that is created. All of the catalog tables and views have entries in the SYSCAT.TABLES catalog view.

Table 66 (Page 1 of 3). SYSCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	CHAR(8)		Qualified name of the table, view, or alias.
TABNAME	VARCHAR(18)		
DEFINER	CHAR(8)		User who created the table, view, or alias.
TYPE	CHAR(1)		The type of object: A=Alias S=Summary table T=Table V=View
STATUS	CHAR(1)		The type of object: N=Normal table, view or alias C=Check pending on table X=Inoperative view
BASE_TABSCHEMA	CHAR(8)	Yes	If TYPE=A, these columns identify the table, view, or alias that is referenced by this alias; otherwise they are null.
BASE_TABNAME	VARCHAR(18)	Yes	
CREATE_TIME	TIMESTAMP		The timestamp indicating when the object was created.
STATS_TIME	TIMESTAMP	Yes	Last time when any change was made to recorded statistics for this table. Null if no statistics available.
COLCOUNT	SMALLINT		Number of columns in table.
TABLEID	SMALLINT		Internal table identifier.
TBSPACEID	SMALLINT		Internal identifier of primary table space for this table.
CARD	INTEGER		Total number of rows in the table; -1 if statistics are not gathered or the row describes a view or alias.
NPAGES	INTEGER		Total number of pages on which the rows of the table exist; -1 if statistics are not gathered or the row describes a view or alias; -2 for a subtable.
FPAGES	INTEGER		Total number of pages; -1 if statistics are not gathered or the row describes a view or alias; -2 for a subtable.
OVERFLOW	INTEGER		Total number of overflow records in the table; -1 if statistics are not gathered or the row describes a view or alias; -2 for a subtable.
TBSPACE	VARCHAR(18)	Yes	Name of primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. Null for aliases and views.

## SYSCAT.TABLES

Table 66 (Page 2 of 3). SYSCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
INDEX_TBSPACE	VARCHAR(18)	Yes	Name of table space that holds all indexes created on this table. Null for aliases and views, or if the INDEX IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
LONG_TBSPACE	VARCHAR(18)	Yes	Name of table space that holds all long data (LONG or LOB column types) for this table. Null for aliases and views, or if the LONG IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
PARENTS	SMALLINT	Yes	Number of parent tables of this table (the number of referential constraints in which this table is a dependent).
CHILDREN	SMALLINT	Yes	Number of dependent tables of this table (the number of referential constraints in which this table is a parent).
SELFREFS	SMALLINT	Yes	Number of self-referencing referential constraints for this table (the number of referential constraints in which this table is both a parent and a dependent).
KEYCOLUMNS	SMALLINT	Yes	Number of columns in the primary key of the table.
KEYINDEXID	SMALLINT	Yes	Index ID of the primary index. This field is null or 0 if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique constraints (other than primary key) defined on this table.
CHECKCOUNT	SMALLINT		Number of check constraints defined on this table.
DATA_CAPTURE	CHAR(1)		Y=Table participates in data replication N=Does not participate
CONST_CHECKED	CHAR(32)		Byte 1 represents foreign key constraints. Byte 2 represents check constraints. Byte 3 represents constraint Datalink_Reconcile_Pending. Byte 4 represents constraint Datalink_Reconcile_Not_Possible. Byte 5 represents summary table. Other bytes are reserved. Encodes constraint information on checking. Values:  Y=Checked by system U=Checked by user N=Not checked (pending)
PMAP_ID	SMALLINT	Yes	Identifier of the partitioning map used by this table. Null for aliases and views.
PARTITION_MODE	CHAR(1)		Mode used for tables in a partitioned database.  <b>H</b> hash on the partitioning key <b>R</b> table replicated across database partitions  Blank for aliases, views and tables in single partition nodegroups with no partitioning key defined.

## SYSCAT.TABLES

Table 66 (Page 3 of 3). SYSCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
LOG_ATTRIBUTE	CHAR(1)		0=Default logging 1=Table created not logged initially
PCTFREE	SMALLINT		Percentage of each page to be reserved for future inserts. Can be changed by ALTER TABLE.
REMARKS	VARCHAR(254)	Yes	User-provided comment.

## SYSCAT.TABLESPACES

---

### SYSCAT.TABLESPACES

Contains a row for each table space.

Table 67. SYSCAT.TABLESPACES Catalog View

Column Name	Data Type	Nullable	Description
TBSPACE	VARCHAR(18)		Name of table space.
DEFINER	CHAR(8)		Authorization ID of table space definer.
CREATE_TIME	TIMESTAMP		Creation time of table space.
TBSPACEID	INTEGER		Internal table space identifier.
TBSPACETYPE	CHAR(1)		The type of the table space: S=System managed space D=Database managed space
DATATYPE	CHAR(1)		Type of data that can be stored: A=All types of permanent data L=Long data only T=Temporary tables only
EXTENTS_SIZE	INTEGER		Size of extent, in 4K pages. This many pages are written to one container in the table space before switching to the next container.
PREFETCH_SIZE	INTEGER		Number of 4K pages to be read when prefetch is performed.
OVERHEAD	DOUBLE		Controller overhead and disk seek and latency time in milliseconds.
TRANSFERRATE	DOUBLE		Time to read one 4K page into the buffer.
PAGESIZE	INTEGER		Size (in bytes) of pages in the table space.
NGNAME	VARCHAR(18)		Name of the nodegroup for the table space.
BUFFERPOOLID	INTEGER		ID of buffer pool used by this tablespace (1 indicates default buffer pool).
REMARKS	VARCHAR(254)	Yes	User-provided comment.



**SYSCAT.TRIGDEP**

Contains a row for every dependency of a trigger on some other object.

Table 68. SYSCAT.TRIGDEP Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	CHAR(8)		Qualified name of the trigger.
TRIGNAME	VARCHAR(18)		
BTYPE	CHAR(1)		Type of object BNAME: <ul style="list-style-type: none"> <li>• A=Alias</li> <li>• F=Function instance</li> <li>• H=Table or view hierarchy</li> <li>• R=Structured type</li> <li>• S=Summary table</li> <li>• T=Table</li> <li>• V=View</li> </ul>
BSCHEMA	CHAR(8)		Qualified name of object depended on by a trigger.
BNAME	VARCHAR(18)		
TBAUTH	SMALLINT	Yes	If BTYPE=T or V, encodes the privileges on the table or view that are required by this trigger; otherwise null.

## SYSCAT.TRIGGERS

---

### SYSCAT.TRIGGERS

Contains one row for each trigger.

Table 69. SYSCAT.TRIGGERS Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	CHAR(8)		Qualified name of the trigger.
TRIGNAME	VARCHAR(18)		
DEFINER	CHAR(8)		Authorization ID under which the trigger was defined.
TABSCHEMA	CHAR(8)		Qualified name of the table to which this trigger applies.
TABNAME	VARCHAR(18)		
TRIGTIME	CHAR(1)		Time when triggered actions are applied to the base table, relative to the event that fired the trigger: B=Trigger applied before event A=Trigger applied after event
TRIGEVENT	CHAR(1)		Event that fires the trigger. I=Insert D=Delete U=Update
GRANULARITY	CHAR(1)		Trigger is executed once per: S=Statement R=Row
VALID	CHAR(1)		Y=Trigger is valid X=Trigger is inoperative; must be re-created.
TEXT	CLOB(32K)		The full text of the CREATE TRIGGER statement, exactly as typed.
CREATE_TIME	TIMESTAMP		Time at which the trigger was defined. Used in resolving functions and types.
FUNC_PATH	VARCHAR(254)		Function path at the time the trigger was defined. Used in resolving functions and types.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

**SYSCAT.VIEWDEP**

Contains a row for every dependency of a view or a summary table on some other object. Also encodes how privileges on this view depend on privileges on underlying tables and views.

Table 70. SYSCAT.VIEWDEP Catalog View

Column Name	Data Type	Nullable	Description
VIEWSCHEMA	CHAR(8)		Name of the view or the name of a summary table having dependencies on a base table.
VIEWNAME	VARCHAR(18)		
DEFINER	CHAR(8)	Yes	Authorization ID of the creator of the view.
BTYPE	CHAR(1)		Type of object BNAME: <ul style="list-style-type: none"> <li>• A=Alias</li> <li>• F=Function instance</li> <li>• H=Table or view hierarchy</li> <li>• I=Index if recording dependency on a base table</li> <li>• R=Structured type</li> <li>• S=Summary table</li> <li>• T=Table</li> <li>• V=View</li> </ul>
BSCHEMA	CHAR(8)		Qualified name of object depended on by the view.
BNAME	VARCHAR(18)		
TABAUTH	SMALLINT	Yes	Encodes the privileges on the underlying table or view that this view depends on. Otherwise null.

## SYSCAT.VIEWS

---

### SYSCAT.VIEWS

Contains one or more rows for each view that is created.

*Table 71. SYSCAT.VIEWS Catalog View*

Column Name	Data Type	Nullable	Description
VIEWSHEMA	CHAR(8)		Name of the view or the name of a table used to define a summary table.
VIEWNAME	VARCHAR(18)		
DEFINER	CHAR(8)		Authorization ID of the creator of the view.
SEQNO	SMALLINT		Sequence number of this row; the first portion of the view is on row one, and successive rows have increasing values of SEQNO.
VIEWCHECK	CHAR(1)		States the type of view checking: N=No check option L=Local check option C=Cascaded check option
READONLY	CHAR(1)		Y=View is read-only because of its definition. N=View is not read-only.
VALID	CHAR(1)		Y=View or summary table definition is valid. X=View or summary table definition is inoperative; must be re-created.
FUNC_PATH	VARCHAR(254)		The SQL path of the view creator at the time the view was defined. When the view is used in data manipulation statements, this path must be used to resolve function calls in the view. SYSIBM for views created before Version 2.
TEXT	VARCHAR(3600)		Text or portion of the text of the CREATE VIEW statement.

**SYSSTAT.COLDIST**

Each row describes the Nth-most-frequent value or Nth quantile value of some column.

Table 72. SYSSTAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Description	Updatable
TABSCHEMA	CHAR(8)		Qualified name of the table to which this entry applies.	
TABNAME	VARCHAR(18)			
COLNAME	VARCHAR(18)		Name of the column to which this entry applies.	
TYPE	CHAR(1)		Type of statistic collected: F=Frequency (most frequent value) Q=Quantile value	
SEQNO	SMALLINT		If TYPE=F, then N in this column identifies the Nth most frequent value. If TYPE=Q, then N in this column identifies the Nth quantile value.	
COLVALUE	VARCHAR(33)	Yes	The data value, as a character literal or a null value.  This column can be updated with a valid representation of the value appropriate to the column that the statistic is associated with. If null is the required frequency value, the column should be set to NULL.	Yes
VALCOUNT	INTEGER		If TYPE=F, then VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE=Q, then VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.  This column can be only updated with the following values: • >= 0 (zero)	Yes
DISTCOUNT	INTEGER		If TYPE=q, this column records the number of distinct values that are less than or equal to COLVALUE (null is unavailable.) the number of rows whose value is less than or equal to COLVALUE.	Yes

## SYSSTAT.COLUMNS

---

### SYSSTAT.COLUMNS

Contains one row for each column for which statistics can be updated.

**Note:** Summary tables are not included in this view for Version 5.2. If updating statistics for a summary table, use the OBJSTAT.COLUMNS view (see Appendix E, “Catalog Views For Use With Structured Types” on page 831 for how to define this view).

Table 73. SYSSTAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description	Updatable
TABSCHEMA	CHAR(8)		Qualified name of the table that contains the column.	
TABNAME	VARCHAR(18)			
COLNAME	VARCHAR(18)		Column name.	
COLCARD	INTEGER		Number of distinct values in the column; -1 if statistics are not gathered.  For any column, COLCARD cannot have a value higher than the cardinality of the table containing that column.  This column can only be updated with the following values: <ul style="list-style-type: none"><li>-1 or &gt;= 0 (zero)</li></ul>	Yes
HIGH2KEY	VARCHAR(33)		Second highest value of the column. This field is empty if statistics are not gathered.  This column can be updated with a valid representation of the value appropriate to the column that the statistic is associated with.  LOWKEY2 should not be greater than HIGH2KEY.	Yes
LOW2KEY	VARCHAR(33)		Second lowest value of the column. Empty if statistics not gathered.  This column can be updated with a valid representation of the value appropriate to the column that the statistic is associated with.	Yes
AVGCOLLEN	INTEGER		Average column length. -1 if a long field or LOB, or statistics have not been collected.  This column can only be updated with the following values: <ul style="list-style-type: none"><li>-1 or &gt;= 0 (zero)</li></ul>	Yes

**SYSSTAT.FUNCTIONS**

Contains a row for each user-defined function (scalar or aggregate). Does not include built-in functions.

*Table 74 (Page 1 of 2). SYSSTAT.FUNCTIONS Catalog View*

Column Name	Data Type	Nullable	Description	Updatable
FUNCSHEMA	CHAR(8)		Qualified function name.	
FUNCNAME	VARCHAR(18)			
SPECIFICNAME	VARCHAR(18)		Function specific (instance) name.	
IOS_PER_INVOC	DOUBLE		Estimated number of I/Os per invocation; -1 if not known (0 default).  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
INSTS_PER_INVOC	DOUBLE		Estimated number of instructions per invocation; -1 if not known (450 default).  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
IOS_PER_ARGBYTE	DOUBLE		Estimated number of I/O's per input argument byte; -1 if not known (0 default).  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
INSTS_PER_ARGBYTE	DOUBLE		Estimated number of instructions per input argument byte; -1 if not known (0 default).  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
PERCENT_ARGBYTES	SMALLINT		Estimated average percent of input argument bytes that the function will actually read; -1 if not known (100 default).  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or between 100 and 0 (zero)</li> </ul>	Yes
INITIAL_IOS	DOUBLE		Estimated number of I/O's performed the first/last time the function is invoked; -1 if not known (0 default).  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes

## SYSSTAT.FUNCTIONS

Table 74 (Page 2 of 2). SYSSTAT.FUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description	Updatable
INITIAL_INSTS	DOUBLE		Estimated number of instructions executed the first/last time the function is invoked; -1 if not known (0 default).  This column can only be updated with the following values: <ul style="list-style-type: none"><li>• -1 or <math>\geq 0</math> (zero)</li></ul>	Yes
CARDINALITY	INTEGER		The predicted cardinality of a table function. -1 if not known, or if function is not a table function.	Yes



---

**SYSSTAT.INDEXES**

Contains one row for each index that is defined for a table.

Table 75 (Page 1 of 3). SYSSTAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
INDSCHEMA	CHAR(8)		Qualified name of the index.	
INDNAME	VARCHAR(18)			
NLEAF	INTEGER		Number of leaf pages; -1 if statistics are not gathered.  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt; 0 (zero)</li> </ul>	Yes
NLEVELS	SMALLINT		Number of index levels; -1 if statistics are not gathered.  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt; 0 (zero)</li> </ul>	Yes
FIRSTKEYCARD	INTEGER		Number of distinct first key values; -1 if statistics are not gathered.  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
FIRST2KEYCARD	INTEGER		Number of distinct keys using the first two columns of the index (-1 if no statistics or inapplicable)  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
FIRST3KEYCARD	INTEGER		Number of distinct keys using the first three columns of the index (-1 if no statistics or inapplicable)  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
FIRST4KEYCARD	INTEGER		Number of distinct keys using the first four columns of the index (-1 if no statistics or inapplicable)  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes

## SYSSTAT.INDEXES

Table 75 (Page 2 of 3). SYSSTAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
FULLKEYCARD	INTEGER		<p>Number of distinct full key values; -1 if statistics are not gathered.</p> <p>This column can only be updated with the following values:</p> <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
CLUSTERRATIO	SMALLINT		<p>This is used by the optimizer. It indicates the degree of data clustering with the index; -1 if statistics are not gathered or if detailed index statistics have been gathered.</p> <p>This column can only be updated with the following values:</p> <ul style="list-style-type: none"> <li>-1 or between 0 and 100</li> </ul>	Yes
CLUSTERFACTOR	DOUBLE		<p>This is used by the optimizer. It is a finer measurement of degree of clustering, or -1 if detailed index statistics have not been gathered.</p> <p>This column can only be updated with the following values:</p> <ul style="list-style-type: none"> <li>-1 or between 0 and 1</li> </ul>	Yes
SEQUENTIAL_PAGES	INTEGER		<p>Number of leaf pages located on disk in index key order with few or no large gaps between them. (-1 if no statistics are available.)</p> <p>This column can only be updated with the following values:</p> <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
DENSITY	INTEGER		<p>Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100, -1 if no statistics are available.)</p> <p>This column can only be updated with the following values:</p> <ul style="list-style-type: none"> <li>-1 or between 0 and 100</li> </ul>	Yes

## SYSSTAT.INDEXES

Table 75 (Page 3 of 3). SYSSTAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
PAGE_FETCH_PAIRS	VARCHAR(254)		<p>A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the index using that hypothetical buffer. (Zero-length string if no data available.)</p> <p>This column can be updated with the following input values:</p> <ul style="list-style-type: none"><li>• The pair delimiter and pair separator characters are the only non-numeric characters accepted</li><li>• Blanks are the only characters recognized as a pair delimiter and pair separator</li><li>• Each number entry must have an accompanying partner number entry with the two being separated by the pair separator character</li><li>• Each pair must be separated from any other pairs by the pair delimiter character</li><li>• Each expected number entry must be between 0-9 (only positive values)</li></ul>	Yes

## SYSSTAT.TABLES

---

### SYSSTAT.TABLES

Contains one row for each *base* table. Views or aliases are, therefore, not included. For typed tables, only the root table of a table hierarchy is included in this view. The CARD value applies to the root table only while the other statistics apply to the entire table hierarchy.

**Note:** Summary tables are not included in this view for Version 5.2. If updating statistics for a summary table, use the OBJSTAT.TABLES view (see Appendix E, “Catalog Views For Use With Structured Types” on page 831 for how to define this view).

Table 76. SYSSTAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
TABSCHEMA	CHAR(8)		Qualified name of the table.	
TABNAME	VARCHAR(18)			
CARD	INTEGER		Total number of rows in the table; -1 if statistics are not gathered.  An update to CARD for a table should not attempt to assign it a value less than the COLCARD value of any of the columns in that table.  This column can only be updated with the following values: <ul style="list-style-type: none"><li>-1 or &gt;= 0 (zero)</li></ul>	Yes
NPAGES	INTEGER		Total number of pages on which the rows of the table exist; -1 if statistics are not gathered.  This column can only be updated with the following values: <ul style="list-style-type: none"><li>-1 or &gt;= 0 (zero)</li></ul>	Yes
FPAGES	INTEGER		Total number of pages in the file; -1 if statistics are not gathered.  This column can only be updated with the following values: <ul style="list-style-type: none"><li>-1 or &gt;= 0 (zero)</li></ul>	Yes
OVERFLOW	INTEGER		Total number of overflow records in the table; -1 if statistics are not gathered.  This column can only be updated with the following values: <ul style="list-style-type: none"><li>-1 or &gt;= 0 (zero)</li></ul>	Yes

---

## Appendix E. Catalog Views For Use With Structured Types

When using structured types, typed tables and typed views, another set of views of the catalog provide more useful information than the SYSCAT and SYSSTAT catalog views. These views are not created automatically. The views are created in the OBJCAT and OBJSTAT schemas and SELECT privilege on all views is granted to PUBLIC by default.

**WARNING:**

This set of views is for temporary use only until the next version that supports catalog migration. Applications should not presume that these views exist in every database and should consider that these catalog views may not be provided in future versions. The information from these views will be supported through the SYSCAT and SYSSTAT views in a future version.

The views can be created by following these steps.

- Using the Command Line Processor, connect to the database with an authorization ID that has SYSADM or DBADM authority.
- Ensure that you are in the home directory of the DB2 instance.
- In a UNIX-based system, issue the command:

```
db2 -tvf sql1lib/samples/clp/objcat.clp
```

In an OS/2 or Windows based system, issue the command:

```
db2 -tvf sql1lib\samples\clp\objcat.clp
```

**Note:** If the database already includes schemas called OBJCAT and/or OBJSTAT, you may need to make your own copy of the file objcat.clp and change the schema names in the second and third CREATE SCHEMA statements to suitable names.

The statements in the OBJCAT.CLP file will create all OBJCAT and OBJSTAT catalog views. There is an OBJCAT catalog view corresponding to each SYSCAT catalog view, and one OBJSTAT updatable catalog view corresponding to each SYSSTAT updatable catalog view.

This appendix contains a description of the OBJCAT and OBJSTAT catalog views for which the column definitions and/or acceptable values differ from their corresponding SYSCAT and SYSSTAT views. For views that have identical definitions as their SYSCAT and SYSSTAT counterparts, the reader is referred to those for column definitions.

The catalog views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the catalog views is available through normal SQL query facilities. Columns have consistent names based on the type of objects that they describe:

## Catalog Views for Structured Types

Described Object	Column Names
Table	TABSCHEMA, TABNAME
Index	INDSCHEMA, INDNAME
View	VIEWSCHEMA, VIEWNAME
Constraint	CONSTSCHEMA, CONSTNAME
Trigger	TRIGSCHEMA, TRIGNAME
Package	PKGSCHEMA, PKGNAME
Type	TYPESCHEMA, TYPENAME, TYPEID
Function	FUNCSCHEMA, FUNCNAME, FUNCID
Column	COLNAME
Attribute	ATTR_NAME
Schema	SCHEMANAME
Table Space	TBSPACE
Nodegroup	NGNAME
Buffer pool	BPNAME
Event Monitor	EVMONNAME
Creation Timestamp	CREATE_TIME

---

### Updatable Catalog Views For Use With Structured Types

The catalog views created in the OBJSTAT schema are updatable catalog views that correspond to the SYSTAT updatable catalog views.

The OBJSTAT views are used in the same way as the SYSSTAT views.

Before changing any statistics for the first time, it is advised to issue the RUNSTATS command so that all statistics will reflect the current state.

---

### “Roadmap” to Catalog Views for Structured Types

Description	Catalog View	Page
attributes of structured data types	OBJCAT.ATTRIBUTES	835
authorities on database	OBJCAT.DBAUTH <sup>92</sup>	786
Buffer pool configuration on nodegroup	OBJCAT.BUFFERPOOLS <sup>92</sup>	776
Buffer pool size on node	OBJCAT.BUFFERPOOLSNODE <sup>92</sup>	777
check constraints	OBJCAT.CHECKS	836
column privileges	OBJCAT.COLAUTH <sup>92</sup>	779
columns	OBJCAT.COLUMNS	838
columns referenced by check constraints	OBJCAT.COLCHECKS	837
columns used in keys	OBJCAT.KEYCOLUSE	851
constraint dependencies	OBJCAT.CONSTDEP	841
datatypes	OBJCAT.DATATYPES	842
event monitor definitions	OBJCAT.EVENTMONITORS <sup>92</sup>	787

## Catalog Views (structured types)

Description	Catalog View	Page
events currently monitored	OBJCAT.EVENTS <sup>92</sup>	788
function parameters	OBJCAT.FUNCPARMS	843
hierarchies (types, tables, views)	OBJCAT.HIERARCHIES	847
indexes	OBJCAT.INDEXES	848
detailed column statistics	OBJCAT.COLDIST <sup>92</sup>	781
nodegroup definitions	OBJCAT.NODEGROUPS <sup>92</sup>	799
nodegroup nodes	OBJCAT.NODEGROUPDEF <sup>92</sup>	798
partitioning maps	OBJCAT.PARTITIONMAPS <sup>92</sup>	805
package dependencies	OBJCAT.PACKAGEDEP	852
package privileges	OBJCAT.PACKAGEAUTH <sup>92</sup>	800
packages	OBJCAT.PACKAGES <sup>92</sup>	802
stored procedures	OBJCAT.PROCEDURES <sup>92</sup>	806
procedure parameters	OBJCAT.PROCPARMS <sup>92</sup>	807
referential constraints	OBJCAT.REFERENCES	853
schema privileges	OBJCAT.SCHEMAAUTH <sup>92</sup>	809
schemas	OBJCAT.SCHEMATA <sup>92</sup>	810
statements in packages	OBJCAT.STATEMENTS <sup>92</sup>	811
table constraints	OBJCAT.TABCONST	854
table privileges	OBJCAT.TABAUTH <sup>92</sup>	812
tables	OBJCAT.TABLES	855
table spaces	OBJCAT.TABLESPACES <sup>92</sup>	818
trigger dependencies	OBJCAT.TRIGDEP	858
triggers	OBJCAT.TRIGGERS	859
user-defined functions	OBJCAT.FUNCTIONS	844
view dependencies	OBJCAT.VIEWDEP	860
views	OBJCAT.TABLES	855
	OBJCAT.VIEWS <sup>92</sup>	822

### “Roadmap” to Updatable Catalog Views For Structured Types

Description	Catalog View	Page
columns	OBJSTAT.COLUMNS <sup>93</sup>	824
indexes	OBJSTAT.INDEXES <sup>93</sup>	827

<sup>92</sup> The column definition of these catalog views is identical to their SYSCAT counterparts of the same name. The page reference points to the SYSCAT catalog views for the column definitions.

## Catalog Views (structured types)

	<b>Description</b>	<b>Catalog View</b>	<b>Page</b>
	detailed column statistics	OBJSTAT.COLDIST <sup>93</sup>	823
	tables	OBJSTAT.TABLES	861
	user-defined functions	OBJSTAT.FUNCTIONS <sup>93</sup>	825

---

<sup>93</sup> The column definition of these catalog views is identical to their SYSSTAT counterparts of the same name. The page reference points to the SYSSTAT catalog views for the column definitions.



**OBJCAT.ATTRIBUTES**

Contains one row for each attribute (including inherited attributes where applicable) that is defined for a user-defined structured data type.

*Table 77. OBJCAT.ATTRIBUTES Catalog View*

Column Name	Data Type	Nullable	Description
TYPESHEMA	CHAR(8)		Qualified name of the structured data type that includes the attribute.
TYPENAME	VARCHAR(18)		
ATTR_NAME	VARCHAR(18)		Attribute name.
ATTR_TYPESHEMA	CHAR(8)		Contains the qualified name of the type of the attribute.
ATTR_TYPENAME	VARCHAR(18)		
TARGET_TYPESHEMA	CHAR(8)		Qualified name of the target type, if the type of the attribute is REFERENCE. Null value if the type of the attribute is not REFERENCE.
TARGET_TYPENAME	VARCHAR(18)		
ORIGIN_TYPESHEMA	CHAR(8)		Qualified name of the data type in the data type hierarchy where the attribute was introduced.
ORIGIN_TYPENAME	VARCHAR(18)		
POSITION	SMALLINT		Position of the attribute in the definition of the structured data type starting with zero.
LENGTH	INTEGER		Maximum length of data. 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields.
SCALE	SMALLINT		Scale for DECIMAL fields; 0 if not DECIMAL.
CODEPAGE	SMALLINT		Code page of the attribute. For character-string attributes not defined with FOR BIT DATA, the value is the database code page. For graphic-string attributes, the value is the DBCS code page implied by the (composite) database code page. Otherwise, the value is 0.
LOGGED	CHAR(1)		Applies only to attributes whose type is LOB or distinct based on LOB (blank otherwise). Y=Attribute is logged. N=Attribute is not logged.
COMPACT	CHAR(1)		Applies only to attributes whose type is LOB or distinct based on LOB (blank otherwise). Y=Attribute is compacted in storage. N=Attribute is not compacted.
DL_FEATURES	CHAR(10)		Applies to DATALINK type attributes only. Blank for REFERENCE type attributes. Null otherwise. Encodes various DATALINK features such as linktype, control mode, recovery, and unlink properties.

## OBJCAT.CHECKS

---

### OBJCAT.CHECKS

Contains one row for each CHECK constraint defined for a table.

*Table 78. OBJCAT.CHECKS Catalog View*

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the check constraint (unique within a table.)
DEFINER	CHAR(8)		Authorization ID under which the check constraint was defined.
TABSCHEMA	CHAR(8)		Qualified name of the table to which this constraint applies.
TABNAME	VARCHAR(18)		
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table on which the constraint was defined.
ORIGIN_TABNAME	VARCHAR(18)		
CREATE_TIME	TIMESTAMP		The time at which the constraint was defined. Used in resolving functions that are used in this constraint. No functions will be chosen that were created after the definition of the constraint.
FUNC_PATH	VARCHAR(254)		The current SQL path that was used when the constraint was created.
TEXT	CLOB(32K)		The text of the CHECK clause.

## OBJCAT.COLCHECKS

### OBJCAT.COLCHECKS

Each row represents some column that is referenced by a CHECK constraint defined for a table.

Table 79. OBJCAT.COLCHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the check constraint. (Unique within a table. May be system generated.)
TABSCHEMA	CHAR(8)		Qualified name of table containing referenced column.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Name of column.
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table on which the constraint was defined.
ORIGIN_TABNAME	VARCHAR(18)		

## OBJCAT.COLUMNS

### OBJCAT.COLUMNS

Contains one row for each column (including inherited columns where applicable) that is defined for a table or view. All of the catalog views have entries in the OBJCAT.COLUMNS table.

Table 80 (Page 1 of 3). OBJCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
TABSHEMA	CHAR(8)		Qualified name of the table or view that contains the column.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Column name.
COLNO	SMALLINT		Numerical place of column in table or view, beginning at zero.
TYPESHEMA	CHAR(8)		Contains the qualified name of the type, if the data type of the column is distinct. Otherwise
TYPENAME	VARCHAR(18)		TYPESHEMA contains the value SYSIBM and TYPENAME contains the data type of the column (in long form, for example, CHARACTER). If FLOAT or FLOAT( <i>n</i> ) with <i>n</i> greater than 24 is specified, TYPENAME is renamed to DOUBLE. If FLOAT( <i>n</i> ) with <i>n</i> less than 25 is specified, TYPENAME is renamed to REAL. Also, NUMERIC is renamed to DECIMAL.
LENGTH	INTEGER		Maximum length of data. 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields.
SCALE	SMALLINT		Scale for DECIMAL fields; 0 if not DECIMAL.
DEFAULT	VARCHAR(254)	Yes	Default value for the column of a table expressed as a constant, special register, or cast-function appropriate for the data type of the column. May also be the keyword NULL.  Values may be converted from what was specified as a default value. For example, date and time constants are presented in ISO format and cast-function names are qualified with schema name and the identifiers are delimited (see Note 3).  Null value if a DEFAULT clause was not specified or the column is a view column.
NULLS	CHAR(1)		Y=Column is nullable. N=Column is not nullable.  The value can be N for a view column that is derived from an expression or function. Nevertheless, such a column allows nulls when the statement using the view is processed with warnings for arithmetic errors.  See Note 1.

## OBJCAT.COLUMNS

Table 80 (Page 2 of 3). OBJCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
CODEPAGE	SMALLINT		Code page of the column. For character-string columns not defined with the FOR BIT DATA attribute, the value is the database code page. For graphic-string columns, the value is the DBCS code page implied by the (composite) database code page. Otherwise, the value is 0.
LOGGED	CHAR(1)		Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise).  Y=Column is logged. N=Column is not logged.
COMPACT	CHAR(1)		Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise).  Y=Column is compacted in storage. N=Column is not compacted.
COLCARD	INTEGER		Number of distinct values in the column; -1 if statistics are not gathered; -2 for an inherited column of a subtable.
HIGH2KEY	VARCHAR(33)		Second highest value of the column. This field is empty if statistics are not gathered. See Note 2.
LOW2KEY	VARCHAR(33)		Second lowest value of the column. Empty if statistics not gathered. See Note 2.
AVGCOLLEN	INTEGER		Average column length. -1 if a long field or LOB, or statistics have not been collected; -2 for an inherited column of a subtable.
KEYSEQ	SMALLINT	Yes	The column's numerical position within the table's primary key. This field is null or 0 if the column is not part of the primary key.
PARTKEYSEQ	SMALLINT	Yes	The column's numerical position within the table's partitioning key. This field is null or 0 if the column is not part of the partitioning key.
NQUANTILES	SMALLINT		Number of quantile values recorded in OBJCAT.SYSCOLDIST for this column; -1 if no statistics; -2 for an inherited column of a subtable.
NMOSTFREQ	SMALLINT		Number of most-frequent values recorded in OBJCAT.COLDIST for this column; -1 if statistics not gathered; -2 for an inherited column in a subtable.
TARGET_TYPESHEMA	CHAR(8)	Yes	Qualified name of the target type, if the type of the column is REFERENCE. Null value if the type of the column is not REFERENCE.
TARGET_TYPENAME	VARCHAR(18)	Yes	
SCOPE_TABSCHEMA	CHAR(8)	Yes	Qualified name of the scope (target table), if the type of the column is REFERENCE. Null value if the type of the column is not REFERENCE or the scope is not defined.
SCOPE_TABNAME	VARCHAR(18)	Yes	

## OBJCAT.COLUMNS

Table 80 (Page 3 of 3). OBJCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table or view in the respective hierarchy where the column was introduced.
ORIGIN_TABNAME	VARCHAR(18)		
DL_FEATURES	CHAR(10)	Yes	Applies to DATALINK type columns only. Null otherwise. Each character position is defined as follows: <ol style="list-style-type: none"><li>1. Link type (U for URL)</li><li>2. Link control (F for file, N for no)</li><li>3. Integrity (A for all, N for none)</li><li>4. Read permission (F for file system, D for database)</li><li>5. Write permission (F for file system, B for blocked)</li><li>6. Recovery (Y for yes, N for no)</li><li>7. On unlink (R for restore, D for delete, N for not applicable)</li></ol> Characters 8 through 10 are reserved for future use.
SPECIAL_PROPS	CHAR(8)	Yes	Applies to REFERENCE type columns only. Null otherwise. 'Y' in the first byte indicates an object identifier (OID) column ('N' otherwise). 'U' in the second byte indicates user generated reference values.
REMARKS	VARCHAR(254)	Yes	User-supplied comment.

**Note:**

1. Starting with Version 2, value D (indicating not null with a default) is no longer used. Instead, use of WITH DEFAULT is indicated by a non-null value in the DEFAULT column.
2. Starting with Version 2, representation of numeric data has been changed to character literals. The size has been enlarged from 16 to 33 bytes.
3. For Version 2.1.0, cast-function names were not delimited and may still appear this way in the DEFAULT column. Also, some view columns included default values which will still appear in the DEFAULT column.

## OBJCAT.CONSTDEP

### OBJCAT.CONSTDEP

Contains a row for every dependency of a constraint on some other object.

Table 81. OBJCAT.CONSTDEP Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint.
TABSCHEMA	CHAR(8)		Qualified name of the table to which the constraint applies.
TABNAME	VARCHAR(18)		
BTYPE	CHAR(1)		Type of object that the constraint depends on. Possible values: F=function instance. I=index instance. R=structured type
BSCHEMA	CHAR(8)		Qualified name of object that the constraint depends on.
BNAME	VARCHAR(18)		

## OBJCAT.DATATYPES

### OBJCAT.DATATYPES

Contains a row for every data type, including built-in and user-defined types.

Table 82. OBJCAT.DATATYPES Catalog View

Column Name	Data Type	Nullable	Description
TYPESHEMA	CHAR(8)		Qualified name of the data type (for built-in types, TYPESHEMA is SYSIBM).
TYPENAME	VARCHAR(18)		
DEFINER	CHAR(8)		Authorization ID under which type was created.
SOURCESHEMA	CHAR(8)	Yes	Qualified name of the source type for distinct types.
SOURCENAME	VARCHAR(18)	Yes	Qualified name of the builtin type used as the reference type that is used as the representation for references to structured types. Null for other types.
METATYPE	CHAR(1)		R=user-defined structured type S=System predefined type T=user-defined distinct type
TYPEID	SMALLINT		Internal type ID.
SOURCETYPEID	SMALLINT	Yes	Internal type ID of source type (null for built-in types). For user-defined structured types, this is the internal type ID of the reference representation type.
LENGTH	INTEGER		Maximum length of the type. 0 for system predefined parameterized types (for example, DECIMAL and VARCHAR). For user-defined structured types, this indicates the length of the reference representation type.
SCALE	SMALLINT		Scale for distinct types or reference representation types based on the system predefined DECIMAL type. 0 for all other types (including DECIMAL itself).
CODEPAGE	SMALLINT		Code page for character and graphic distinct types or reference representation types; 0 otherwise.
CREATE_TIME	TIMESTAMP		Creation time of the data type.
INSTANTIABLE	CHAR(1)		'Y' to indicate type can be instantiated.
INLINE_LENGTH	INTEGER		Length of structured type that can be kept with base table row. Always 0.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.



**OBJCAT.FUNCPARMS**

Contains a row for every parameter or result of a function defined in OBJCAT.FUNCTIONS.

Table 83. OBJCAT.FUNCPARMS Catalog View

Column Name	Data Type	Nullable	Description
FUNCSHEMA	CHAR(8)		Qualified function name.
FUNCNAME	VARCHAR(18)		
SPECIFICNAME	VARCHAR(18)		The name of the function instance (may be system-generated).
ROWTYPE	CHAR(1)		P=parameter R=result before casting C=result after casting
ORDINAL	SMALLINT		If ROWTYPE=P, the parameter's numerical position within the function signature. Otherwise 0.
PARAMNAME	VARCHAR(18)		Name of parameter or result column, or null if no name exists.
TYPESHEMA	CHAR(8)		Qualified name of data type of parameter or result.
TYPENAME	VARCHAR(18)		
LENGTH	INTEGER		Length of parameter or result. 0 if parameter or result is a distinct type. See Note 1.
SCALE	SMALLINT		Scale of parameter or result. 0 if parameter or result is a distinct type. See Note 1.
CODEPAGE	SMALLINT		Code page of parameter. 0 denotes either not applicable or a column for character data declared with the FOR BIT DATA attribute.
TARGET_TYPESHEMA	CHAR(8)		Qualified name of the target type, if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE.
TARGET_TYPENAME	VARCHAR(18)		
SCOPE_TABSCHEMA	CHAR(8)		Qualified name of the scope (target table), if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE or the scope is not defined.
SCOPE_TABNAME	VARCHAR(18)		
CAST_FUNCID	INTEGER	Yes	Internal function ID.
AS_LOCATOR	CHAR(1)		Y=Parameter or result is passed in the form of a locator N=Not passed in the form of a locator.

**Note:**

1. LENGTH and SCALE are set to 0 for sourced functions (functions defined with a reference to another function) because they inherit the length and scale of parameters from their source.

## OBJCAT.FUNCTIONS

### OBJCAT.FUNCTIONS

Contains a row for each user-defined function (scalar, table or sourced). Does not include built-in functions.

Table 84 (Page 1 of 3). OBJCAT.FUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCSHEMA	CHAR(8)		Qualified function name.
FUNCNAME	VARCHAR(18)		
SPECIFICNAME	VARCHAR(18)		The name of the function instance (may be system-generated).
DEFINER	CHAR(8)		Authorization ID of function definer.
FUNCID	INTEGER		Internally-assigned function ID.
RETURN_TYPE	SMALLINT		Internal type code of return type of function.
ORIGIN	CHAR(1)		B=Built-in E=User-defined, external U=User-defined, based on a source S=System-generated
TYPE	CHAR(1)		S=Scalar function C=Column function T=Table function
METHOD	CHAR(1)		N=Not a method Y=Method
EFFECT	CHAR(2)		MU=mutator method OB=observer method CN=constructor method blanks=not a method
PARAM_COUNT	SMALLINT		Number of function parameters.
PARAM_SIGNATURE	VARCHAR(180) FOR BIT DATA		Concatenation of up to 90 parameter types, in internal format. Zero length if function takes no parameters.
CREATE_TIME	TIMESTAMP		Timestamp of function creation. Set to 0 for Version 1 functions.
FUNC_PATH	VARCHAR(254)	Yes	Function path at the time the function was defined.
TYPE_PRESERVING	CHAR(1)		Always blank.
VARIANT	CHAR(1)		Y=Variant (results may differ) N=Invariant (results are consistent) Blank if ORIGIN is not E
SIDE_EFFECTS	CHAR(1)		E=Function has external side-effects (number of invocations is important) N=No side-effects Blank if ORIGIN is not E
FENCED	CHAR(1)		Y=Fenced N=Not fenced Blank if ORIGIN is not E

## OBJCAT.FUNCTIONS

Table 84 (Page 2 of 3). OBJCAT.FUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
NULLCALL	CHAR(1)		Y=Nullcall N=No nullcall (function result is implicitly null if operand(s) are null). Blank if ORIGIN is not E.
CAST_FUNCTION	CHAR(1)		Y=This is a cast function N=This is not a cast function
ASSIGN_FUNCTION	CHAR(1)		Y=Implicit assignment function N=Not an assignment function
SCRATCHPAD	CHAR(1)		Y=This function has a scratch pad N=This function does not have a scratch pad Blank if ORIGIN is not E
FINAL_CALL	CHAR(1)		Y=Final call is made to this function at run time end-of-statement. N=No final call is made. Blank if ORIGIN is not E
PARALLELIZABLE	CHAR(1)		Y=Function can be executed in parallel N=Function cannot be executed in parallel Blank if ORIGIN is not E
CONTAINS_SQL	CHAR(1)		Indicates wheter an external function contains SQL. N=Function does not contain SQL statements. R=Contains read-only SQL statements. M=Contains SQL statements that modify data. Blank if ORIGIN is not E
DBINFO	CHAR(1)		Indicates whether a DBINFO parameter is passed to an external function.  Y=DBINFO is passed. N=DBINFO is not passed. Blank if ORIGIN is not E
RESULT_COLS	SMALLINT		For a table function (TYPE=T) contains the number of columns in the result table; otherwise contains 1.
LANGUAGE	CHAR(8)		Implementation language of function body. Possible values are C, JAVA or OLE. Blank if ORIGIN is not E.
IMPLEMENTATION	VARCHAR(254)	Yes	If ORIGIN=E, identifies the path/module/function that implements this function. If ORIGIN=U and the source function is built-in, this column contains the name and signature of the source function. Null otherwise.
PARAM_STYLE	CHAR(8)		Indicates the parameter style declared in the CREATE FUNCTION statement. Values:  DB2SQL DB2GENRL

## OBJCAT.FUNCTIONS

Table 84 (Page 3 of 3). OBJCAT.FUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
SOURCE_SCHEMA	CHAR(8)	Yes	If ORIGIN=U and the source function is a user-defined function, contains the qualified name of the source function. If ORIGIN=U and the source function is built-in, SOURCE_SCHEMA is 'SYSIBM' and SOURCE_SPECIFIC is 'N/A for built-in'. Null if ORIGIN is not U.
SOURCE_SPECIFIC	VARCHAR(18)	Yes	
IOS_PER_INVOC	DOUBLE		Estimated number of I/Os per invocation; -1 if not known (0 default).
INSTS_PER_INVOC	DOUBLE		Estimated number of instructions per invocation; -1 if not known (450 default).
IOS_PER_ARGBYTE	DOUBLE		Estimated number of I/O's per input argument byte; -1 if not known (0 default).
INSTS_PER_ARGBYTE	DOUBLE		Estimated number of instructions per input argument byte; -1 if not known (0 default).
PERCENT_ARGBYTES	SMALLINT		Estimated average percent of input argument bytes that the function will actually read; -1 if not known (100 default).
INITIAL_IOS	DOUBLE		Estimated number of I/O's performed the first/last time the function is invoked; -1 if not known (0 default).
INITIAL_INSTS	DOUBLE		Estimated number of instructions executed the first/last time the function is invoked; -1 if not known (0 default).
CARDINALITY	INTEGER	Yes	The predicted cardinality of a table function. -1 if not known or if function is not a table function.
BODY	CLOB(1M)	Yes	Always null (future use).
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

## OBJCAT.HIERARCHIES

### OBJCAT.HIERARCHIES

Each row represents the immediate subtype/supertype, subtable/supertable, or subview/superview relationship in the database.

Table 85. OBJCAT.HIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR(1)		Indicates the type of database objects to which this row is applicable.  R=relationship between structured types T=relationship between tables V=relationship between views
SUB_SCHEMA	CHAR(8)		Qualified name of subtype, subtable, or subview.
SUB_NAME	VARCHAR(18)		
SUPER_SCHEMA	CHAR(8)		Qualified name of supertype, supertable, or superview.
SUPER_NAME	VARCHAR(18)		

## OBJCAT.INDEXES

### OBJCAT.INDEXES

Contains one row for each index (including inherited indexes where applicable) that is defined for a table.

Table 86 (Page 1 of 3). OBJCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	CHAR(8)		Name of the index.
INDNAME	VARCHAR(18)		
DEFINER	CHAR(8)		User who created the index.
TABSCHEMA	CHAR(8)		Qualified name of the table on which the index is defined.
TABNAME	VARCHAR(18)		
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table in the table hierarchy where the index was introduced.
ORIGIN_TABNAME	VARCHAR(18)		
COLNAMES	VARCHAR(320)		List of column names, each preceded by + or – to indicate ascending or descending order respectively.
UNIQUERULE	CHAR(1)		Unique rule: <ul style="list-style-type: none"><li>• D=duplicates allowed</li><li>• P=primary index.</li><li>• U=unique entries only allowed</li></ul>
MADE_UNIQUE	CHAR(1)		<ul style="list-style-type: none"><li>• Y=Index was originally non-unique but was converted to a unique index to support a unique or primary key constraint. If the constraint is dropped, the index will revert to non-unique.</li><li>• N=Index remains as it was created.</li></ul>
COLCOUNT	SMALLINT		Number of columns in the key plus the number of include columns if any.
UNIQUE_COLCOUNT	SMALLINT		The number of columns required for a unique key. Always <=COLCOUNT. < COLCOUNT only if there a include columns. –1 if index has no unique key (permits duplicates).
INDEXTYPE	CHAR(4)		Type of index. <ul style="list-style-type: none"><li>• CLUS =Clustering</li><li>• REG =Regular</li></ul>
PCTFREE	SMALLINT		Percentage of each index leaf page to be reserved during initial building of the index. This space is available for future inserts after the index is built.
IID	SMALLINT		Internal index ID.
NLEAF	INTEGER		Number of leaf pages. –1 if statistics are not gathered; –2 for an inherited index on a subtable.
NLEVELS	SMALLINT		Number of index levels. –1 if statistics are not gathered; –2 for an inherited index on a subtable.

## OBJCAT.INDEXES

Table 86 (Page 2 of 3). OBJCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
FIRSTKEYCARD	INTEGER		Number of distinct first key values. -1 if statistics are not gathered; -2 for an inherited index on a subtable.
FIRST2KEYCARD	INTEGER		Number of distinct keys using the first two columns of the index. -1 if no statistics or inapplicable; -2 for an inherited index on a subtable.
FIRST3KEYCARD	INTEGER		Number of distinct keys using the first three columns of the index. -1 if no statistics or inapplicable; -2 for an inherited index on a subtable.
FIRST4KEYCARD	INTEGER		Number of distinct keys using the first four columns of the index. -1 if no statistics or inapplicable; -2 for an inherited index on a subtable.
FULLKEYCARD	INTEGER		Number of distinct full key values. -1 if statistics are not gathered; -2 for an inherited index on a subtable.
CLUSTERRATIO	SMALLINT		Degree of data clustering with the index. -1 if statistics are not gathered or if detailed index statistics are gathered (in which case, CLUSTERFACTOR will be used instead); -2 for an inherited index on a subtable.
CLUSTERFACTOR	DOUBLE		Finer measurement of degree of clustering. -1 if detailed index statistics have not been gathered; -2 for an inherited index on a subtable.
SEQUENTIAL_PAGES	INTEGER		Number of leaf pages located on disk in index key order with few or no large gaps between them. -1 if no statistics are available; -2 for an inherited index on a subtable.
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100). -1 if no statistics are available; -2 for an inherited index on a subtable.
USER_DEFINED	SMALLINT		1 if this index was defined by a user and has not been dropped; otherwise 0.
SYSTEM_REQUIRED	SMALLINT		<ul style="list-style-type: none"> <li>• 1 if this index is required for primary key or unique key constraint, OR if this is the index on the object identifier (OID) column of a typed table.</li> <li>• 2 if this index is required for primary key or unique key constraint, AND this is the index on the object identifier (OID) column of a typed table.</li> <li>• 0 otherwise.</li> </ul>
CREATE_TIME	TIMESTAMP		Time when the index was created.
STATS_TIME	TIMESTAMP	Yes	Last time when any change was made to recorded statistics for this index. Null if no statistics available, or for an inherited index on a subtable.

## OBJCAT.INDEXES

Table 86 (Page 3 of 3). OBJCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
PAGE_FETCH_PAIRS	VARCHAR(254)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available, or for an inherited index on a subtable.)
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.
TEXT	CLOB(32K)	Yes	Reserved for future use.



**OBJCAT.KEYCOLUSE**

Lists all columns that participate in a key (including inherited primary or unique keys where applicable) defined by a unique, primary key or foreign key constraint.

*Table 87. OBJCAT.KEYCOLUSE Catalog View*

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint (unique within a table).
TABSCHEMA	CHAR(8)		Qualified name of the table containing the column.
TABNAME	VARCHAR(18)		
COLNAME	VARCHAR(18)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the key (initial position=1).
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table in the table hierarchy where the key was introduced.
ORIGIN_TABNAME	VARCHAR(18)		

## OBJCAT.PACKAGEDEP

### OBJCAT.PACKAGEDEP

Contains a row for each dependency that packages have on indexes, tables, views, functions, aliases, types, and hierarchies.

Table 88. OBJCAT.PACKAGEDEP Catalog View

Column Name	Data Type	Nullable	Description
PKGSHEMA	CHAR(8)		Name of the package.
PKGNAME	CHAR(8)		
BINDER	CHAR(8)	Yes	Binder of the package.
BTYPE	CHAR(1)		Type of object BNAME: <ul style="list-style-type: none"><li>• A=alias</li><li>• F=function-instance</li><li>• H=table or view hierarchy</li><li>• I=index</li><li>• R=structured type</li><li>• S=summary table</li><li>• T=table</li><li>• V=view</li></ul>
BSCHEMA	CHAR(8)		Qualified name of an object on which the package is dependent.
BNAME	VARCHAR(18)		
TBAUTH	SMALLINT	Yes	If BTYPE is T(table) or V(view), encodes the privileges that are required by this package (Select, Insert, Delete, Update).

**Note:**

1. When a depended-on function-instance is dropped, the package is placed into an “inoperative” state from which it must be explicitly rebound. When any other depended-on object is dropped, the package is placed into an “invalid” state from which the system will attempt to rebind it automatically when a package is first referenced.

**OBJCAT.REFERENCES**

Contains a row for each defined referential constraint.

Table 89. OBJCAT.REFERENCES Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of constraint.
TABSCHEMA	CHAR(8)		Qualified name of the constraint.
TABNAME	VARCHAR(18)		
DEFINER	CHAR(8)		User who created the constraint.
REFKEYNAME	VARCHAR(18)		Name of parent key.
REFTABSCHEMA	CHAR(8)		Name of the parent table.
REFTABNAME	VARCHAR(18)		
COLCOUNT	SMALLINT		Number of columns in the foreign key.
DELETERULE	CHAR(1)		Delete rule: A=NO ACTION C=CASCADE N=SET NULL R=RESTRICT
UPDATERULE	CHAR(1)		Update rule: A=NO ACTION R=RESTRICT
CREATE_TIME	TIMESTAMP		The timestamp when the referential constraint was defined.
FK_COLNAMES	VARCHAR(320)		List of foreign key column names.
PK_COLNAMES	VARCHAR(320)		List of parent key column names.
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table where the referential constraint was introduced.
ORIGIN_TABNAME	VARCHAR(18)		

**Note:**

1. The OBJCAT.REFERENCES view is based on the SYSIBM.SYSRELS table from Version 1.

## OBJCAT.TABCONST

---

### OBJCAT.TABCONST

Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY.

*Table 90. OBJCAT.TABCONST Catalog View*

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint (unique within a table).
TABSCHEMA	CHAR(8)		Qualified name of the table to which this constraint applies.
TABNAME	VARCHAR(18)		
DEFINER	CHAR(8)		Authorization ID under which the constraint was defined.
TYPE	CHAR(1)		Indicates the constraint type: K=CHECK P=PRIMARY KEY F=FOREIGN KEY U=UNIQUE
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table where the constraint was introduced.
ORIGIN_TABNAME	VARCHAR(18)		
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

**OBJCAT.TABLES**

Contains one row for each table, view, or alias that is created. All of the catalog tables and views have entries in the OBJCAT.TABLES catalog view.

Table 91 (Page 1 of 3). OBJCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	CHAR(8)		Qualified name of the table, view, or alias.
TABNAME	VARCHAR(18)		
DEFINER	CHAR(8)		User who created the table, view, or alias.
TYPE	CHAR(1)		The type of object: A=Alias S=Summary table T=Table V=View
STATUS	CHAR(1)		The type of object: N=Normal table, view or alias C=Check pending on table X=Inoperative view
BASE_TABSCHEMA	CHAR(8)	Yes	If TYPE=A, these columns identify the table, view, or alias that is referenced by this alias; otherwise they are null.
BASE_TABNAME	VARCHAR(18)	Yes	
ROWTYPESCHEMA	CHAR(8)	Yes	These columns identify the structured type used to define the table. Values are null if the table is not defined using a structured type.
ROWTYPENAME	VARCHAR(18)	Yes	
CREATE_TIME	TIMESTAMP		The timestamp indicating when the object was created.
STATS_TIME	TIMESTAMP	Yes	Last time when any change was made to recorded statistics for this table. Null if no statistics available.
COLCOUNT	SMALLINT		Number of columns in table.
TABLEID	SMALLINT		Internal table identifier.
TBSPACEID	SMALLINT		Internal identifier of primary table space for this table.
CARD	INTEGER		Total number of rows in the table; -1 if statistics are not gathered or the row describes a view or alias.
NPAGES	INTEGER		Total number of pages on which the rows of the table exist; -1 if statistics are not gathered or the row describes a view or alias; -2 for a subtable.
FPAGES	INTEGER		Total number of pages; -1 if statistics are not gathered or the row describes a view or alias; -2 for a subtable.
OVERFLOW	INTEGER		Total number of overflow records in the table; -1 if statistics are not gathered or the row describes a view or alias; -2 for a subtable.
TBSPACE	VARCHAR(18)	Yes	Name of primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. Null for aliases and views.

## OBJCAT.TABLES

Table 91 (Page 2 of 3). OBJCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
INDEX_TBSPACE	VARCHAR(18)	Yes	Name of table space that holds all indexes created on this table. Null for aliases and views, or if the INDEX IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
LONG_TBSPACE	VARCHAR(18)	Yes	Name of table space that holds all long data (LONG or LOB column types) for this table. Null for aliases and views, or if the LONG IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
PARENTS	SMALLINT	Yes	Number of parent tables of this table (the number of referential constraints in which this table is a dependent).
CHILDREN	SMALLINT	Yes	Number of dependent tables of this table (the number of referential constraints in which this table is a parent).
SELFREFS	SMALLINT	Yes	Number of self-referencing referential constraints for this table (the number of referential constraints in which this table is both a parent and a dependent).
KEYCOLUMNS	SMALLINT	Yes	Number of columns in the primary key of the table.
KEYINDEXID	SMALLINT	Yes	Index ID of the primary index. This field is null or 0 if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique constraints (other than primary key) defined on this table.
CHECKCOUNT	SMALLINT		Number of check constraints defined on this table.
DATA_CAPTURE	CHAR(1)		Y=Table participates in data replication N=Does not participate
CONST_CHECKED	CHAR(32)		Byte 1 represents foreign key constraints. Byte 2 represents check constraints. Byte 3 represents constraint Datalink_Reconcile_Pending. Byte 4 represents constraint Datalink_Reconcile_Not_Possible. Byte 5 represents summary table. Other bytes are reserved. Encodes constraint information on checking. Values:  Y=Checked by system U=Checked by user N=Not checked (pending)
PMAP_ID	SMALLINT	Yes	Identifier of the partitioning map used by this table. Null for aliases and views.
PARTITION_MODE	CHAR(1)		Mode used for tables in a partitioned database.  <b>H</b> hash on the partitioning key  <b>R</b> table replicated across database partitions  Blank for aliases, views and tables in single partition nodegroups with no partitioning key defined.

## OBJCAT.TABLES

Table 91 (Page 3 of 3). OBJCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
LOG_ATTRIBUTE	CHAR(1)		0=Default logging 1=Table created not logged initially
PCTFREE	SMALLINT		Percentage of each page to be reserved for future inserts. Can be changed by ALTER TABLE.
REMARKS	VARCHAR(254)	Yes	User-provided comment.

## OBJCAT.TRIGDEP

---

### OBJCAT.TRIGDEP

Contains a row for every dependency of a trigger on some other object.

Table 92. OBJCAT.TRIGDEP Catalog View

---

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	CHAR(8)		Qualified name of the trigger.
TRIGNAME	VARCHAR(18)		
BTYPE	CHAR(1)		Type of object BNAME: <ul style="list-style-type: none"><li>• A=Alias</li><li>• F=Function instance</li><li>• H=Table or view hierarchy</li><li>• R=Structured type</li><li>• S=Summary table</li><li>• T=Table</li><li>• V=View</li></ul>
BSCHEMA	CHAR(8)		Qualified name of object depended on by a trigger.
BNAME	VARCHAR(18)		
TABAUTH	SMALLINT	Yes	If BTYPE=T or V, encodes the privileges on the table or view that are required by this trigger; otherwise null.

---



## OBJCAT.TRIGGERS

### OBJCAT.TRIGGERS

Contains one row for each trigger.

Table 93. OBJCAT.TRIGGERS Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	CHAR(8)		Qualified name of the trigger.
TRIGNAME	VARCHAR(18)		
DEFINER	CHAR(8)		Authorization ID under which the trigger was defined.
TABSCHEMA	CHAR(8)		Qualified name of the table to which this trigger applies.
TABNAME	VARCHAR(18)		
TRIGTIME	CHAR(1)		Time when triggered actions are applied to the base table, relative to the event that fired the trigger: B=Trigger applied before event A=Trigger applied after event
TRIGEVENT	CHAR(1)		Event that fires the trigger. I=Insert D=Delete U=Update
GRANULARITY	CHAR(1)		Trigger is executed once per: S=Statement R=Row
VALID	CHAR(1)		Y=Trigger is valid X=Trigger is inoperative; must be re-created.
TEXT	CLOB(32K)		The full text of the CREATE TRIGGER statement, exactly as typed.
CREATE_TIME	TIMESTAMP		Time at which the trigger was defined. Used in resolving functions and types.
FUNC_PATH	VARCHAR(254)		Function path at the time the trigger was defined. Used in resolving functions and types.
ORIGIN_TABSCHEMA	CHAR(8)		Qualified name of the table where the trigger was introduced.
ORIGIN_TABNAME	VARCHAR(18)		
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

## OBJCAT.VIEWDEP

---

### OBJCAT.VIEWDEP

Contains a row for every dependency of a view or a summary table on some other object. Also encodes how privileges on this view depend on privileges on underlying tables and views.

*Table 94. OBJCAT.VIEWDEP Catalog View*

Column Name	Data Type	Nullable	Description
VIEWSCHEMA	CHAR(8)		Name of the view or the name of a summary table having dependencies on a base table.
VIEWNAME	VARCHAR(18)		
DEFINER	CHAR(8)	Yes	Authorization ID of the creator of the view.
BTYPE	CHAR(1)		Type of object BNAME: <ul style="list-style-type: none"><li>• A=Alias</li><li>• F=Function instance</li><li>• H=Table or view hierarchy</li><li>• I=Index if recording dependency on a base table</li><li>• R=Structured type</li><li>• S=Summary table</li><li>• T=Table</li><li>• V=View</li></ul>
BSCHEMA	CHAR(8)		Qualified name of object depended on by the view.
BNAME	VARCHAR(18)		
TABAUTH	SMALLINT	Yes	Encodes the privileges on the underlying table or view that this view depends on. Otherwise null.

**OBJSTAT.TABLES**

Contains one row for each table. Views or aliases are, therefore, not included.

Table 95. SYSSTAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
TABSCHEMA	CHAR(8)		Qualified name of the table.	
TABNAME	VARCHAR(18)			
CARD	INTEGER		Total number of rows in the table; -1 if statistics are not gathered.  An update to CARD for a table should not attempt to assign it a value less than the COLCARD value of any of the columns in that table.  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
NPAGES	INTEGER		Total number of pages on which the rows of the table exist; -1 if statistics are not gathered; -2 for a subtable. <sup>94</sup>  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
FPAGES	INTEGER		Total number of pages in the file; -1 if statistics are not gathered; -2 for a subtable. <sup>94</sup>  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes
OVERFLOW	INTEGER		Total number of overflow records in the table; -1 if statistics are not gathered; -2 for a subtable. <sup>94</sup>  This column can only be updated with the following values: <ul style="list-style-type: none"> <li>-1 or &gt;= 0 (zero)</li> </ul>	Yes

<sup>94</sup> A value of -2 indicates a subtable. A value of -2 is set by the database manager, and cannot be updated with another value. The value of this column cannot be updated with the value -2.

## OBJSTAT.TABLES

---

## Appendix F. Sample Tables

This appendix shows the information contained in the sample tables, and how to install and remove them. The sample tables are used in the examples that appear in this manual and other manuals in this library. In addition, the data contained in the sample files with BLOB and CLOB data types is shown.

The following sections are included in this appendix:

- “The Sample Database”
- “To Install the Sample Database”
- “To Erase the Sample Database” on page 864
- “CL\_SCHED Table” on page 864
- “DEPARTMENT Table” on page 864
- “EMPLOYEE Table” on page 865
- “EMP\_ACT Table” on page 867
- “EMP\_PHOTO Table” on page 869
- “EMP\_RESUME Table” on page 869
- “IN\_TRAY Table” on page 870
- “ORG Table” on page 870
- “PROJECT Table” on page 870
- “SALES Table” on page 871
- “STAFF Table” on page 872
- “STAFFG Table” on page 874
- “Sample Files with BLOB and CLOB Data Type” on page 875
- “Quintana Photo” on page 875
- “Quintana Resume” on page 875
- “Nicholls Photo” on page 876
- “Nicholls Resume” on page 876
- “Adamson Photo” on page 877
- “Adamson Resume” on page 878
- “Walker Photo” on page 879
- “Walker Resume” on page 879.

In the sample tables, a dash (-) indicates a null value.

---

### The Sample Database

The examples in this book use a sample database. To use these examples, you must install the SAMPLE database. To use it, the database manager must be installed.

### To Install the Sample Database

An executable file installs the sample database.<sup>95</sup> To install a database you must have SYSADM authority.

---

<sup>95</sup> For information related to this command, see the DB2SAMPL command in the *Command Reference*.

## Sample Tables

- **When Using UNIX-based Systems**

If you are using the operating system command prompt, type:

```
sql1lib/misc/db2samp1 <path>
```

from the home directory of the database manager instance owner, where *path* is an optional parameter specifying the path where the sample database is to be created. Press Enter.<sup>96</sup> The schema for DB2SAMPL is the CURRENT SCHEMA special register value.

- **When using OS/2, Windows 95 or Windows NT**

If you are using the operating system command prompt, type:

```
db2samp1 e
```

where *e* is an optional parameter specifying the drive where the database is to be created. Press Enter.<sup>97</sup>

If you are not logged on to your workstation through User Profile Management, you will be prompted to do so.

## To Erase the Sample Database

If you do not need to access the sample database, you can erase it by using the DROP DATABASE command:

```
db2 drop database sample
```

## CL\_SCHED Table

Name:	CLASS_CODE	DAY	STARTING	ENDING
Type:	char(7)	smallint	time	time
Desc:	Class Code (room:teacher)	Day # of 4 day schedule	Class Start Time	Class End Time

## DEPARTMENT Table

Name:	DEPTNO	DEPTNAME	MGRNO	ADMNDEPT	LOCATION
Type:	char(3) not null	varchar(29) not null	char(6)	char(3) not null	char(16)
Desc:	Department number	Name describing general activ- ities of department	Employee number (EMPNO) of department manager	Department (DEPTNO) to which this department reports	Name of the remote location

<sup>96</sup> If the path parameter is not specified, the sample tables are installed in the default path specified by the DFTDBPATH parameter in the database manager configuration file.

<sup>97</sup> If the drive parameter is not specified, the sample tables are installed on the same drive as DB2.

## Sample Tables

<b>Name:</b>	<b>DEPTNO</b>	<b>DEPTNAME</b>	<b>MGRNO</b>	<b>ADMRDEPT</b>	<b>LOCATION</b>
Values:	A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	-
	B01	PLANNING	000020	A00	-
	C01	INFORMATION CENTER	000030	A00	-
	D01	DEVELOPMENT CENTER	-	A00	-
	D11	MANUFACTURING SYSTEMS	000060	D01	-
	D21	ADMINISTRATION SYSTEMS	000070	D01	-
	E01	SUPPORT SERVICES	000050	A00	-
	E11	OPERATIONS	000090	E01	-
	E21	SOFTWARE SUPPORT	000100	E01	-

### EMPLOYEE Table

<b>Names:</b>	<b>EMPNO</b>	<b>FIRSTNAME</b>	<b>MIDINIT</b>	<b>LASTNAME</b>	<b>WORKDEPT</b>	<b>PHONENO</b>	<b>HIREDATE</b>
Type:	char(6) not null	varchar(12) not null	char(1) not null	varchar(15) not null	char(3)	char(4)	date
Desc:	Employee number	First name	Middle initial	Last name	Department (DEPTNO) in which the employee works	Phone number	Date of hire

<b>JOB</b>	<b>EDLEVEL</b>	<b>SEX</b>	<b>BIRTHDATE</b>	<b>SALARY</b>	<b>BONUS</b>	<b>COMM</b>
char(8)	smallint not null	char(1)	date	dec(9,2)	dec(9,2)	dec(9,2)
Job	Number of years of formal education	Sex (M male, F female)	Date of birth	Yearly salary	Yearly bonus	Yearly com- mission

See the following page for the values in the EMPLOYEE table.

# Sample Tables

EMPNO	FIRSTNAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
char(6) not null	varchar(12) not null	char(1) not null	varchar(15) not null	char(3)	char(4)	date	char(8)	smallint not null	char(1)	date	dec(9,2)	dec(9,2)	dec(9,2)
000010	CHRISTINE	I	HAAS	A00	3978	1985-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCHESSI	A00	3490	1988-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN	A00	O'CONNELL	A00	2167	1983-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE	D11	ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID	D11	BROWN	D11	4501	1986-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1988-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1989-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FIELDREP	16	M	1932-08-11	19950	400	1596
000330	WING	E21	LEE	E21	2103	1976-02-23	FIELDREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FIELDREP	16	M	1926-05-17	23840	500	1907



## Sample Tables

### EMP\_ACT Table

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
Type:	char(6) not null	char(6) not null	smallint not null	dec(5,2)	date	date
Desc:	Employee number	Project number	Activity number	Proportion of employee's time spent on project	Date activity starts	Date activity ends
Values:	000010	AD3100	10	.50	1982-01-01	1982-07-01
	000070	AD3110	10	1.00	1982-01-01	1983-02-01
	000230	AD3111	60	1.00	1982-01-01	1982-03-15
	000230	AD3111	60	.50	1982-03-15	1982-04-15
	000230	AD3111	70	.50	1982-03-15	1982-10-15
	000230	AD3111	80	.50	1982-04-15	1982-10-15
	000230	AD3111	180	1.00	1982-10-15	1983-01-01
	000240	AD3111	70	1.00	1982-02-15	1982-09-15
	000240	AD3111	80	1.00	1982-09-15	1983-01-01
	000250	AD3112	60	1.00	1982-01-01	1982-02-01
	000250	AD3112	60	.50	1982-02-01	1982-03-15
	000250	AD3112	60	.50	1982-12-01	1983-01-01
	000250	AD3112	60	1.00	1983-01-01	1983-02-01
	000250	AD3112	70	.50	1982-02-01	1982-03-15
	000250	AD3112	70	1.00	1982-03-15	1982-08-15
	000250	AD3112	70	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.50	1982-10-15	1982-12-01
	000250	AD3112	180	.50	1982-08-15	1983-01-01
	000260	AD3113	70	.50	1982-06-15	1982-07-01
	000260	AD3113	70	1.00	1982-07-01	1983-02-01
	000260	AD3113	80	1.00	1982-01-01	1982-03-01
	000260	AD3113	80	.50	1982-03-01	1982-04-15
	000260	AD3113	180	.50	1982-03-01	1982-04-15
	000260	AD3113	180	1.00	1982-04-15	1982-06-01
	000260	AD3113	180	.50	1982-06-01	1982-07-01
	000270	AD3113	60	.50	1982-03-01	1982-04-01
	000270	AD3113	60	1.00	1982-04-01	1982-09-01
	000270	AD3113	60	.25	1982-09-01	1982-10-15
	000270	AD3113	70	.75	1982-09-01	1982-10-15
	000270	AD3113	70	1.00	1982-10-15	1983-02-01
	000270	AD3113	80	1.00	1982-01-01	1982-03-01
	000270	AD3113	80	.50	1982-03-01	1982-04-01
	000030	IF1000	10	.50	1982-06-01	1983-01-01

## Sample Tables

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	000130	IF1000	90	1.00	1982-01-01	1982-10-01
	000130	IF1000	100	.50	1982-10-01	1983-01-01
	000140	IF1000	90	.50	1982-10-01	1983-01-01
	000030	IF2000	10	.50	1982-01-01	1983-01-01
	000140	IF2000	100	1.00	1982-01-01	1982-03-01
	000140	IF2000	100	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-10-01	1983-01-01
	000010	MA2100	10	.50	1982-01-01	1982-11-01
	000110	MA2100	20	1.00	1982-01-01	1982-03-01
	000010	MA2110	10	1.00	1982-01-01	1983-02-01
	000200	MA2111	50	1.00	1982-01-01	1982-06-15
	000200	MA2111	60	1.00	1982-06-15	1983-02-01
	000220	MA2111	40	1.00	1982-01-01	1983-02-01
	000150	MA2112	60	1.00	1982-01-01	1982-07-15
	000150	MA2112	180	1.00	1982-07-15	1983-02-01
	000170	MA2112	60	1.00	1982-01-01	1983-06-01
	000170	MA2112	70	1.00	1982-06-01	1983-02-01
	000190	MA2112	70	1.00	1982-02-01	1982-10-01
	000190	MA2112	80	1.00	1982-10-01	1983-10-01
	000160	MA2113	60	1.00	1982-07-15	1983-02-01
	000170	MA2113	80	1.00	1982-01-01	1983-02-01
	000180	MA2113	70	1.00	1982-04-01	1982-06-15
	000210	MA2113	80	.50	1982-10-01	1983-02-01
	000210	MA2113	180	.50	1982-10-01	1983-02-01
	000050	OP1000	10	.25	1982-01-01	1983-02-01
	000090	OP1010	10	1.00	1982-01-01	1983-02-01
	000280	OP1010	130	1.00	1982-01-01	1983-02-01
	000290	OP1010	130	1.00	1982-01-01	1983-02-01
	000300	OP1010	130	1.00	1982-01-01	1983-02-01
	000310	OP1010	130	1.00	1982-01-01	1983-02-01
	000050	OP2010	10	.75	1982-01-01	1983-02-01
	000100	OP2010	10	1.00	1982-01-01	1983-02-01
	000320	OP2011	140	.75	1982-01-01	1983-02-01
	000320	OP2011	150	.25	1982-01-01	1983-02-01
	000330	OP2012	140	.25	1982-01-01	1983-02-01
	000330	OP2012	160	.75	1982-01-01	1983-02-01
	000340	OP2013	140	.50	1982-01-01	1983-02-01
	000340	OP2013	170	.50	1982-01-01	1983-02-01
	000020	PL2100	30	1.00	1982-01-01	1982-09-15

**EMP\_PHOTO Table**

<b>Name:</b>	<b>EMPNO</b>	<b>PHOTO_FORMAT</b>	<b>PICTURE</b>
Type:	char(6) not null	varchar(10) not null	blob(100k)
Desc:	Employee number	Photo format	Photo of employee
Values:	000130	bitmap	db200130.bmp
	000130	gif	db200130.gif
	000130	xwd	db200130.xwd
	000140	bitmap	db200140.bmp
	000140	gif	db200140.gif
	000140	xwd	db200140.xwd
	000150	bitmap	db200150.bmp
	000150	gif	db200150.gif
	000150	xwd	db200150.xwd
	000190	bitmap	db200190.bmp
	000190	gif	db200190.gif
	000190	xwd	db200190.xwd

- “Quintana Photo” on page 875 shows the picture of the employee, Delores Quintana.
- “Nicholls Photo” on page 876 shows the picture of the employee, Heather Nicholls.
- “Adamson Photo” on page 877 shows the picture of the employee, Bruce Adamson.
- “Walker Photo” on page 879 shows the picture of the employee, James Walker.

**EMP\_RESUME Table**

<b>Name:</b>	<b>EMPNO</b>	<b>RESUME_FORMAT</b>	<b>RESUME</b>
Type:	char(6) not null	varchar(10) not null	clob(5k)
Desc:	Employee number	Resume Format	Resume of employee
Values:	000130	ascii	db200130.asc
	000130	script	db200130.scr
	000140	ascii	db200140.asc
	000140	script	db200140.scr
	000150	ascii	db200150.asc
	000150	script	db200150.scr
	000190	ascii	db200190.asc
	000190	script	db200190.scr

- “Quintana Resume” on page 875 shows the resume of the employee, Delores Quintana.

## Sample Tables

- “Nicholls Resume” on page 876 shows the resume of the employee, Heather Nicholls.
- “Adamson Resume” on page 878 shows the resume of the employee, Bruce Adamson.
- “Walker Resume” on page 879 shows the resume of the employee, James Walker.

## IN\_TRAY Table

Name:	RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
Type:	timestamp	char(8)	char(64)	varchar(3000)
Desc:	Date and Time received	User id of person sending note	Brief description	The note

## ORG Table

Name:	DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
Type:	smallint not null	varchar(14)	smallint	varchar(10)	varchar(13)
Desc:	Department number	Department name	Manager number	Division of corporation	City
Values:	10	Head Office	160	Corporate	New York
	15	New England	50	Eastern	Boston
	20	Mid Atlantic	10	Eastern	Washington
	38	South Atlantic	30	Eastern	Atlanta
	42	Great Lakes	100	Midwest	Chicago
	51	Plains	140	Midwest	Dallas
	66	Pacific	270	Western	San Francisco
	84	Mountain	290	Western	Denver

## PROJECT Table

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
Type:	char(6) not null	varchar(24) not null	char(3) not null	char(6) not null	dec(5,2)	date	date	char(6)
Desc:	Project number	Project name	Department responsible	Employee responsible	Estimated mean staffing	Estimated start date	Estimated end date	Major project, for a sub-project
Values:	AD3100	ADMIN SERVICES	D01	000010	6.5	1982-01-01	1983-02-01	-
	AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	1982-01-01	1983-02-01	AD3100
	AD3111	PAYROLL PROGRAMMING	D21	000230	2	1982-01-01	1983-02-01	AD3110

## Sample Tables

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
	AD3112	PERSONNEL PROGRAMMING	D21	000250	1	1982-01-01	1983-02-01	AD3110
	AD3113	ACCOUNT PROGRAMMING	D21	000270	2	1982-01-01	1983-02-01	AD3110
	IF1000	QUERY SERVICES	C01	000030	2	1982-01-01	1983-02-01	-
	IF2000	USER EDUCATION	C01	000030	1	1982-01-01	1983-02-01	-
	MA2100	WELD LINE AUTOMATION	D01	000010	12	1982-01-01	1983-02-01	-
	MA2110	W L PROGRAMMING	D11	000060	9	1982-01-01	1983-02-01	MA2100
	MA2111	W L PROGRAM DESIGN	D11	000220	2	1982-01-01	1982-12-01	MA2110
	MA2112	W L ROBOT DESIGN	D11	000150	3	1982-01-01	1982-12-01	MA2110
	MA2113	W L PROD CONT PROGS	D11	000160	3	1982-02-15	1982-12-01	MA2110
	OP1000	OPERATION SUPPORT	E01	000050	6	1982-01-01	1983-02-01	-
	OP1010	OPERATION	E11	000090	5	1982-01-01	1983-02-01	OP1000
	OP2000	GEN SYSTEMS SERVICES	E01	000050	5	1982-01-01	1983-02-01	-
	OP2010	SYSTEMS SUPPORT	E21	000100	4	1982-01-01	1983-02-01	OP2000
	OP2011	SCP SYSTEMS SUPPORT	E21	000320	1	1982-01-01	1983-02-01	OP2010
	OP2012	APPLICATIONS SUPPORT	E21	000330	1	1982-01-01	1983-02-01	OP2010
	OP2013	DB/DC SUPPORT	E21	000340	1	1982-01-01	1983-02-01	OP2010
	PL2100	WELD LINE PLANNING	B01	000020	1	1982-01-01	1982-09-15	MA2100

## SALES Table

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
Type:	date	varchar(15)	varchar(15)	int
Desc:	Date of sales	Employee's last name	Region of sales	Number of sales
Values:	12/31/1995	LUCCHESSI	Ontario-South	1
	12/31/1995	LEE	Ontario-South	3

## Sample Tables

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
	12/31/1995	LEE	Quebec	1
	12/31/1995	LEE	Manitoba	2
	12/31/1995	GOUNOT	Quebec	1
	03/29/1996	LUCCHESSI	Ontario-South	3
	03/29/1996	LUCCHESSI	Quebec	1
	03/29/1996	LEE	Ontario-South	2
	03/29/1996	LEE	Ontario-North	2
	03/29/1996	LEE	Quebec	3
	03/29/1996	LEE	Manitoba	5
	03/29/1996	GOUNOT	Ontario-South	3
	03/29/1996	GOUNOT	Quebec	1
	03/29/1996	GOUNOT	Manitoba	7
	03/30/1996	LUCCHESSI	Ontario-South	1
	03/30/1996	LUCCHESSI	Quebec	2
	03/30/1996	LUCCHESSI	Manitoba	1
	03/30/1996	LEE	Ontario-South	7
	03/30/1996	LEE	Ontario-North	3
	03/30/1996	LEE	Quebec	7
	03/30/1996	LEE	Manitoba	4
	03/30/1996	GOUNOT	Ontario-South	2
	03/30/1996	GOUNOT	Quebec	18
	03/30/1996	GOUNOT	Manitoba	1
	03/31/1996	LUCCHESSI	Manitoba	1
	03/31/1996	LEE	Ontario-South	14
	03/31/1996	LEE	Ontario-North	3
	03/31/1996	LEE	Quebec	7
	03/31/1996	LEE	Manitoba	3
	03/31/1996	GOUNOT	Ontario-South	2
	03/31/1996	GOUNOT	Quebec	1
	04/01/1996	LUCCHESSI	Ontario-South	3
	04/01/1996	LUCCHESSI	Manitoba	1
	04/01/1996	LEE	Ontario-South	8
	04/01/1996	LEE	Ontario-North	-
	04/01/1996	LEE	Quebec	8
	04/01/1996	LEE	Manitoba	9
	04/01/1996	GOUNOT	Ontario-South	3
	04/01/1996	GOUNOT	Ontario-North	1
	04/01/1996	GOUNOT	Quebec	3
	04/01/1996	GOUNOT	Manitoba	7

## STAFF Table

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	smallint not null	varchar(9)	smallint	char(5)	smallint	dec(7,2)	dec(7,2)

## Sample Tables

<b>Name:</b>	<b>ID</b>	<b>NAME</b>	<b>DEPT</b>	<b>JOB</b>	<b>YEARS</b>	<b>SALARY</b>	<b>COMM</b>
Desc:	Employee number	Employee name	Department number	Job type	Years of service	Current salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	-
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	-
	40	O'Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	-
	60	Quigley	38	Sales	-	16808.30	650.25
	70	Rothman	15	Sales	7	16502.83	1152.00
	80	James	20	Clerk	-	13504.60	128.20
	90	Koonitz	42	Sales	6	18001.75	1386.70
	100	Plotz	42	Mgr	7	18352.80	-
	110	Ngan	15	Clerk	5	12508.20	206.60
	120	Naughton	38	Clerk	-	12954.75	180.00
	130	Yamaguchi	42	Clerk	6	10505.90	75.60
	140	Fraye	51	Mgr	6	21150.00	-
	150	Williams	51	Sales	6	19456.50	637.65
	160	Molinare	10	Mgr	7	22959.20	-
	170	Kermisch	15	Clerk	4	12258.50	110.10
	180	Abrahams	38	Clerk	3	12009.75	236.50
	190	Sneider	20	Clerk	8	14252.75	126.50
	200	Scoutten	42	Clerk	-	11508.60	84.20
	210	Lu	10	Mgr	10	20010.00	-
	220	Smith	51	Sales	7	17654.50	992.80
	230	Lundquist	51	Clerk	3	13369.80	189.65
	240	Daniels	10	Mgr	5	19260.25	-
	250	Wheeler	51	Clerk	6	14460.00	513.30
	260	Jones	10	Mgr	12	21234.00	-
	270	Lea	66	Mgr	9	18555.50	-
	280	Wilson	66	Sales	9	18674.50	811.50
	290	Quill	84	Mgr	10	19818.00	-
	300	Davis	84	Sales	5	15454.50	806.10
	310	Graham	66	Sales	13	21000.00	200.30
	320	Gonzales	66	Sales	4	16858.20	844.00
	330	Burke	66	Clerk	1	10988.00	55.50
	340	Edwards	84	Sales	7	17844.00	1285.00
	350	Gafney	84	Clerk	5	13030.50	188.00

## Sample Tables

### STAFFG Table

**Note:** STAFFG is only created for double-byte code pages.

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	smallint not null	varchar(9)	smallint	graphic(5)	smallint	dec(9,0)	dec(9,0)
Desc:	Employee number	Employee name	Department number	Job type	Years of service	Current salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	-
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	-
	40	O'Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	-
	60	Quigley	38	Sales	-	16808.30	650.25
	70	Rothman	15	Sales	7	16502.83	1152.00
	80	James	20	Clerk	-	13504.60	128.20
	90	Koonitz	42	Sales	6	18001.75	1386.70
	100	Plotz	42	Mgr	7	18352.80	-
	110	Ngan	15	Clerk	5	12508.20	206.60
	120	Naughton	38	Clerk	-	12954.75	180.00
	130	Yamaguchi	42	Clerk	6	10505.90	75.60
	140	Fraye	51	Mgr	6	21150.00	-
	150	Williams	51	Sales	6	19456.50	637.65
	160	Molinare	10	Mgr	7	22959.20	-
	170	Kermisch	15	Clerk	4	12258.50	110.10
	180	Abrahams	38	Clerk	3	12009.75	236.50
	190	Sneider	20	Clerk	8	14252.75	126.50
	200	Scoutten	42	Clerk	-	11508.60	84.20
	210	Lu	10	Mgr	10	20010.00	-
	220	Smith	51	Sales	7	17654.50	992.80
	230	Lundquist	51	Clerk	3	13369.80	189.65
	240	Daniels	10	Mgr	5	19260.25	-
	250	Wheeler	51	Clerk	6	14460.00	513.30
	260	Jones	10	Mgr	12	21234.00	-
	270	Lea	66	Mgr	9	18555.50	-
	280	Wilson	66	Sales	9	18674.50	811.50
	290	Quill	84	Mgr	10	19818.00	-
	300	Davis	84	Sales	5	15454.50	806.10
	310	Graham	66	Sales	13	21000.00	200.30
	320	Gonzales	66	Sales	4	16858.20	844.00
	330	Burke	66	Clerk	1	10988.00	55.50
	340	Edwards	84	Sales	7	17844.00	1285.00
	350	Gafney	84	Clerk	5	13030.50	188.00



---

**Sample Files with BLOB and CLOB Data Type**

This section shows the data found in the EMP\_PHOTO files (pictures of employees) and EMP\_RESUME files (resumes of employees).

**Quintana Photo**



*Figure 14. Delores M. Quintana*

**Quintana Resume**

The following text is found in the db200130.asc and db200130.scr files.

**Resume: Delores M. Quintana**

**Personal Information**

Address: 1150 Eglinton Ave Mellonville, Idaho 83725  
 Phone: (208) 555-9933  
 Birthdate: September 15, 1925  
 Sex: Female  
 Marital Status: Married  
 Height: 5'2"  
 Weight: 120 lbs.

**Department Information**

Employee Number: 000130  
 Dept Number: C01  
 Manager: Sally Kwan  
 Position: Analyst  
 Phone: (208) 555-4578  
 Hire Date: 1971-07-28

**Education**

1965 Math and English, B.A. Adelphi University  
 1960 Dental Technician Florida Institute of Technology

## Sample Tables

### Work History

10/91 - present	Advisory Systems Analyst Producing documentation tools for engineering department.
12/85 - 9/91	Technical Writer Writer, text programmer, and planner.
1/79 - 11/85	COBOL Payroll Programmer Writing payroll programs for a diesel fuel company.

### Interests

- Cooking
- Reading
- Sewing
- Remodeling

## Nicholls Photo



Figure 15. Heather A. Nicholls

## Nicholls Resume

The following text is found in the db200140.asc and db200140.scr files.

### Resume: Heather A. Nicholls

#### Personal Information

Address:	844 Don Mills Ave Mellonville, Idaho 83734
Phone:	(208) 555-2310
Birthdate:	January 19, 1946
Sex:	Female
Marital Status:	Single
Height:	5'8"
Weight:	130 lbs.

### Department Information

Employee Number: 000140  
Dept Number: C01  
Manager: Sally Kwan  
Position: Analyst  
Phone: (208) 555-1793  
Hire Date: 1976-12-15

### Education

1972 Computer Engineering, Ph.D. University of Washington  
1969 Music and Physics, M.A. Vassar College

### Work History

2/83 - present Architect, OCR Development Designing the architecture of OCR products.  
12/76 - 1/83 Text Programmer Optical character recognition (OCR) programming in PL/I.  
9/72 - 11/76 Punch Card Quality Analyst Checking punch cards met quality specifications.

### Interests

- Model railroading
- Interior decorating
- Embroidery
- Knitting

### Adamson Photo



Figure 16. Bruce Adamson

## Sample Tables

### Adamson Resume

The following text is found in the db200150.asc and db200150.scr files.

#### Resume: Bruce Adamson

##### Personal Information

Address: 3600 Steeles Ave Mellonville, Idaho 83757  
Phone: (208) 555-4489  
Birthdate: May 17, 1947  
Sex: Male  
Marital Status: Married  
Height: 6'0"  
Weight: 175 lbs.

##### Department Information

Employee Number: 000150  
Dept Number: D11  
Manager: Irving Stern  
Position: Designer  
Phone: (208) 555-4510  
Hire Date: 1972-02-12

##### Education

1971 Environmental Engineering, M.Sc. Johns Hopkins University  
1968 American History, B.A. Northwestern University

##### Work History

8/79 - present Neural Network Design Developing neural networks for machine intelligence products.  
2/72 - 7/79 Robot Vision Development Developing rule-based systems to emulate sight.  
9/71 - 1/72 Numerical Integration Specialist Helping bank systems communicate with each other.

##### Interests

- Racing motorcycles
- Building loudspeakers
- Assembling personal computers
- Sketching

**Walker Photo**



Figure 17. James H. Walker

**Walker Resume**

The following text is found in the db200190.asc and db200190.scr files.

**Resume: James H. Walker**

**Personal Information**

Address: 3500 Steeles Ave Mellonville, Idaho 83757  
 Phone: (208) 555-7325  
 Birthdate: June 25, 1952  
 Sex: Male  
 Marital Status: Single  
 Height: 5'11"  
 Weight: 166 lbs.

**Department Information**

Employee Number: 000190  
 Dept Number: D11  
 Manager: Irving Stern  
 Position: Designer  
 Phone: (208) 555-2986  
 Hire Date: 1974-07-26

**Education**

1974 Computer Studies, B.Sc. University of Massachusetts  
 1972 Linguistic Anthropology, B.A. University of Toronto

**Work History**

## Sample Tables

6/87 - present	Microcode Design Optimizing algorithms for mathematical functions.
4/77 - 5/87	Printer Technical Support Installing and supporting laser printers.
9/74 - 3/77	Maintenance Programming Patching assembly language compiler for mainframes.

### Interests

- Wine tasting
- Skiing
- Swimming
- Dancing

---

### Appendix G. Reserved Schema Names and Reserved Words

This appendix describes the restrictions of certain names used by the database manager. In some cases, names are reserved and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs though not prevented by the database manager.

---

#### Reserved Schemas

The following schema names are reserved:

- SYSCAT
- SYSFUN
- SYSIBM
- SYSSTAT

In addition, it is strongly recommended that schema names never begin with the SYS prefix, as SYS is by convention used to indicate an area reserved by the system.

No user-defined functions, user-defined distinct types, triggers, or aliases can be placed into a schema whose name starts with SYS (SQLSTATE 42939).

---

#### Reserved Words

There are no words that are specifically reserved words in DB2 Version 5.2.

Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement unless it is delimited.

IBM SQL and ISO/ANSI SQL92 include reserved words, listed in the following section. These reserved words are not enforced by DB2 Universal Database, however it is recommended that they not be used as ordinary identifiers, since this reduces portability.

---

#### IBM SQL Reserved Words

The IBM SQL reserved words are as follows.

## Reserved Schema Names and Reserved Words

ACQUIRE	DATABASE	IDENTIFIED	OBID	SCHEDULE
ADD	DATE	IMMEDIATE	OF	SCHEMA
ALL	DAY	IN	ON	SECOND
ALLOCATE	DAYS	INDEX	ONLY	SECONDS
ALTER	DBA	INDICATOR	OPTIMIZE	SECQTY
AND	DBSPACE	INNER	OPTION	SELECT
ANY	DEFAULT	INOUT	OR	SET
AS	DELETE	INSERT	ORDER	SHARE
ASC	DESC	INTERSECT	OUT	SIMPLE
AUDIT	DESCRIPTOR	INTO	OUTER	SOME
AUTHORIZATION	DISTINCT	IS		STATISTICS
AVG	DOUBLE	ISOLATION	PACKAGE	STOGROUP
	DROP		PAGE	STORPOOL
BETWEEN		JOIN	PAGES	SUBPAGES
BUFFERPOOL	EDITPROC		PART	SUBSTRING
BY	END-EXEC	KEY	PCTFREE	SUM
	ELSE		PCTINDEX	SYNONYM
CALL	ERASE	LABEL	PLAN	
CAPTURE	ESCAPE	LEFT	PRECISION	TABLE
CASE	EXCEPT	LIKE	PRIMARY	TABLESPACE
CAST	EXCEPTION	LOCK	PRIQTY	TO
CCSID	EXCLUSIVE	LOCKSIZE	PRIVATE	TRANSACTION
CHAR	EXECUTE	LONG	PRIVILEGES	TRIM
CHARACTER	EXISTS		PROCEDURE	
CHECK	EXPLAIN	MAX	PROGRAM	UNION
CLUSTER	EXTERNAL	MICROSECOND	PUBLIC	UNIQUE
COLLECTION		MICROSECONDS		UPDATE
COLUMN	FETCH	MIN	REFERENCES	USER
COMMENT	FIELDPROC	MINUTE	RELEASE	USING
COMMIT	FOR	MINUTES	RESET	
CONCAT	FOREIGN	MODE	RESOURCE	VALIDPROC
CONNECT	FROM	MONTH	REVOKE	VALUES
CONNECTION	FULL	MONTHS	RIGHT	VARIABLE
CONSTRAINT			ROLLBACK	VCAT
COUNT	GO	NAMED	ROW	VIEW
CREATE	GOTO	NHEADER	ROWS	VOLUMES
CROSS	GRANT	NOT	RRN	
CURRENT	GRAPHIC	NULL	RUN	WHERE
CURRENT_DATE	GROUP	NUMPARTS		WITH
CURRENT_SERVER				WORK
CURRENT_TIME	HAVING			
CURRENT_TIMESTAMP	HOUR			YEAR
CURRENT_TIMEZONE	HOURS			YEARS
CURRENT_USER				
CURSOR				

---

## ISO/ANS SQL92 Reserved Words

The ISO/ANS SQL92 reserved words that are not also in the IBM SQL list are as follows.



## Reserved Schema Names and Reserved Words

ABSOLUTE	FALSE	OCTET_LENGTH	UNKNOWN
ACTION	FIRST	OPEN	UPPER
ARE	FLOAT	OUTER	USAGE
ASSERTION	FOUND	OUTPUT	
AT	FULL	OVERLAPS	VALUE
			VARCHAR
BEGIN	GET	PAD	VARYING
BIT_LENGTH	GLOBAL	PARTIAL	
BOTH		POSITION	WHEN
	IDENTITY	PREPARE	WHENEVER
CASCADED	INITIALLY	PRESERVE	WRITE
CATALOG	INNER	PRIOR	
CHAR_LENGTH	INPUT		ZONE
CHARACTER_LENGTH	INSENSITIVE	READ	
CLOSE	INTERVAL	REAL	
COALESCE		RELATIVE	
COLLATE	JOIN	RIGHT	
COLLATION			
CONSTRAINTS	LANGUAGE	SCROLL	
CONTINUE	LAST	SECTION	
CONVERT	LEADING	SESSION	
CORRESPONDING	LEFT	SESSION_USER	
	LEVEL	SIZE	
DEALLOCATE	LOCAL	SMALLINT	
DEC	LOWER	SPACE	
DECIMAL		SQL	
DECLARE	MATCH	SQLCODE	
DEFERRABLE	MODULE	SQLERROR	
DEFERRED		SQLSTATE	
DESCRIBE	NAMES	SYSTEM_USER	
DIAGNOSTICS	NATIONAL		
DISCONNECT	NATURAL	TEMPORARY	
DOMAIN	NCHAR	THEN	
	NEXT	TIMEZONE_HOUR	
END	NO	TIMEZONE_MINUTE	
EXTRACT	NULLIF	TRAILING	
	NUMERIC	TRANSLATION	
		TRUE	

# Reserved Schema Names and Reserved Words

## Appendix H. Comparison of Isolation Levels

The following table summarizes information about isolation levels described in “Isolation Level” on page 22.

	UR	CS	RS	RR
Can the application see uncommitted changes made by other application processes?	Yes	No	No	No
Can the application update uncommitted changes made by other application processes?	No	No	No	No
Can the re-execution of a statement be affected by other application processes? <i>See phenomenon P3 (phantom) below.</i>	Yes	Yes	Yes	No
Can “updated” rows be updated by other application processes?	No	No	No	No
Can “updated” rows be read by other application processes that are running at an isolation level other than UR?	No	No	No	No
Can “updated” rows be read by other application processes that are running at the UR isolation level?	Yes	Yes	Yes	Yes
Can “accessed” rows be updated by other application processes? <i>See phenomenon P2 (nonrepeatable read) below.</i>	Yes	Yes	No	No
Can “accessed” rows be read by other application processes?	Yes	Yes	Yes	Yes
Can “current” row be updated or deleted by other application processes? <i>See phenomenon P1 (dirty-read) below.</i>	See Note below	See Note below	No	No

### Note:

1. If the cursor is not updatable, with CS the current row may be updated or deleted by other application processes in some cases.

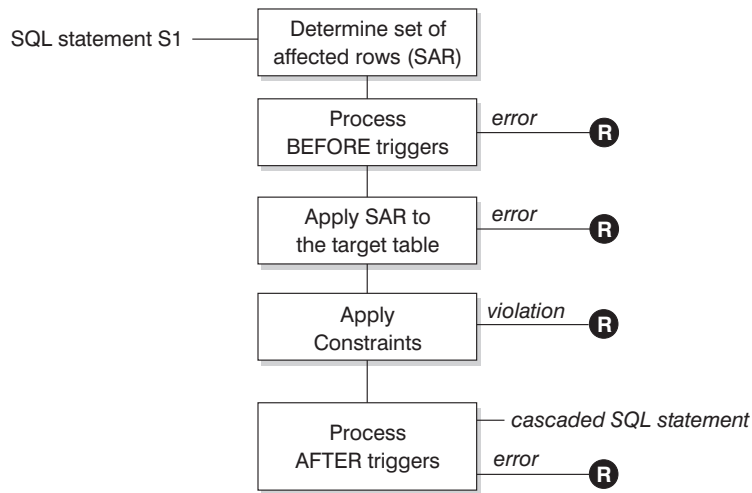
### Examples of Phenomena:

- P1** *Dirty Read.* Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 performs a COMMIT. If UW1 then performs a ROLLBACK, UW2 has read a nonexistent row.
- P2** *Nonrepeatable Read.* Unit of work UW1 reads a row. Unit of work UW2 modifies that row and performs a COMMIT. If UW1 then re-reads the row, it might receive a modified value.
- P3** *Phantom.* Unit of work UW1 reads the set of  $n$  rows that satisfies some search condition. Unit of work UW2 then INSERTs one or more rows that satisfies the search condition. If UW1 then repeats the initial read with the same search condition, it obtains the original rows plus the inserted rows.

## Isolation Levels

### Appendix I. Interaction of Triggers and Constraints

This appendix describes the interaction of triggers with referential constraints and check constraints that may result from an update operation. Figure 18 and the associated description are representative of the processing that is performed for an SQL statement that updates data in the database.



**R** = rollback changes to before S1

Figure 18. Processing an SQL statement with associated triggers and constraints

Figure 18 shows the general order of processing for an SQL statement that updates a table. It assumes a situation where the table includes before triggers, referential constraints, check constraints and after triggers that cascade. The following is a description of the boxes and other items found in Figure 18.

- SQL statement  $S_1$   
This is the DELETE, INSERT, or UPDATE statement that begins the process. The SQL statement  $S_1$  identifies a table (or an updatable view over some table) referred to as the *target table* throughout this description.
- Determine set of affected rows (SAR)  
This step is the starting point for a process that repeats for referential constraint delete rules of CASCADE and SET NULL and for cascaded SQL statements from after triggers.  
  
The purpose of this step is to determine the *set of affected rows* for the SQL statement. The set of rows included in SAR is based on the statement:
  - for DELETE, all rows that satisfy the search condition of the statement (or the current row for a positioned DELETE)
  - for INSERT, the rows identified by the VALUES clause or the fullselect

## Interaction of Triggers and Constraints

- for UPDATE, all rows that satisfy the search condition (or the current row for a positioned update).

If *SAR* is empty, there will be no BEFORE triggers, changes to apply to the target table, or constraints to process for the SQL statement.

- Process BEFORE triggers

All BEFORE triggers are processed in ascending order of creation. Each BEFORE trigger will process the triggered action once for each row in *SAR*.

An error may occur during the processing of a triggered action in which case all changes made as a result of the original SQL statement  $S_1$  (so far) are rolled back.

If there are no BEFORE triggers or the *SAR* is empty, this step is skipped.

- Apply *SAR* to the target table

The actual delete, insert, or update is applied using *SAR* to the target table in the database.

An error may occur when applying *SAR* (such as attempting to insert a row with a duplicate key where a unique index exists) in which case all changes made as a result of the original SQL statement  $S_1$  (so far) are rolled back.

- Apply Constraints

The constraints associated with the target table are applied if *SAR* is not empty. This includes unique constraints, unique indexes, referential constraints, check constraints and checks related to the WITH CHECK OPTION on views. Referential constraints with delete rules of cascade or set null may cause additional triggers to be activated.

A violation of any constraint or WITH CHECK OPTION results in an error and all changes made as a result of  $S_1$  (so far) are rolled back.

- Process AFTER triggers

All AFTER triggers activated by  $S_1$  are processed in ascending order of creation.

FOR EACH STATEMENT triggers will process the triggered action exactly once, even if *SAR* is empty. FOR EACH ROW triggers will process the triggered action once for each row in *SAR*.

An error may occur during the processing of a triggered action in which case all changes made as a result of the original  $S_1$  (so far) are rolled back.

The triggered action of a trigger may include triggered SQL statements that are DELETE, INSERT or UPDATE statements. For the purposes of this description, each such statement is considered a *cascaded SQL statement*.

A cascaded SQL statement is a DELETE, INSERT, or UPDATE statement that is processed as part of the triggered action of an AFTER trigger. This statement starts a cascaded level of trigger processing. This can be thought of as assigning the triggered SQL statement as a new  $S_1$  and performing all of the steps described here recursively.

## Interaction of Triggers and Constraints

Once all triggered SQL statements from all AFTER triggers activated by each  $S_1$  have been processed to completion, the processing of the original  $S_1$  is completed.

- **R** = roll back changes to before  $S_1$

Any error (including constraint violations) that occurs during processing results in a roll back of all the changes made directly or indirectly as a result of the original SQL statement  $S_1$ . The database is therefore back in the same state as immediately prior to the execution of the original SQL statement  $S_1$

## Interaction of Triggers and Constraints



---

## Appendix J. Incompatibilities Between Releases

This appendix identifies the incompatibilities that exist between DB2 Universal Database and previous releases of DB2.

An “incompatibility” is defined to be a part of DB2 Universal Database that works differently than it did in a previous release of DB2 in such a way that if it used in an existing application it will produce a different result, necessitate a change to the application, or reduce performance. In this definition, “application” can apply to a broad range of things, such as:

- Application program code
- Third-party utilities
- Interactive SQL queries
- Command and/or API invocation.

This appendix does not describe incompatibilities where certain operations in the current release are less likely to generate an error condition than they did in the previous release, as those changes will only have a positive impact on existing applications.

This appendix lists incompatibilities in the following categories:

- “System Catalog Tables/Views” on page 892
- “Application Programming” on page 894
- “SQL” on page 907
- “Database Security and Tuning” on page 916
- “Utilities and Tools” on page 918
- “Connectivity and Coexistence” on page 921
- “Configuration Parameters” on page 925

Each incompatibility includes a description of the change in DB2 Version 5 that causes an incompatibility with previous releases, the symptom or effect this will have on your environment if no changes are made to it, and the possible resolutions that are available. There is also an indicator at the beginning of each incompatibility telling you what platforms are applicable as follows:

<b>DB2 PE</b>	DB2 Parallel Edition, Version 1.2
<b>OS/2</b>	OS/2
<b>UNIX</b>	Unix-based operating systems supported by DB2
<b>WIN</b>	Microsoft Windows platforms supported by DB2

---

## System Catalog Tables/Views

### System Catalog Views

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

#### Change

A set of views have been created in DB2 Version 2 with the qualifiers (also known as *schema names*) of SYSCAT and SYSSTAT. For this reason, the SYSCAT and SYSSTAT schemas are now reserved.

#### Symptom

If there are any objects belonging to these schemas in a Version 1 database, migration will fail with SQLCODE SQL1704N (reason code 1).

#### Resolution

The only way to get through the migration successfully will be to recreate the objects currently under the SYSCAT and SYSSTAT schemas under new high level qualifiers.

### System Catalog Tables

UNIX	OS/2	WIN
------	------	-----

#### Change

A variety of changes have been made to the SYSIBM tables. This section will discuss the subset which could cause incompatibilities. To see a description of all changes (for example, new columns, new values in a column, and so on) refer to the *SQL Reference*.

#### SYSCOLUMNS

<b>COLTYPE:</b>	Changed values: "FLOAT" to "DOUBLE"
<b>NULLS:</b>	Changed values: "D" to "N". (Default flag now found in SYSCAT.COLUMNS.DEFAULT)
<b>HIGH2KEY:</b>	Changed type: VARCHAR(16) to VARCHAR(33). Changed values: Values now stored in printable format rather than binary format
<b>LOW2KEY:</b>	Changed type: VARCHAR(16) to VARCHAR(33). Changed values: Values are now stored in printable format rather than binary format for all datatypes.

#### SYSINDEXES

<b>CLUSTERRATIO:</b>	Changed value: Value will always be -1 if the columns CLUSTERFACTOR and PAGE_FETCH_PAIRS are populated.
<b>SECT_INFO:</b>	Changed type: LONG VARCHAR to BLOB(1M).
<b>HOST_VARS:</b>	Changed type: LONG VARCHAR to BLOB(1M).

**ISOLATION:** Changed type: CHAR(1) to CHAR(2). Changed values: "R" to "RR", "S" to "RS", "C" to "CS", "U" to "UR".

#### **SYSRELS**

**RELNAME:** Changed type: CHAR(8) to VARCHAR(18).

#### **SYSSECTION**

**SECTION:** Changed type: VARCHAR(3900) to VARCHAR(3600)

#### **SYSSTMT**

**TEXT:** Changed type: VARCHAR(3900) to VARCHAR(3600)

#### **SYSTABLES**

**PACKED\_DESC:** Changed type: LONG VARCHAR to BLOB(10M)

**VIEW\_DESC:** Changed type: LONG VARCHAR to BLOB(32K)

**REL\_DESC** Changed type: LONG VARCHAR to BLOB(32K)

**FID** Will no longer uniquely identify a table on its own. Must be used with TID to uniquely identify a table.

#### **SYSVIEWS**

**CHECK:** Changed values: "Y" to "L".

**TEXT:** Changed type: VARCHAR(3900) to VARCHAR(3600). Contains the full text of the create view statement (including the CREATE VIEW). In Version 1, only the select portion was shown.

### **Symptom**

A variety of symptoms could occur.

If you have an application which has a qualified search on a field that has had a value change (for example, ISOLATION in SYSIBM.SYSPLAN) this will cause your application to react differently than you would want.

If you have an application which accesses some field where the field type or size has changed (such as SECTION in SYSIBM.SYSSECTION), you could retrieve an incomplete set of data, too much data, or have the wrong type defined in your application to represent the data type of the table column.

### **Resolution**

If you use the SYSIBM tables for application processing or anything else, you must review the changes listed above to decide whether or not you are affected and what the appropriate action to correct the situation is. You may need to refer to the *SQL Reference* to understand what the new columns, new values for columns and other changes that were made to these tables.

If you need a rough approximation of the degree of clustering, select both CLUSTERRATIO and CLUSTERFACTOR and choose the "greater" one.

## Unique Table Identification

UNIX	OS/2	WIN
------	------	-----

### Change

With the introduction of table spaces, the TID column of SYSIBM.SYSTABLES is now used to identify a table space. The FID column of SYSIBM.SYSTABLES will no longer uniquely identify a table in a database. FID now uniquely identifies a table in a table space. This means that to uniquely identify a table in a database you need the combination of TID and FID.

### Symptom

Any application which assumes FID will uniquely identify a table in a database may process incorrectly should the FID be duplicated in multiple table spaces.

### Resolution

Change the application to use TBSPACEID and TABLEID from the SYSCAT.TABLES view as the unique identifier. You can also use the columns TID and FID from SYSIBM.SYSTABLES.

---

## Application Programming

### NS, NW and NX Locks

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

Due to the addition of NS and NX lock modes in DB2 Version 5, there is a difference in the behaviour of index scans with isolation level Cursor Stability (CS) or Read Stability (RS).

### Symptom

In DB2 Version 5, an index scan with isolation level, CS or RS, will not see an uncommitted delete of a row that is within the scanned range. In DB2 Version 2, the scanner would not see an uncommitted delete of a row that was at the end of the scanned range. However, if the deleted row was within the range, the scanner would remain in a lock wait until the delete was committed or rolled back.

For example in DB2 Version 5, the following can occur with an index on Column A:

Sequence	Application 1	Application 2
1	delete from t1 where a=3	
2		select a from t1 where a>1 and a<5 A ----- 2 4 5
3	rollback	
4		select a from t1 where a>1 and a<5 A ----- 2 3 4 5

The same scenario in previous versions of DB2 would result in application 2 being in lock wait until Application 1 committed or rolled back.

### Resolution

There is no resolution as this is an enhancement to isolation level Cursor Stability or Read Stability.

### Symptom

The previous example showed what occurs with an uncommitted deletion. A similar situation could also arise when inserting new values.

For example, you could have a scenario where you are scanning a table using an index on a column and looking for a value greater than or equal to two, but less than or equal to six, while using an isolation level of RS. The existing values that qualify in this example are two, four, and six. Then another user inserts five. An NS lock is obtained on columns returning two, four, and six; and the NW lock attempt on the column containing six succeeds, so the insertion of five is not blocked by the scan.

In Version 2, an S lock would be obtained on columns with the values two, four, and six; and the attempt to get an X lock on the column returning six would wait. The insert of five would wait for the S lock on six to be released.

### Resolution

In general, since more concurrency is supported in Version 5, applications built with a previous version of DB2 that were created with dependencies on some lock waiting may require modification.

## CREATE TABLE NOT LOGGED INITIALLY

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

In DB2 PE V1.2, in the unit of work in which a table is created with the NOT LOGGED INITIALLY option, an error on this table will cause the unit of work to be rolled back. In Version 5, the range of errors that will cause a roll back has been increased.

### Symptom

In Version 5, in the unit of work in which a table is created with the NOT LOGGED INITIALLY option, an error in any operation involving any table will cause the unit of work to be rolled back.

### Resolution

Correct the error and run the transaction again.

## DB2 Call Level Interface (DB2 CLI) Defaults

UNIX	OS/2	WIN
------	------	-----

### Change

The default values for **AUTOCOMMIT** and **CURSORHOLD** have changed. Both AUTOCOMMIT and CURSORHOLD will now default to ON.

### Symptom

If an application was written assuming that AUTOCOMMIT was OFF or that WITH HOLD semantics was NOT used for cursors, then these default changes could cause the application to fail.

### Resolution

Add one or both of the following two lines to your DB2CLI.INI file.

- AUTOCOMMIT = 0
- CURSORHOLD = 0

## Obsolete DB2 CLI Keywords

UNIX	OS/2	WIN
------	------	-----

### Change

You can control DB2 configurable features by specifying a set of optional keywords in an DB2 CLI initialization file. In DB2 Version 2, some of these keywords become obsolete, as follows:

1. UNDERSCORE
2. TRANSLATEDLL
3. TRANSLATIONOPTION

### Symptom

These keywords will be ignored if they still exist. You may notice behavioral changes based on the removal of these settings.

### Resolution

You will need to review the new list of valid parameters to decide what the appropriate keywords and settings are for your environment. See the *CLI Guide and Reference* for information on these keywords.

## DB2 CLI SQLSTATES

UNIX	OS/2	WIN
------	------	-----

### Change

A more explicit set of SQLSTATES (in the S1090 to S1110 range) has replaced the generic SQLSTATE S1009.

### Symptom

SQLSTATE values returned to the application calling DB2 CLI APIs have changed.

### Resolution

Update your application to check for the new SQLSTATES. Refer to the *Messages Reference* for a complete list of these SQLSTATES.

## DB2 CLI Mixing Embedded SQL, Without CONNECT RESET

UNIX	OS/2	WIN
------	------	-----

### Change

DB2 CLI's Version 2 support of multiple connections may affect your existing applications. If your application connects to a database using any non-CLI interface (including embedded SQL using the command line processor or administrative APIs) and does NOT issue a reset before connecting to a database using DB2 CLI, your applications will be affected by this change.

### Symptom

The second connect will fail with an SQLSTATE of 08001 since it is not same type of connection as the first connect.

### Resolution

The application must issue a CONNECT RESET before calling a DB2 CLI connect function.

## DB2 CLI Use of VARCHAR FOR BIT DATA

UNIX	OS/2	WIN
------	------	-----

### Change

Character data defined with the FOR BIT DATA clause is now by default mapped to the new C buffer type, SQL\_C\_BINARY. If data is defined as FOR BIT DATA, it is transferred to:

- SQL\_C\_BINARY buffers unchanged
- SQL\_C\_CHAR buffers as a character representation of the hexadecimal value of the data. Each byte is represented by two ASCII characters, (meaning the SQL\_C\_CHAR buffer must be double the size of the FOR BIT DATA string.)

### Symptom

Existing applications that explicitly use SQL\_C\_CHAR with data defined as FOR BIT DATA, will get a different result and may receive only half of the original data.

### Resolution

In order to have DB2 CLI treat FOR BIT DATA the same as it did in Version 1, add the following line to DB2CLI.INI:

```
BITDATA = 0
```

## DB2 CLI Data Conversion Values for SQLGetInfo

UNIX	OS/2	WIN
------	------	-----

### Change

The SQL\_CONVERT\_XXXX *flInfoType* is defined by ODBC to indicate supported conversion functions. A change has been made in how we handle SQL\_CONVERT\_XXXX *flInfoTypes* which were used with the corresponding SQL\_CVT\_XXX comparison masks to correctly follow ODBC standards.

### Symptom

DB2 CLI will no longer return bit masks for the SQL\_CONVERT\_XXX *flInfoTypes* and corresponding SQL\_CVT\_XXX comparison masks. DB2 CLI Version 2 now returns zero for all SQL\_CONVERT\_XXX *flInfoTypes*.

### Resolution

This is to correct Version 1 processing which was not ODBC compliant. There is no resolution.

## DB2 CLI/ODBC Configuration Keyword Defaults

UNIX	OS/2	WIN
------	------	-----

### Change

The default value for the CLI/ODBC configuration keyword DEFERREDPREPARE has changed. In DB2 CLI Version 5 deferred prepare is now on by default.

### Symptom

Applications that rely on the prepare to be executed as soon as it is issued will not function as expected. In particular, the row and cost estimates normally returned in the SQLERRD(3) and SQLERRD(4) of the SQLCA of a prepare statement may become



zeros. The application will not be able to use this information to decide whether or not to continue the execution of the SQL statement.

### Resolution

Add the following line to your *db2cli.ini* file:

```
DEFERREDPREPARE = 0
```

## Obsolete DB2 CLI/ODBC Configuration Keywords

UNIX	OS/2	WIN
------	------	-----

### Change

You can change the behavior of the DB2 CLI/ODBC driver by specifying a set of optional keywords in the *db2cli.ini* file. In Version 5, the AUTOCOMMIT keyword has become obsolete.

### Symptom

These keywords will be ignored if they still exist. You may notice behavioral changes based on the removal of these settings.

### Resolution

You will need to review the new list of valid parameters to decide what the appropriate keywords and settings are for your environment. See the *CLI Guide and Reference* for information on these keywords.

## DB2 CLI SQLSTATES

UNIX	OS/2	WIN
------	------	-----

### Change

The category of SQLSTATES that started with S1 in DB2 CLI Version 2 have been renamed to begin with HY in Version 5. For example, the SQLSTATE S1010 is now HY010.

### Symptom

SQLSTATE values returned to the application calling DB2 CLI APIs have changed.

### Resolution

Applications should be updated to expect the new HY class of SQLSTATES. Alternatively, the environment attribute SQL\_ATTR\_ODBC\_VERSION can be set to SQL\_OV\_ODBC2 using the DB2 CLI function SQLSetEnvAttr(). The DB2 CLI/ODBC driver will then return the S1 class of SQLSTATES.

## Stored Procedure Catalog Table

UNIX	OS/2	WIN
------	------	-----

### Change

Version 5 now has 2 system catalog views used to store information about all the stored procedures on the server (SYSCAT.PROCEDURES and SYSCAT.PROCPARMS). These replace the Version 2 pseudo catalog table for stored procedures

### Symptom

By default the server will look in the new system catalog views for information about stored procedures, not the older pseudo catalog table. DB2 CLI functions such as `SQLProcedureColumns()` and `SQLProcedures()` will therefore not return the appropriate information.

### Resolution

Register the stored procedures using the `CREATE PROCEDURE` SQL command. See the *SQL Reference* for more information. Alternatively, the DB2 CLI/ODBC configuration keyword `PATCH1` can be set to 262144 to force the DB2 CLI/ODBC driver to use the pseudo catalog table as it did in Version 2.

## PREP Command - LANGLEVEL

UNIX	OS/2	WIN
------	------	-----

### Change

When the `LANGLEVEL MIA` option of the `PREP` command is used, all C null-terminated strings are padded with blanks and the null-terminating character is placed in the last byte of the string.

### Symptom

Although this change was made for MIA compliance, it has caused a change to the way C null-terminated strings are handled.

### Resolution

There is another `LANGLEVEL` setting (`SAA1`) which may cause the behavior to better match your needs. You should review the options and decide what is best for your environment.

## Change to SMALLINT Constants

UNIX	OS/2	WIN
------	------	-----

### Change

Integer constants in the range -32,768 to 32,767 are now treated as INTEGER types, rather than SMALLINT. This resolves an incompatibility with IBM SQL, as well as simplifying the rules for determining literal types.

It is also worth mentioning that the smallest INTEGER constant in Version 1 (-2147483648) is a DECIMAL constant with a precision of 10 and a scale of 0 in Version 5.

Further, the smallest literal representation of a large INTEGER constant is -2147483647 and not -2147483648 (which is the limit for large INTEGER values). The INTEGER constant -2147483648 is a BIGINT, not a DECIMAL (as it was before Version 5.2).

In general, if an integer constant is outside the range of a large integer and within the range of a BIGINT, then it is a BIGINT. If it is too big for a BIGINT, then it is a DECIMAL.

### Symptom

This affects the result precision and scale of decimal operations. (Which impacts, for example, the print width of decimal fields.)

### Resolution

There is no resolution. This change in handling integers results in an increase in precision.

## Down-level Client and Distinct Types Sourced on BIGINT

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

A distinct type based on BIGINT in a Version 5.2 server is reported in a DESCRIBE to a down-level client as a DECIMAL(19,0) instead of as a BIGINT which is not supported by the client. This data type cannot be implicitly cast on assignment to the distinct type on which it is based. This is different than other situations where the client perceives a distinct type as a built-in data type and is able to assign host variables of the built-in type to columns of the associated distinct type.

### Symptom

An SQLCODE of -408 (SQLSTATE 42821) is returned when using a data type of DECIMAL(19,0) for the host variable (or parameter marker) assigned to the distinct type value that was described to the down-level client as DECIMAL(19,0).

### Resolution

The database should include a function that will cast a DECIMAL(19,0) to the distinct type. This can be defined as a sourced function based on the function that casts a BIGINT to the distinct type. The application (at the client) must then explicitly apply this function to the DECIMAL(19,0) host variable (or parameter marker) in the INSERT or UPDATE statement.

For example, if the distinct type sourced on BIGINT is called DT1, then updating the column C1 of type DT1 would require the following sourced function to be defined:

```
CREATE FUNCTION DT1(DECIMAL(19,0)) RETURNS DT1 SOURCE DT1(BININT);
```

And then the update statement in the application would be:

```
UPDATE table SET c1=DT1(:dechv1);
```

## Error Handling

UNIX	OS/2	WIN
------	------	-----

### Change

Errors which were previously reported at bind time may now not occur until statement execution. For instance, if you create a table using incorrect SQL syntax such as:

```
CREATE TABLE T1 (C1 CHAR(5), C1 CHAR(10))
```

The error that a duplicate column name was used will be flagged at run time instead of bind time. For all DDL statements, syntax errors are reported at bind time and semantic errors are reported at run time.

### Symptom

Some errors which were reported at bind time in Version 1 will now be reported at execution time.

### Resolution

As long as the application has proper error handling routines, this should not cause a problem. There will be some additional errors which can now occur during execution.

## Maximum Number of Sections in a Package

UNIX	OS/2	WIN
------	------	-----

### Change

The limit for the maximum number of sections in a package has changed from 400 to whatever the storage allows. This limit used to be hard-coded at 400, but now depends on the type of SQL statements in the program. As a result of this change, the constant for the maximum number of SQL statements has been removed from the common include files *sql.h*, *sql.cbl*, and *sql.f*.

### Symptom

If an application program references the following constants, it will not compile successfully in Version 5:

- SQL\_MAXSTMTS (in *sql.h*)
- SQL-MAXSTMTS (in *sql.cbl*)
- SQL\_MAXSTMTS (in *sql.f*)

### Resolution

Remove references to these constants or define them directly within your application.

## Bind Warnings

UNIX	OS/2	WIN
------	------	-----

### Change

Version 1 reports a warning at bind time if the number of host variables in an INTO clause is less than the number of select list items. Version 2 reports the same bind time warning if there are more or less host variables than select list items.

### Symptom

You will receive bind time warning messages where one was not received in Version 1.

### Resolution

Rebind the application with the new bind option SQLWARN NO and warnings will not be reported.

## Bind Options

UNIX	OS/2	WIN
------	------	-----

### Change

The new SQLWARN bind option has a default value of 'YES'.

### Symptom

By default, positive SQLCODEs may now be returned on DESCRIBE, PREPARE, and EXECUTE IMMEDIATE requests which were previously not returned. (For instance, a SQLCODE of +236 may be returned).

### Resolution

Rebind with SQLWARN NO if your application cannot tolerate positive SQLCODEs or treats them as errors.

## PREP with BINDFILE

UNIX	OS/2	WIN
------	------	-----

### Change

In Version 2, under certain circumstances, the DB2 PREP (precompile) command allows a bind file to be created even if certain errors occur. If the BINDFILE option, but not the PACKAGE option, is specified on the prep command, the following object existence and authority errors will be tolerated:

**SQL0117N** The number of values assigned is not the same as the number of specified or implied columns.

**SQL0204N** "<name>" is an undefined name.

**SQL0205N** "<name>" is not a column of table "<table-name>".

**SQL0206N** "<name>" is not a column in an inserted table, updated table, or any table identified in a FROM clause or is not a valid transition variable for the subject table of a trigger.

**SQL0440N** No function by the name "<function-name>" having compatible arguments was found in the function path.

**SQL0551N** "<authorization-ID>" does not have the privilege to perform operation "<operation>" on object "<name>".

**SQL0552N** "<authorization-ID>" does not have the privilege to perform operation "<operation>".

### Symptom

This may cause precompilation of some applications to succeed with errors where they failed in previous versions. The resultant bind file will fail if it is bound to a database with similar omissions of objects or authorities.

### Resolution

Check bind errors instead of precompiler errors for this condition.

## Varchar Structures in COBOL

UNIX	OS/2	WIN
------	------	-----

### Change

The COBOL precompiler in Version 2 and Version 5 supports declaration of group data items as host variables. (Refer to the *Embedded SQL Programming Guide* for more information.) This may cause some incompatibility with existing applications which did not adhere to the precise declaration format for VARCHAR host variables in COBOL.

The level numbers for subordinate items, as documented in DB2 manuals, must be 49. The following declaration would be accepted by the COBOL precompiler in Version 1, but not in Version 2 or Version 5:

```
01 MY-VAR.  
  10 MY-LENGTH PIC S9(4) COMP-5.  
  10 MY-DATA   PIC X(100).
```

If not coded correctly, the Version 2 and Version 5 precompiler will treat declarations like the above as structures with two members, a short integer and a fixed-length character string. When such a variable is used in an SQL statement, the reference to the would-be VARCHAR would be replaced with references to the two subordinates.

### Symptom

Depending on the situation, this may result in the following message:

SQL0087N Host variable "<name>" is a structure used where structure references are not permitted.

### Resolution

Applications being migrated to Version 5 that contain host variables which are intended to be VARCHARs should be declared with the subordinates at level 49.

### Incompatible APIs

UNIX	OS/2	
------	------	--

### Change

Several APIs have been changed or removed since Version 1. See the charts in the *API Reference* showing descriptions of the changes.

### Symptom

In most cases, the original API call will still work, however, you cannot take advantage of any of the new Version 5 functionality while using the old API calls or parameters.

### Resolution

Applications should be upgraded to use the new Version 5 API calls as described in the *API Reference*.

### Supported Level of JDBC

UNIX	OS/2	WIN
------	------	-----

### Change

The supported level of the JDBC (Java Support) API has changed. DB2 Version 5 provides a driver for JDBC 1.1 instead of JDBC 1.0, which came with DB2 Version 2.1.2.

### Symptom

Compiled JDBC 1.0 clients fail when executed directly as a DB2 Version 5 client. Old Java classes are not found.

### Resolution

To continue using JDBC 1.0 clients, run them on a DB2 Version 2.1.2 client, with a remote DB2 Version 5 server. Modify the client source code to upgrade to the JDBC 1.1 API. Run the JDBC 1.1 clients on a Java Development Kit Version 1.1-compatible virtual machine.

### Calling Convention for Java Stored Procedures and UDFs

UNIX	OS/2	WIN
------	------	-----

### Change

The calling convention for Java stored procedures and user-defined functions (UDFs) has changed in DB2 Version 5.

### Symptom

Java stored procedures and UDFs written for DB2 Version 2.1.2 will not be found when run on DB2 Version 5.

### Resolution

Change the Java stored procedure and UDF source code to use the new calling convention. Refer to the *Embedded SQL Programming Guide* for details.

## Java Runtime Environment

UNIX	OS/2	WIN
------	------	-----

### Change

The level of the Java runtime environment required for Java stored procedures, user-defined functions, and JDBC clients has changed in DB2 Version 5.

### Symptom

The JDBC DLL will not load when JDBC 1.1 clients are run. Java stored procedures and UDFs will fail.

### Resolution

Install a compatible Java 1.1 runtime environment at the client and server. At the server, set the `jdk11_path` configuration parameter.

## Obsolete System Monitor Requests for DB2 PE Version 1.2

			DB2 PE
--	--	--	--------

### Change

Some request types that were available with the DB2 PE Version 1.2 system monitor are no longer supported. See the tables in the *System Monitor Guide and Reference* showing descriptions of the changes.

### Symptom

The old request types will not work.

### Resolution

Applications should be upgraded to use the new DB2 Version 5 requests types as described in the *System Monitor Guide and Reference*.



---

## SQL

### Updating Partitioning Key Columns

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

#### Change

In DB2 PE Version 1.2, partitioning key columns could be updated if the table was in a single-node nodegroup. In DB2 Version 5, partitioning key columns can be updated if the table is in a table space in a single-node nodegroup, and there is no partitioning key defined.

#### Symptom

An update statement fails with SQL270N (SQLCODE -270, SQLSTATE 42997) with reason code 2. The same error is returned if the table is in a table space in a single or multiple node nodegroup.

#### Resolution

If the table is in a table space in a single node nodegroup, then use the ALTER TABLE statement to DROP the partitioning key. As with DB2 PE Version 1.2, if the table is in a table space in a multiple node nodegroup, the nodegroup must be changed to a single-node nodegroup and REDISTRIBUTE NODEGROUP must be issued before attempting to update partitioning key columns.

### Column NGNAME

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

#### Change

In DB2 PE Version 1.2, a table was directly associated with a nodegroup. In DB2 Version 5, a table is in a table space, which is within a nodegroup. Since there is no longer a direct relationship with a nodegroup, there is no need for a column, named NGNAME in the SYSIBM.SYSTABLES catalog table.

#### Symptom

An SQL statement that refers to the NGNAME column from SYSIBM.SYSTABLES catalog table will return an SQLCODE of -206 (SQLSTATE 42703).

#### Resolution

Remove the column NGNAME from the SQL statement. To determine the nodegroup name for the table, refer to NGNAME in the row of SYSCAT.TABLESPACES catalog view, that relates to the table space in which the table is stored.

## Node Number Temporary Space Usage

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

When using a temporary table that requires row identifiers, the amount of space needed is increased to include the node number. The space limit for temporary tables is 4005 bytes. If temporary tables are close to the 4005 byte limit, any further increase can exceed this limit.

### Symptom

There are two possible symptoms of this change.

- An SQL statement may fail to compile and return an SQLCODE of SQL0670N (SQLSTATE 54010).
- The temporary table is not used, which may affect the performance of the query.

### Resolution

You should review and use the directions in the Actions section of the message details for SQL0670N to fix the error.

## Authorities for Create and Drop Nodegroups

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

The authorization required for creating or dropping a nodegroup has changed from SYSADM or DBADM to SYSADM or SYSCTRL. This means that a user ID with DBADM authority cannot create, alter, or drop nodegroups.

### Symptom

A user ID, with DBADM authority, issuing a CREATE NODEGROUP or DROP NODEGROUP statement will receive an SQL00551N (SQLSTATE 42501).

### Resolution

Issue the statement using a user ID that has SYSADM or SYSCTRL authority. For your convenience, you may wish to include the user ID in the SYSCTRL group. Refer to the *Administration Guide* for further information.

## Target Map in REDISTRIBUTE NODEGROUP

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

The specification of a target map in the REDISTRIBUTE NODEGROUP command or API no longer causes database partitions to be implicitly added or dropped from the node group. This means that the target map cannot include nodes that are not defined to the node group. An undefined node that is included in the target map file will cause

an error to be returned. A database partition, which has been defined to the node group, can be excluded from the target map file and will not appear in the partition map.

### Symptom

If a node is included in the target map file and was not defined to the node group, the REDISTRIBUTE NODEGROUP command will return an SQLCODE–6053 with a reason code 6.

### Resolution

Before issuing the REDISTRIBUTE NODEGROUP command, add the database partition to the node group, using the ALTER NODEGROUP statement. You can also drop the node from the node group using the ALTER NODEGROUP statement, either before or after issuing the REDISTRIBUTE NODEGROUP command. Refer to the *SQL Reference* for further information on the ALTER NODEGROUP statement.

## Node Group for Create Table

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

In DB2 PE Version 1, a table was directly associated with a node group. In DB2 Version 5, a table is in a table space within a node group. When a user issues a CREATE TABLE statement, the name following the IN keyword is a table space name, not a node group name. The default table space selected may not be defined in the IBMDEFAULTGROUP node group, which was the default node group in DB2 PE Version 1.

### Symptom

If you use existing CREATE TABLE statements from DB2 PE Version 1, they may fail with an SQLCODE of SQL0204N (SQLSTATE 42704), with the name specified following the IN keyword in the message. This will occur if a table space with the same name as the node group has not been automatically created during database migration.

If you are using CREATE TABLE statements that do not specify the IN keyword, the table space selected, by default, may not be using the node group, IBMDEFAULTGROUP, and will not include data on all the database nodes. You can check the partition map for the table to confirm this.

### Resolution

Ensure that any name specified following the IN keyword on the CREATE TABLE statement is the name of a defined table space. For existing statements, you could set up a table space for each node group with the same name.

To ensure that tables default to the IBMDEFAULTGROUP for all users, define a table space called IBMDEFAULTGROUP, defined in the node group, IBMDEFAULTGROUP. This ensures that tables created by any users will default to use this table space.

**Note:** This is done automatically during database migration from DB2 PE Version 1 to DB2 Version 5.

## Revoking CONTROL on Tables or Views

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

A user can grant privileges on a table or view using the CONTROL privilege. In DB2 Version 5, the WITH GRANT OPTION provides a mechanism to determine a user's authorization to grant privileges on tables and views to other users. This mechanism is used in place of CONTROL to determine whether a user may grant privileges to others. When CONTROL is revoked, users will continue to be able to grant privileges to others.

### Symptom

A user can still grant privileges on tables or views, following the revocation of CONTROL privilege.

### Resolution

If a user should no longer be authorized to grant privileges on tables or views to others, revoke all privileges on the table or view and grant only those required.

## High Level Qualifiers for Objects in DB2 Version 5

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

In DB2 PE Version 1, users would create a table, view, index or package with any schema name or qualifier with the exception of SYSIBM. This differs from other IBM database products and is not compliant with SQL92. In DB2 Version 5, there are limits of the schema names that you can use.

- The schema names for tables, views, indexes, and packages cannot be SYSIBM, SYSCAT, SYSSTAT, OR SYSFUN.

**Note:** The schema names for all other objects must not start with SYS.

- Each schema is an object defined in the database catalog.

Users require IMPLICIT\_SCHEMA authority to implicitly create a schema. Once a schema is created, specific privileges allow users to create objects (CREATEIN privilege), drop any object in the schema (DROPIN privilege), or alter (comment on) any object in the schema (ALTERIN privilege). The change to supporting schemas, as objects with privileges, has resulted in changes to privileges associated with various statements.

- For creating objects in an existing schemas, you must have CREATEIN privilege.
- For creating objects in a schema that does not exist, you must have IMPLICIT\_SCHEMA authority.

- For dropping objects in a schema, you must be the definer of the object, have CONTROL privilege, or have DROPIN privilege on the schema.
- For altering, including commenting on, objects in a schema you must be the definer of the object, have CONTROL privilege, or have ALTERIN privilege on the schema.

**Note:** For altering or commenting on a table, the ALTER privilege on the table is also valid.

### Symptom

If you create an object with an invalid schema name, the CREATE statement returns an SQLCODE of SQL0553N. This message indicates that the object cannot be created with the schema name.

If a CREATE, ALTER, COMMENT ON or DROP statement returns an SQLCODE of SQL0551N, you did not have the necessary privilege. This may be the result of schema-related privileges and could indicate that:

- The object cannot be created because the schema does exist and you do not have the IMPLICIT\_SCHEMA authority.
- The object cannot be created because the schema does not exist and you do not have the CREATEIN privilege.
- The object cannot be dropped because another user created the object and you do not have the DROPIN privilege.
- The object cannot be altered (commented on) because another user created the object and you do not have ALTERIN privilege.

### Resolution

Depending on the symptom:

- Do not create schema names with SYS.
- If a user can create a table, view, index or package, grant the necessary authority or privilege using the GRANT (Database Authorities) statement for IMPLICIT\_SCHEMA authority, or the GRANT (Schema Privileges) statement for CREATEIN, DROPIN or ALTERIN privilege on the schema. A user with DBADM authority must first create the schema.

### Inoperative VIEWS

UNIX	OS/2	WIN
------	------	-----

### Change

In DB2 Version 2, a view is made inoperative if a SELECT privilege upon which the view definition is dependent is revoked or if an object upon which the view definition is dependent was dropped (or possibly made inoperative in the case of another view). This is in contrast to the behavior in DB2 Version 1 where the view would have been dropped under the same circumstances.

## Symptom

If the use of an inoperative VIEW is attempted, an SQL0575N will be returned to the application.

## Resolution

To resolve this problem, you will need to do two things:

1. Resolve the dependency (such as CREATE the dropped table).
2. Execute a CREATE VIEW.

Since the view is only inoperative and not dropped, you can query the TEXT column of SYSCAT.VIEWS to retrieve the current definition of the view.

## Unusable VIEWS

UNIX	OS/2	WIN
------	------	-----

## Change

If you currently have a view defined with SELECT \* on a table as part of the view definition, the view may be unusable after migration.

## Symptom

You will receive an SQL0158N error if you attempt to use a view that is unusable.

## Resolution

In order to resolve this problem you will need to:

1. Drop the existing view (DROP VIEW command).
2. Re-create the view (CREATE VIEW command), specifying column names in place of "\*" .

## SQLCODE Changes

UNIX	OS/2	WIN
------	------	-----

## Change

The SQLCODEs returned for an INSERT or UPDATE statement resulting in data being out of range have changed. These are:

- SQL0406N is now SQL0413N
- SQL0404N is now SQL0433N

The message has changed from "A numeric value/string in the UPDATE or INSERT statement is ..." to "Overflow occurred during numeric data type conversion". Note that there have been no changes to the corresponding SQLSTATES.

## Symptom

These SQLCODEs are caused by trying to place a value in a column that is outside a limit that exists on the data in that column. For applications, different values will now be

returned in SQLCA.SQLCODE. In any interactive situation (such as using the command line processor), a different error code will be reported to the user.

### Resolution

If your application specifically looks for the old SQLCODEs, you will need to change the comparison to use the new codes.

## WITH CHECK OPTION on CREATE VIEW

UNIX	OS/2	WIN
------	------	-----

### Change

The default used when WITH CHECK OPTION is specified without keywords has changed from LOCAL in Version 1 to CASCADED in Version 2.

### Symptom

This will cause the constraints of all dependant views to be applied.

### Resolution

Explicitly specify the LOCAL keyword with the WITH CHECK OPTION to get the same behavior as in Version 1.

## SQLSTATE Changes

UNIX	OS/2	WIN
------	------	-----

### Change

With DB2 Version 2, the SQLSTATEs have been updated to comply with the final published SQL92 standard.

### Symptom

In some cases, the value of SQLCA.SQLSTATE will be different than it would be in Version 1 for the same error or situation.

### Resolution

If your application is expecting a specific SQLSTATE, you may need to update the value in the comparison.

## FOR BIT DATA Comparisons

UNIX	OS/2	WIN
------	------	-----

### Change

In Version 1, all character strings, including FOR BIT DATA, were compared according to the database collating sequence. In Version 2, character strings with the FOR BIT DATA attribute will be compared according to their bit values, irrespective of the database collating sequence.

Whenever the database manager compares two character strings, if either comparand has the FOR BIT DATA attribute, the comparison is performed with the bit values of the comparands, without consideration of the database collating sequence. If the comparands are of different lengths, there is a logical blank padding (with X'20' on the right) of the shorter string to the length of the longer string.

### Symptom

Comparison results will differ from results in Version 1 when the collating sequence and the bit values are in different orders (only for FOR BIT DATA columns). For example, 'A' = x'41' and 'a' = x'61'. 'A' > 'a', however, x'41' < x'61'.

Keep in mind that comparisons take place in many situations including:

- Evaluation of basic predicates
- Use of the ORDER BY clause
- Use of the MIN and MAX column functions

### Resolution

You should replicate the data from the FOR BIT DATA column to a column with type CHAR. This will allow the data to be sorted according to the collating sequence instead of their bit values.

## Code Page Conversion

UNIX	OS/2	WIN
------	------	-----

### Change

Code page conversion rules for operands changed in Version 2. These changes improve DB2 compliance with SQL92 standards. It is important to understand that in most cases this will not affect result sets, however, it is possible to find scenarios where output would be different from DB2 Version 1 to Version 2 or to Version 5. In these cases, the output in Version 1 would be the incorrect output from the standpoint of the SQL92 standards compliance.

A few scenarios will be discussed where different output may be experienced:

- When using the LIKE predicate, it will always be the second operand which is converted to the first operand's code page.
- The result type for a UNION ALL set operation is determined in a binary fashion. For queries involving two or more UNION operations, and a mixture of fixed length and varying length character columns, intermediate fixed length datatypes may result in additional trailing blanks. If unequal code pages or columns defined FOR BIT DATA are part this type of operation, the conversion rules are applied to each intermediate result instead of using the final code page throughout the operation.
- The change to consistent conversion rules for result types means that the VALUE scalar function could have a result with a different code page than in previous versions of DB2.



### Symptom

The result set may be different since DB2 now adheres to SQL92 standards.

### Resolution

There is no resolution as this is an improvement for compliance with SQL92 standards.

## Isolation Levels and Blocking All

UNIX	OS/2	WIN
------	------	-----

### Change

When a cursor is declared without either the FOR UPDATE or FOR READ ONLY clause, it is considered to be an *ambiguous* cursor. If a package containing dynamically declared cursors is bound with the bind option BLOCKING=ALL, but without the bind option LANGLVL=MIA, then any ambiguous cursors will be treated as if FOR READ ONLY had been specified.

### Symptom

Your application may receive an SQLCODE of SQL0510N (SQLSTATE 42828) when performing a DELETE WHERE CURRENT OF CURSOR.

### Resolution

Rebind with the BLOCKING=UNAMBIG or LANGLVL=MIA options or add a FOR UPDATE clause to the cursor.

## ORDER BY Temporary Space Usage

UNIX	OS/2	WIN
------	------	-----

### Change

Whenever an ORDER BY is performed on a column which does not have an index, a temporary table is used to perform the sort. Beginning in Version 2, LONG VARCHAR and LONG VARGRAPHIC column types will use an increased amount of space as compared to Version 1 in these temporary tables. This may cause the query result rows to exceed the maximum row size (4005 bytes).

### Symptom

ORDER BY queries with one or more LONG VARCHAR (or LONG VARGRAPHIC) columns in the SELECT list and for which the select list is physically large, may fail to execute in Version 2 with SQLCODE SQL0670N (SQLSTATE 54010).

### Resolution

The following are some ways of attempting to resolve or avoid this scenario:

- Reduce the size of the SELECT list by removing some SELECT items (such as the LONG VARCHAR column(s))

- Apply the SUBSTR function to CHAR, GRAPHIC, VARCHAR, or VARGRAPHIC select items
- Create an index on the ORDER BY fields.

## Using Quotes in SQL Statements

UNIX	OS/2	WIN
------	------	-----

### Change

A defect in previous versions of DB2 allowed double quotes to be used in SQL statements as delimiters of some keywords and operators. For instance, though this is unpredictable, a query of the form *SELECT C1 "FROM" T1* was processed as if the *FROM* was not delimited.

Beginning in Version 2, this behaviour has been corrected.

### Symptom

SQL statements which incorrectly use double quotes to delimit keywords or operators will return errors during statement parsing.

### Resolution

The statement syntax should be changed to removed the unnecessary double quotes. For static SQL, if the application source code is unavailable, bind files can be carefully edited to remove the unnecessary quotes from the statements. Note that SQL identifiers may require the use of double quotes (these are called delimited identifiers).

---

## Database Security and Tuning

### GROUP Authorizations

UNIX	OS/2	
------	------	--

### Change

In DB2 Version 1, there was no way to indicate whether a privilege being granted was applicable to a user or to a group. In Version 2, a new field, called GRANTEETYPE, has been added to SYSCAT.DBAUTH, SYSCAT.INDEXAUTH, SYSCAT.PLANAUTH and SYSCAT.TABAUTH. GRANTEETYPE is either a 'U' to represent the GRANTEE is a user or 'G' to represent that the GRANTEE is a group.

During database migration, an attempt is made to determine whether existing privileges defined in the SYSIBM tables are for a user or a group. If the current privileges are for both a user and a group, only the user portion will be represented in the Version 2 database.

### Symptom

Loss of authorization if you are a member of a group which is also defined in the operating system as a user.

### Resolution

If this access is meant for groups (that is, where the environment variable DB2GROUPS=ON was used in Version 1), then execute the appropriate GRANT command for the appropriate access to the group.

### Authentication Type

UNIX	OS/2	WIN
------	------	-----

### Change

In Version 1, you could provide an authentication type on the CREATE DATABASE command. Beginning in Version 2, this option is ignored. All databases now have the same authentication type as the instance.

### Symptom

If the DB2 Version 5 instance authentication type is different than the Version 1 database authentication type, then authentication will behave differently after migration.

### Resolution

Make sure that the instance authentication type is the type you want for the databases within that instance.

### SYSADM Groups

UNIX	OS/2	
------	------	--

### Change

The SYSADM group must be explicitly set in the database manager configuration file.

### Symptom

This is automatically taken care of during migration, but a problem could arise if you use a script or command file to change SYSADM groups.

### Resolution

Update the script or command file to include the required changes in the database manager configuration file.

### Security Enhancements

UNIX	OS/2	
------	------	--

### Change

Several security enhancements have been made to the product to make Version 2 and following versions more secure than Version 1. A few of the changes are listed here, however, this is not a complete list.

- Authorization is no longer automatically inferred from file permissions (AIX).

- A general user can execute any DB2 executable, but will not be able to perform the function of that executable unless they have the correct authority. Examples include: db2start and db2trc. See the *Command Reference* for information on db2start and the *Troubleshooting Guide* for information on db2trc.
- Authorization for some commands, such as MIGRATE DATABASE, have changed. You should refer to the *Command Reference* and the *API Reference* for the authorization requirements for an individual command or API.

### Symptom

You may not be able to execute a DB2 command or API that you used to be able to execute. You will receive a “not authorized” type of SQLCODE.

### Resolution

Acquire the proper authorization for the task to be performed.

## Obsolete Profile Registry and Environment Variables

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

The following profile registry values or environment variables are obsolete:

- DB2THREADIF
- DB2\_INDEX\_FREE

### Resolution

There is no longer a need for this profile registry value. There is no need to disable DB2 support for multi-threaded applications.

---

## Utilities and Tools

### Executable Name Changes

	OS/2	
--	------	--

### Change

The following executables have changed names:

- STARTDBM.EXE is now DB2START.EXE
- STOPDBM.EXE is now DB2STOP.EXE
- SQLPREP.EXE is now the DB2 PREP command
- SQLBIND.EXE is now the DB2 BIND command
- SQLTRC.EXE is now DB2TRC.EXE
- EXPLAIN.EXE is now DB2EXPLN.EXE

### Symptom

The original executable names will still be accepted; however, some Version 2 functions are not available (such as new PREP and BIND options).

### Resolution

Use the Version 2 executables or commands.

## Backup and Restore - BUFF\_SIZE Parameter

UNIX	OS/2	
------	------	--

### Change

The parameter BUFF\_SIZE has changed for the backup and restore APIs. The minimum is now 16 allocation units (of 4K) instead of 8 units, and the increments must be in steps of 16 instead of 1.

### Symptom

You may receive a SQLCODE of SQL5130N.

### Resolution

Upgrade your application to use a BUFF\_SIZE value which is valid for Version 2.

## Backup and Restore - Changes Only Option

	OS/2	
--	------	--

### Change

In Version 1 there was the ability to backup and restore “Changes Only” to a database. This ability no longer exists. However, applications making the Version 1 API calls will not fail. DB2 simply ignores the second parameter (TYPE) in the sqluback() API call and performs a full backup.

### Symptom

A full backup will be taken when specifying a “Changes Only” backup.

### Resolution

None exist. “Changes Only” backups are no longer supported.

## Backup and Restore - User Exits

	OS/2	
--	------	--

### Change

Due to the table space capabilities available beginning in Version 2, it is no longer possible to determine the original location of the backup files. For this reason, user exits which use XCOPY or relied on the database sub-directory format in Version 1 will no longer function beginning in Version 2.

### Symptom

If you continue to use User Exits that move the backup files to another location, the restore may not function correctly.

### Resolution

User Exits can still be used for log archiving and retrieving. Use the supported parameters and options on the backup command to define the location the backup files will reside.

## Backup and Restore - Authority

UNIX	OS/2	
------	------	--

### Change

You must have SYSADM, SYSCTRL, or SYSMANT authority to use the BACKUP command. DBADM authority is no longer sufficient.

### Symptom

If you attempt a backup with DBADM authority only, you will be told that you do not have sufficient authority to perform the backup.

### Resolution

There are two choices:

1. Log on with an ID that has the proper authority.
2. Set the proper authority for the current ID.

## Import - IMPORT REPLACE Option

UNIX	OS/2	WIN
------	------	-----

### Change

A downlevel client cannot issue an IMPORT REPLACE command to a Version 2 server.

### Symptom

If this is attempted, the application will receive an SQL3188N error.

### Resolution

There are three possible resolutions to this scenario:

1. Upgrade the client to DB2 Version 5.
2. Execute this command from the DB2 Version 5 server.
3. Split the IMPORT REPLACE into two commands:
  - A DELETE from the table
  - An IMPORT INSERT into the table

## LOAD TERMINATE

			DB2 PE
--	--	--	--------

### Change

The LOAD TERMINATE command has a different function in DB2 UDB than it did in DB2 Parallel Edition Version 1.x. In DB2 Parallel Edition, you could use LOAD TERMINATE if an error occurred during the load operation to ensure that the table data was consistent. In DB2 UDB however, if you use LOAD TERMINATE, the table space is moved into the recovery pending state. (When the table space is in the recovery pending state, you must either restore the table space or drop it.)

### Symptom

Instead of being placed in a consistent state, the table space is placed in a recovery pending state.

### Resolution

Instead of using LOAD TERMINATE to clean up after a failed load operation, you should use LOAD RESTART or LOAD REPLACE. You also have the option of dropping and re-creating the table space.

## REORG - Alternate Path Option

UNIX	OS/2	
------	------	--

### Change

The REORG command and API no longer support an “alternate path” as a work area, but rather support the name of a table space to be used as a work area. APIs and commands will not fail, however, this option will be ignored.

### Symptom

REORG invocations from downlevel clients will ignore the alternate work path and arbitrarily choose a temporary table space to use as a work area.

Another symptom is you may run out of disk space.

### Resolution

Your applications will continue to function, but you should consider upgrading to the DB2 Version 5 calls which contain valid options.

---

## Connectivity and Coexistence

### Distributed Transaction Processing - Connect Type

UNIX	OS/2	
------	------	--

## Change

In an XA Distributed Transaction Processing environment, such as CICS, applications will always run with connect type 2 as the connection setting. In the last release, connect type 1 was used.

## Symptom

It will not be possible to modify the authorization ID on a database connection when the connection already exists.

## Resolution

Modification of the authorization ID on a database connection will have to be performed when the connection does not exist.

## Distributed Transaction Processing - SQLERRD Changes

UNIX	OS/2	
------	------	--

## Change

In an XA Distributed Transaction Processing environment such as CICS, information returned in SQLERRD after a CONNECT has changed. In Version 1, SQLERRD(6) was used to indicate one of the following:

- Non-XA
- DB2/6000 but not supporting XA
- DB2/6000 supporting XA

Beginning in Version 2, SQLERRD(6) is no longer used, but SQLERRD(3) and SQLERRD(4) are used as follows:

**SQLERRD(3)** Updateability in the unit of work

- Updateable
- Read Only

**SQLERRD(4)** Commit type

- One phase commit
- One phase commit, read only
- Two phase commit

## Symptom

The sixth SQLERRD element will no longer contain the information wanted by the application.

## Resolution

Change the application to look at the third and fourth SQLERRD fields.



## DDCS - SQLJSETP

	OS/2	
--	------	--

### Change

DDCS for OS/2 used to have a SQLJSETP environment variable. This item had two uses. Each is listed with the DDCS Version 2.3 and DB2 Connect replacement. (In DB2 Version 5, DDCS changed to DB2 Connect.)

1. By placing /s=e in the environment variable, bind files containing errors could be bound to DRDA servers. The default is to not allow errors. The /s=e function has been replaced by the SQLERROR CONTINUE bind/prepare option.

/s=c meant to prep/bind and perform syntax checking only without actually creating a package. This has been replaced by the SQLERROR CHECK bind/prepare option.

/s=n meant no errors were allowed during prep/bind. A package would only be created if there were no errors. This has been replaced by the SQLERROR NOPACKAGE prep/bind option.

2. This environment variable also captured prep/bind messages in a file for SQL statements which produced errors. This was needed because when the /s=e was specified, all errors were masked and missing from the precompiler generated message file. There is no longer a need for this because all messages are now revealed to the precompiler (and hence its message file) regardless of using SQLERROR CONTINUE or not.

### Symptom

The SQLJSETP environment variable and options are ignored, causing prep/bind to work according to their defaults.

### Resolution

Use the SQLERROR NOPACKAGE and/or SQLERROR CONTINUE options as needed.

## DDCS - DDCSSETP

UNIX		
------	--	--

### Change

The DDCSSETP utility has been removed. There is no longer a need for this because all messages are now revealed to the precompiler (and hence its message file) regardless of using SQLERROR CONTINUE or not. Refer to the prep/bind options discussed in "DDCS - SQLJSETP" for information.

## DDCS - SQLJTRC.CMD

	OS/2	
--	------	--

### Change

DDCS for OS/2 had a utility called `sqljtrc.cmd`. It has been replaced by the `ddcstrc.exe` executable. The invocation syntax has changed.

### Symptom

Attempting to trace DB2 Connect will fail if the old utility and parameters are used.

### Resolution

Execute the command `ddcstrc.exe` with valid parameters. See the *DB2 Connect User's Guide* for the new syntax.

## DDCS - SQLJBIND.CMD

	OS/2	
--	------	--

### Change

The DDCS for OS/2 utility called `sqljbind.cmd` has been removed.

### Symptom

Attempts to use this utility with DB2 Connect will fail.

### Resolution

This utility has been replaced by a three step set of instructions which are described in "Chapter 4, Binding Applications and Utilities" in the *DB2 Connect User's Guide*.

## APPC and APPN Nodes

	OS/2	
--	------	--

### Change

When using APPC, DB2 for OS/2 Version 1 supported the following commands for cataloging entries in the node directory:

- CATALOG APPN NODE
- CATALOG APPC NODE
- CATALOG NODE

Beginning in Version 2, support for the following has been removed:

- CATALOG APPN NODE
- CATALOG NODE

The CATALOG APPC NODE command has been changed to represent the APPC communication parameters required for the Communications Manager for OS/2 CPI Communications (CPIC) Side Information Profile.

The **symbolic destination name** parameter, in the CATALOG APPC NODE, contains the CPIC Side Information profile name in Communications Manager for OS/2. Refer to the *Command Reference* for details on this command.

## Symptom

All of the current connections using APPC will continue to work correctly and DB2 will accept the current catalog information after migration. This is true whether you migrate your server, clients or both.

The current catalogs cannot be modified. If you need to modify the information, you will have to UNCATALOG the node entry, and then recatalog using the new CATALOG APPC command.

If you have existing applications that call the CATALOG NODE API to catalog APPC and APPN node directory entries, these applications will still work. The CATALOG NODE API still supports the Version 1 APPC and APPN API structures.

## Resolution

When you try execute the CATALOG APPC node you will need to:

- Create a CPIC Side Information Profile.
- Reference the above profile in the CATALOG APPC NODE command.

Refer to the *DB2 Connect Personal Edition Quick Beginnings* for details on setting these nodes.

---

## Configuration Parameters

### ADSM\_PASSWORD

OS/2	WIN	UNIX	DB2 PE
------	-----	------	--------

## Change

In DB2 Version 5, ADSM\_PASSWORD is a database configuration parameter. In DB2 Version 2, it was a database manager configuration parameter.

**Note:** In DB2 PE Version 1, this parameter was spelled ADSM\_PASWD.

## Symptom

Any attempt to update or retrieve the DATABASE MANAGER CONFIGURATION value for ADSM\_PASSWORD will be a no-op; that is, no error or value will be returned.

## Resolution

You will have to set the ADSM\_PASSWORD for any databases for which you want to use the parameter.

### Agent Pool Size (NUM\_POOLAGENTS)

OS/2	WIN	UNIX	DB2 PE
------	-----	------	--------

## Change

In DB2 Version 5, NUM\_POOLAGENTS is used as a guideline for how large you want the agent pool to grow. It replaces the MAX\_IDLEAGENTS parameter used in DB2 Version 2. The agent pool contains subagents and idle agents. Idle agents can be used as parallel subagents or as coordinating agents.

## Resolution

You will need to replace the MAX\_IDLEAGENTS parameter with this new parameter.

## MAXDARI and MAXCAGENTS

OS/2	WIN	UNIX	DB2 PE
------	-----	------	--------

## Change

In Version 2, the value of the MAXDARI and MAXCAGENTS parameters were limited by the value of the MAXAGENTS configuration parameter. The default value of -1 means “equal to MAXAGENTS.”

Beginning in DB2 Version 5, the value of these two parameters are limited by the value of the MAX\_COORDAGENTS configuration parameter. The default value of -1 means “equal to MAX\_COORDAGENTS.”

**Note:** On non-Partitioned (non-MPP) configurations, the configuration parameter MAX\_COORDAGENTS can only have a value of -1, meaning “equal to MAXAGENTS.”

## Symptom

Updates to MAXDARI and MAXCAGENTS to values greater than -1 may fail if the value specified is greater than MAX\_COORDAGENTS.

## Resolution

Be aware of how MAX\_COORDAGENTS is set. MAXDARI and MAXCAGENTS cannot be greater than MAX\_COORDAGENTS.

## LOGFILSIZ

OS/2	WIN	UNIX	DB2 PE
------	-----	------	--------

## Change

The data type of this database configuration parameter has changed from being an unsigned 2-byte integer to an unsigned 4-byte integer. A new token has been added for the configuration APIs indicating a 4-byte integer.

For DB2 Version 5, the token is SQLF\_DBTN\_LOGFIL\_SIZ

For DB2 Version 2, the token is SQLF\_DBTN\_LOGFILSIZ

The configuration API will still recognize the Version 2 token, but the full range of values of this parameter is greater than what is supported by a 2-byte integer.

## Symptom

Existing applications will continue to work using the configuration API or via REXX, but the results might be unpredictable because of the larger range in DB2 Version 5.

## Resolution

Recode the application or REXX script to use the new token. For users of the Command Line Processor or the Control Center, this change in the token would not affect your applications.

## PCKCACHEFILSIZ

OS/2	WIN	UNIX	DB2 PE
------	-----	------	--------

## Change

The data type of this database configuration parameter has changed from being an unsigned 2-byte integer to an unsigned 4-byte integer. A new token has been added for the configuration APIs indicating a 4-byte integer.

For DB2 Version 5, the token is `SQLF_DBTN_PCKCACHE_SIZ`  
For Version 2, the token is `SQLF_DBTN_PCKCACHE_SZ`

In Version 2, the value of this parameter was limited by the size of the `APPLHEAPSZ` configuration parameter and indicated the size of a per-agent parameter. In DB2 Version 5, this parameter limits the size of a global per-database cache. Therefore, its value is no longer limited by the size of the `APPLHEAPSZ` configuration parameter.

In DB2 PE Version 1.2, the value of this parameter was limited by 7/8 of the value of the `DPHEAP` configuration parameter, since the cache was allocated from `DBHEAP`. In DB2 Version 5, the value of the cache is allocated out of its own heap. Therefore, the value of the `PCKCACHESZ` configuration parameter is no longer limited by the size of the `DBHEAP` configuration parameter.

## Symptom

The following might occur:

- Existing applications will continue to work using the configuration API or via REXX, but the results might be unpredictable because of the larger range in DB2 Version 5.
- Applications might get error code `SQL0973`, indicating that the `PCKCACHE` heap has been exhausted.

## Resolution

Depending on the symptoms, do one of the following:

- Recode the application to REXX script to use the new token.
- Check the settings of this parameter so that the application reflects the new value.

## APPLHEAPSZ and APP\_CTL\_HEAP\_SZ

OS/2	WIN	UNIX	DB2 PE
------	-----	------	--------

### Change

Beginning in DB2 Version 5, the use of these parameters has changed significantly.

### Symptom

Applications might receive an SQL0973 indicating that the APPLHEAP heap or APP\_CTL\_HEAP has been exhausted.

### Resolution

You will have to reconfigure these parameters for optimum performance. Refer to the *Administration Guide* and the online help for the Control Center for recommendations on tuning these parameters.

## BUFFPAGE and Multiple Buffer Pools

UNIX	OS/2	WIN	DB2 PE
------	------	-----	--------

### Change

In previous versions of DB2, each database had one buffer pool, which was created when the database was created. You could change the size of the buffer pool using the *buffpage* parameter. In DB2 Version 5, each database can have multiple buffer pools. You can create additional buffer pools or change the size of a buffer pool through the CREATE BUFFERPOOL or ALTER BUFFERPOOL statements or through the Control Center using the appropriate command.

If the buffer pool size is specified to be -1, then the value of the database configuration parameter is used as the size of the buffer pool.

**Note:** When the BUFFPAGE database configuration parameter is updated, you will receive an SQLCODE SQL1482W warning.

### Symptom

In DB2 Version 5, a new or migrated database has a default buffer pool. For a new database created in DB2 Version 5, the size of the default buffer pool is determined by the operating system. For a migrated database, the size of the buffer pool is set to -1, which then refers to the *buffpage* configuration parameter.

### Resolution

To resolve this problem, you will need to do the following:

1. For a new database created in DB2 Version 5, you may change the size of the buffer pool using the ALTER BUFFERPOOL statement.
2. Following the creation or migration of a database, you can then create additional buffer pools for the database using the CREATE BUFFERPOOL statement.

## NEWLOGPATH

OS/2	WIN	UNIX	DB2 PE
------	-----	------	--------

### Change

In DB2 Version 5, in a partitioned database, the node number is appended to the path in the form `path_name \NODExxxx` (`path_name /NODExxxx` on UNIX-based systems), where `xxxx` is the 4 digit node number. This maintains the uniqueness of the path across the database partitions.

### Symptom

When updating the NEWLOGPATH configuration parameter, the node number is automatically appended to the path name. This may result in path names that are too long (greater than 242 characters), and the configuration parameter update may fail.

### Resolution

Be aware that the log files will reside in the path that includes the node numbering designation. If the configuration parameter update failed, ensure that the path length, including the node number designation, is less than or equal to 242 characters.

## MULTIPAGE\_ALLOC

			DB2 PE
--	--	--	--------

### Change

In DB2 PE Version 1.2, this database configuration parameter was known as MULTIPGAL and the data type of this database configuration parameter was an unsigned 1-byte integer. In DB2 Version 5, the data type of this parameter is an unsigned 2-byte integer, using a new token.

For DB2 Version 5, the token is `SQLF_DBTN_MULTIPAGE_ALLOC`

For DB2 PE Version 1, the token is `SQLF_DBTN_MULTIPGAL`

### Symptom

Existing applications will continue to work using either the `SQLF_DBTN_MULTIPGAL` or the `SQLF_DBNR_MULTIPAGE_ALLOC` tokens.

### Resolution

While the configuration APIs support both tokens, applications should be updated to use the new tokens.

## EXTENTSIZ vs SEGPAGES

UNIX		
------	--	--

### Change

Beginning in Version 2, new `dft_extent_sz` configuration parameter serves as the default EXTENTSIZ setting for table spaces where this is not specified.

- default value: 32 4K pages
- range: 2-256 4K pages

It is modifiable.

### Symptom

If an application attempts to specify the SEGPAGES parameter in the CREATE DATABASE command, the command will still work; however, the parameter will be ignored. The EXTENTSIZE will be set to the default.

### Resolution

Update the command to specify the new EXTENTSIZE parameter when creating a table space.

## LOCKLIST

UNIX	OS/2	
------	------	--

### Change

In DB2 Version 5, the size of a lock request block has been changed to 36 bytes. As a result, fewer lock request blocks will fit in the configured amount of space allocated for the lock list.

### Symptom

This may result in more frequent lock escalations.

### Resolution

You should increase the setting of the LOCKLIST configuration parameter accordingly.

## BUFFPAGE and SORTHEAP

UNIX	OS/2	
------	------	--

### Change

The tokens for database configuration parameters buffpage and sortheap have changed.

For buffpage:

from SQLF\_DBTN\_BUFFPAGE to SQLF\_DBTN\_BUFF\_PAGE

For sortheap on OS/2:

from SQLF\_DBTN\_SORTHEAP to SQLF\_DBTN\_SORT\_HEAP

For sortheap on AIX:

from SQLF\_DBTN\_SORTHEAPSZ\_P to SQLF\_DBTN\_SORT\_HEAP

The names of the parameters as identified in command line processor or in the Control Center remain the same (buffpage and sortheap). The old tokens are maintained for backlevel binary compatibility.



On AIX, the configuration APIs treat the new token and the old token as indicating a 32 byte unsigned integer. On OS/2 however, the configuration APIs will treat the old token as indicating a 16 byte unsigned integer. This is consistent with Version 1 behavior. The new tokens will be treated as indicating an unsigned 32 byte integer.

**Symptom**

Version 1 applications which specify the old token names will not work against a Version 2 or later database.

**Resolution**

In order to migrate old application code the token names need to be changed. Additionally, on OS/2, the data type of the variable being passed to the configuration APIs will have to be changed to an unsigned 32 byte integer.

**Numeric Values for Database Manager Configuration Tokens**

UNIX		
------	--	--

**Change**

In DB2 for AIX Version 1, the numerical values for the database manager configuration parameter tokens SQLF\_KTN\_MAXDARI and SQLF\_KTN\_KEEPPARI were 22 and 23 respectively. Beginning in Version 2, they are 80 and 81 respectively. Binaries from Version 1 will be supported despite this discrepancy.

**Symptom**

Applications which perform a DATABASE MANAGER CONFIGURATION operation and specify the changed parameters by explicitly stating their numeric values will no longer work as desired.

**Resolution**

If code is being migrated and the token name is used, nothing needs to be changed. If however, the token values were coded explicitly in the application, the application will have to be changed to reflect the new values.

To protect the application from future changes of this type, it is recommended that the token is coded, rather than the actual value.

**Numeric Values for Database Manager Configuration Tokens**

	OS/2	
--	------	--

**Change**

In DB2 for OS/2 Version 1.2, the numerical values for the database manager configuration parameter tokens SQLF\_KTN\_FILESERVER and SQLF\_KTN\_OBJECTNAME were 22 and 23 respectively. Beginning in Version 2, they are 47 and 48 respectively. Binaries from Version 1 will be supported despite this discrepancy.

## Symptom

Applications which perform a DATABASE MANAGER CONFIGURATION operation and specify the changed parameters by explicitly stating their numeric values will no longer work as desired.

## Resolution

If code is being migrated and the token name is used, nothing need be changed. If however, the token values were coded explicitly in the application, the application will have to be changed to reflect the new values.

To protect the application from future changes of this type, it is recommended that the token is coded, rather than the actual value.

## New Generic Out-of-Range Return Codes

UNIX	OS/2	WIN
------	------	-----

## Change

Many return codes indicating an attempt to set a specific parameter outside of its valid range were replaced with generic out-of-range return codes.

The following return codes have been replaced with a return code of -5130 (SQLF\_RC\_INV\_RANGE as defined in sqlutil.h):

- SQLF\_RC\_INVDB
- SQLF\_RC\_INVRIO
- SQLF\_RC\_INVSHPTH
- SQLF\_RC\_INVNULL
- SQLF\_RC\_INVNDBF
- SQLF\_RC\_INVSCP
- SQLF\_RC\_INVNAP
- SQLF\_RC\_INVAHP
- SQLF\_RC\_INVDP
- SQLF\_RC\_INVDLT
- SQLF\_RC\_INVTAF
- SQLF\_RC\_INVSH
- SQLF\_RC\_INVMAL
- SQLF\_RC\_INVSTMTHP
- SQLF\_RC\_INVLOGPRIM
- SQLF\_RC\_INVLOG2ND
- SQLF\_RC\_INVLOGFSZ
- SQLF\_RC\_INVNBP

and SQLF\_RC\_INV\_DBMENT (-5126) is returned, beginning in Version 2, instead of SQLF\_RC\_INVK3 (-5105) which is no longer returned.

## Symptom

If an application is looking for a specific error code which has been replaced by a new one, then this will cause the application to function incorrectly.

### Resolution

Update the application to look for valid return codes.

### Segments versus 4KB Pages

	OS/2	
--	------	--

### Change

All configuration parameters in OS/2 that were expressed in segments in Version 1 are now expressed in 4KB pages.

### Symptom

Beginning in Version 2, when you specify a configuration parameter which used to be a measure of segments, it is treated as a measure of 4KB pages. This will result in a different total amount of space in most cases.

### Resolution

Migration takes care of this incompatibility by allocating the same amount of storage that was allocated before the migration. Existing applications that specify parameter values should be converted to specify the proper number of 4KB page units.

### Obsolete Database Configuration Parameters

	OS/2	
--	------	--

### Change

The following database configuration parameters are obsolete:

- AGENTHEAP
- MAXTOTFILOP (there is now a new database manager level configuration parameter by the same name)
- SQLSTMTSZ

Version 1 binary applications attempting to update or get the value of these parameters will result in a no-operation with a return code of 0.

### Resolution

Applications should be updated to not reference these parameters.

If you are updating or viewing the value for MAXTOTFILOP, then you can now use Database Manager Configuration commands.

### Obsolete Database Manager Configuration Parameters

UNIX	OS/2	
------	------	--

## Change

The following database manager configuration parameters are obsolete:

- MAX\_IDLEAGENTS
- COMHEAPSZ
- RSHEAPSZ
- SVRIOBLK
- NUMRC
- SQLENSSEG (OS/2 only)
- CUINTERVAL

## Symptom

Version 1 binary applications attempting to update or get the value of these parameters will result in a no-operation with a return code of 0.

## Resolution

Applications should be updated to not reference these parameters.

## DB2\_MMAP\_READ and DB2\_MMAP\_WRITE

			DB2 PE
--	--	--	--------

## Change

After you migrate to DB2 UDB from DB2 Parallel Edition Version 1.x, you may, in some situations, notice a performance degradation compared to DB2 Parallel Edition. This is caused by a change in the default value of the DB2\_MMAP\_READ and DB2\_MMAP\_WRITE profile registry values. (These were known as environment variables in DB2 Parallel Edition.) The default value of these registry variables in DB2 Parallel Edition was "OFF." This allowed AIX to cache DB2 data that was read from JFS filesystems into memory (that is, the data was outside the buffer pool). In DB2 UDB, as in the case of DB2 Version 2, the default value of these registry values is "ON" which prevents the AIX caching.

## Symptom

In DB2 UDB, the problem occurs when accessing DB2 data that was previously referenced but is no longer in the DB2 buffer pool. Because the DB2 buffer pool is relatively small, in DB2 UDB such access may require disk I/O. In DB2 Parallel Edition, the request to access the data may have been satisfied by the AIX cache which is much faster.

## Resolution

In situations like the one mentioned above, setting DB2\_MMAP\_READ and DB2\_MMAP\_WRITE to "OFF" in DB2 UDB may result in queries running up to three times faster. However, you should devote a large amount of system memory to the DB2 buffer pool. If you do this, AIX caching provides no additional benefit.

---

## Appendix K. Explain Tables and Definitions

The Explain tables capture access plans when the Explain facility is activated. The following Explain tables and definitions are described in this section:

- “EXPLAIN\_ARGUMENT Table”
- “EXPLAIN\_INSTANCE Table” on page 939
- “EXPLAIN\_OBJECT Table” on page 940
- “EXPLAIN\_OPERATOR Table” on page 942
- “EXPLAIN\_PREDICATE Table” on page 944
- “EXPLAIN\_STATEMENT Table” on page 946
- “EXPLAIN\_STREAM Table” on page 948

The Explain tables must be created before Explain can be invoked. To create them, use the sample command line processor input script provided in the EXPLAIN.DDL file located in the 'misc' subdirectory of the 'sqllib' directory. Connect to the database where the Explain tables are required. Then issue the command: `db2 -tf EXPLAIN.DDL` and the tables will be created. See “Table Definitions for Explain Tables” on page 949 for more information.

The population of the Explain tables by the Explain facility will neither activate any triggers nor activate any referential or check constraints. For example, if an insert trigger were defined on the EXPLAIN\_INSTANCE table and an eligible statement were explained, the trigger would not be activated.

For more details on the Explain facility, see the *Administration Guide*.

### Legend for the Explain Tables:

<b>Heading</b>	<b>Explanation</b>
Column name	Name of the column
Data Type	Data type of the column
Nullable?	Yes: Nulls are permitted No: Nulls are not permitted
Key?	PK: Column is part of a primary key FK: Column is part of a foreign key
Description	Description of the column

---

### EXPLAIN\_ARGUMENT Table

The EXPLAIN\_ARGUMENT table represents the unique characteristics for each individual operator, if there are any.

For the definition of this table, see “EXPLAIN\_ARGUMENT Table Definition” on page 950.

## Explain Tables

Table 96. EXPLAIN\_ARGUMENT Table

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	CHAR(8)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	CHAR(8)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	CHAR(8)	No	FK	Schema, or qualifier, of source of Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	SMALLINT	No	FK	Statement number within package to which this explain information is related.
SECTNO	SMALLINT	No	FK	Section number within package to which this explain information is related.
OPERATOR_ID	SMALLINT	No	No	Unique ID for this operator within this query.
ARGUMENT_TYPE	CHAR(8)	No	No	The type of argument for this operator.
ARGUMENT_VALUE	VARCHAR(30)	No	No	The value of the argument for this operator.

Table 97 (Page 1 of 3). ARGUMENT\_TYPE and ARGUMENT\_VALUE Column Values

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
AGGMODE	COMPLETE PARTIAL INTERMEDIATE FINAL	Partial aggregation indicators.
BUFFERS	INTEGER	Buffers consumed.
BITFLTR	TRUE FALSE	Hash Join will use a bit filter to enhance performance.
CSETEMP	TRUE FALSE	Temporary Table over Common Subexpression Flag.
DIRECT	TRUE	Direct fetch indicator.
DUPLWARN	TRUE FALSE	Duplicates Warning flag.
EARLYOUT	TRUE FALSE	Early out indicator.
FETCHMAX	IGNORE INTEGER	Override value for MAXPAGES argument on FETCH operator.
GROUPBYC	TRUE FALSE	Whether Group By columns were provided.
GROUPBYN	Integer	Number of comparison columns.
GROUPBYR	Each row of this type will contain: <ul style="list-style-type: none"> <li>Ordinal value of column in group by clause (followed by a colon and a space)</li> <li>Name of Column</li> </ul>	Group By requirement.

## Explain Tables

Table 97 (Page 2 of 3). ARGUMENT\_TYPE and ARGUMENT\_VALUE Column Values

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
INNERCOL	Each row of this type will contain: <ul style="list-style-type: none"> <li>Ordinal value of column in order (followed by a colon and a space)</li> <li>Name of Column</li> <li>Order Value <ul style="list-style-type: none"> <li>(A) Ascending</li> <li>(D) Descending</li> </ul> </li> </ul>	Inner order columns.
ISCANMAX	IGNORE INTEGER	Override value for MAXPAGES argument on ISCAN operator.
JN_INPUT	INNER OUTER	Indicates if operator is the operator feeding the inner or outer of a join.
LISTENER	TRUE FALSE	Listener Table Queue indicator.
MAXPAGES	ALL NONE INTEGER	Maximum pages expected for Prefetch.
MAXRIDS	NONE INTEGER	Maximum Row Identifiers to be included in each list prefetch request.
NUMROWS	INTEGER	Number of rows expected to be sorted.
ONEFETCH	TRUE FALSE	One Fetch indicator.
OUTERCOL	Each row of this type will contain: <ul style="list-style-type: none"> <li>Ordinal value of column in order (followed by a colon and a space)</li> <li>Name of Column</li> <li>Order Value <ul style="list-style-type: none"> <li>(A) Ascending</li> <li>(D) Descending</li> </ul> </li> </ul>	Outer order columns.
OUTERJN	LEFT RIGHT	Outer join indicator.
PARTCOLS	Name of Column	Partitioning columns for operator.
PREFETCH	LIST NONE SEQUENTIAL	Type of Prefetch Eligible.
ROWLOCK	EXCLUSIVE NONE REUSE SHARE SHORT (INSTANT) SHARE UPDATE	Row Lock Intent.
ROWWIDTH	INTEGER	Width of row to be sorted. ***
SCANDIR	FORWARD REVERSE	Scan Direction.
SCANGRAN	INTEGER	Intra-partition parallelism, granularity of the intra-partition parallel scan, expressed in SCANUNITS.
SCANTYPE	LOCAL PARALLEL	intra-partition parallelism, Index or Table scan.

## Explain Tables

Table 97 (Page 3 of 3). ARGUMENT\_TYPE and ARGUMENT\_VALUE Column Values

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
SCANUNIT	ROW PAGE	Intra-partition parallelism, scan granularity unit.
SHARED	TRUE	Intra-partition parallelism, shared TEMP indicator.
SLOWMAT	TRUE FALSE	Slow Materialization flag.
SNGLPROD	TRUE FALSE	Intra-partition parallelism sort or temp produced by a single agent.
SORTKEY	Each row of this type will contain: <ul style="list-style-type: none"> <li>• Ordinal value of column in key (followed by a colon and a space)</li> <li>• Name of Column</li> <li>• Order Value <ul style="list-style-type: none"> <li>(A) Ascending</li> <li>(D) Descending</li> </ul> </li> </ul>	Sort key columns.
SORTTYPE	PARTITIONED SHARED ROUND ROBIN REPLICATED	Intra-partition parallelism, sort type.
TABLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Table Lock Intent.
TQDEGREE	INTEGER	intra-partition parallelism, number of subagents accessing Table Queue.
TQMERGE	TRUE FALSE	Merging (sorted) Table Queue indicator.
TQREAD	READ AHEAD STEPPING SUBQUERY STEPPING	Table Queue reading property.
TQSEND	BROADCAST DIRECTED SCATTER SUBQUERY DIRECTED	Table Queue send property.
TQTYPE	LOCAL	Intra-partition parallelism, Table Queue.
TRUNCSRT	TRUE	Truncated sort (limits number of rows produced).
UNIQUE	TRUE FALSE	Uniqueness indicator.
UNIQKEY	Each row of this type will contain: <ul style="list-style-type: none"> <li>• Ordinal value of column in key (followed by a colon and a space)</li> <li>• Name of Column</li> </ul>	Unique key columns.



### EXPLAIN\_INSTANCE Table

The EXPLAIN\_INSTANCE table is the main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table. The EXPLAIN\_INSTANCE table gives basic information about the source of the SQL statements being explained as well as information about the environment in which the explanation took place.

For the definition of this table, see “EXPLAIN\_INSTANCE Table Definition” on page 951.

Table 98 (Page 1 of 2). EXPLAIN\_INSTANCE Table

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	CHAR(8)	No	PK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK	Time of initiation for Explain request.
SOURCE_NAME	CHAR(8)	No	PK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	CHAR(8)	No	PK	Schema, or qualifier, of source of Explain request.
EXPLAIN_OPTION	CHAR(1)	No	No	Indicates what Explain Information was requested for this request.  Possible values are: <b>P</b> PLAN SELECTION
SNAPSHOT_TAKEN	CHAR(1)	No	No	Indicates whether an Explain Snapshot was taken for this request.  Possible values are: <b>Y</b> Yes, an Explain Snapshot(s) was taken and stored in the EXPLAIN_STATEMENT table. Regular Explain information was also captured. <b>N</b> No Explain Snapshot was taken. Regular Explain information was captured. <b>O</b> Only an Explain Snapshot was taken. Regular Explain information was not captured.
DB2_VERSION	CHAR(7)	No	No	Product release number for DB2 Universal Database which processed this explain request. Format is vv.rr.m, where: <b>vv</b> Version Number <b>rr</b> Release Number <b>m</b> Maintenance Release Number
SQL_TYPE	CHAR(1)	No	No	Indicates whether the Explain Instance was for static or dynamic SQL.  Possible values are: <b>S</b> Static SQL <b>D</b> Dynamic SQL
QUERYOPT	INTEGER	No	No	Indicates the query optimization class used by the SQL Compiler at the time of the Explain invocation. The value indicates what level of query optimization was performed by the SQL Compiler for the SQL statements being explained.

## Explain Tables

Table 98 (Page 2 of 2). EXPLAIN\_INSTANCE Table

Column Name	Data Type	Nullable?	Key?	Description
BLOCK	CHAR(1)	No	No	Indicates what type of cursor blocking was used when compiling the SQL statements. For more information, see the BLOCK column in SYSCAT.PACKAGES.  Possible values are: <b>N</b> No Blocking <b>U</b> Block Unambiguous Cursors <b>B</b> Block All Cursors
ISOLATION	CHAR(2)	No	No	Indicates what type of isolation was used when compiling the SQL statements. For more information, see the ISOLATION column in SYSCAT.PACKAGES.  Possible values are: <b>RR</b> Repeatable Read <b>RS</b> Read Stability <b>CS</b> Cursor Stability <b>UR</b> Uncommitted Read
BUFFPAGE	INTEGER	No	No	Contains the value of the BUFFPAGE database configuration setting at the time of the Explain invocation.
AVG_APPLS	INTEGER	No	No	Contains the value of the AVG_APPLS configuration parameter at the time of the Explain invocation.
SORTHEAP	INTEGER	No	No	Contains the value of the SORTHEAP database configuration setting at the time of the Explain invocation.
LOCKLIST	INTEGER	No	No	Contains the value of the LOCKLIST database configuration setting at the time of the Explain invocation.
MAXLOCKS	SMALLINT	No	No	Contains the value of the MAXLOCKS database configuration setting at the time of the Explain invocation.
LOCKS_AVAIL	INTEGER	No	No	Contains the number of locks assumed to be available by the optimizer for each user. (Derived from LOCKLIST and MAXLOCKS.)
CPU_SPEED	DOUBLE	No	No	Contains the value of the CPUSPEED database manager configuration setting at the time of the Explain invocation.
REMARKS	VARCHAR(254)	Yes	No	User-provided comment.
DBHEAP	INTEGER	No	No	Contains the value of the DBHEAP database configuration setting at the time of Explain invocation.
COMM_SPEED	DOUBLE	No	No	Contains the value of the COMM_BANDWIDTH database configuration setting at the time of Explain invocation.
PARALLELISM	CHAR(2)	No	No	Possible values are: <b>N</b> No parallelism <b>P</b> Intra-partition parallelism <b>IP</b> Inter-partition parallelism <b>BP</b> Intra-partition parallelism and inter-partition parallelism

### EXPLAIN\_OBJECT Table

The EXPLAIN\_OBJECT table identifies those data objects required by the access plan generated to satisfy the SQL statement.

## Explain Tables

For the definition of this table, see “EXPLAIN\_OBJECT Table Definition” on page 952.

Table 99 (Page 1 of 2). EXPLAIN\_OBJECT Table

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	CHAR(8)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	CHAR(8)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	CHAR(8)	No	FK	Schema, or qualifier, of source of Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	SMALLINT	No	FK	Statement number within package to which this explain information is related.
SECTNO	SMALLINT	No	FK	Section number within package to which this explain information is related.
OBJECT_SCHEMA	CHAR(8)	No	No	Schema to which this object belongs.
OBJECT_NAME	VARCHAR(18)	No	No	Name of the object.
OBJECT_TYPE	CHAR(2)	No	No	Descriptive label for the type of object.
CREATE_TIME	TIMESTAMP	Yes	No	Time of Object's creation; null if a table function.
STATISTICS_TIME	TIMESTAMP	Yes	No	Last time of update to statistics for this object; null if statistics do not exist for this object.
COLUMN_COUNT	SMALLINT	No	No	Number of columns in this object.
ROW_COUNT	INTEGER	No	No	Estimated number of rows in this object.
WIDTH	INTEGER	No	No	The average width of the object in bytes. Set to -1 for an index.
PAGES	INTEGER	No	No	Estimated number of pages that the object occupies in the buffer pool. Set to -1 for a table function.
DISTINCT	CHAR(1)	No	No	Indicates if the rows in the object are distinct (i.e. no duplicates)  Possible values are: <b>Y</b> Yes <b>N</b> No
TABLESPACE_NAME	VARCHAR(18)	Yes	No	Name of the table space in which this object is stored; set to null if no table space is involved.
OVERHEAD	DOUBLE	No	No	Total estimated overhead, in milliseconds, for a single random I/O to the specified table space. Includes controller overhead, disk seek, and latency times. Set to -1 if no table space is involved.
TRANSFER_RATE	DOUBLE	No	No	Estimated time to read a data page, in milliseconds, from the specified table space. Set to -1 if no table space is involved.
PREFETCHSIZE	INTEGER	No	No	Number of data pages to be read when prefetch is performed. Set to -1 for a table function.
EXTENTSIZE	INTEGER	No	No	Size of extent, in data pages. This many pages are written to one container in the table space before switching to the next container. Set to -1 for a table function.

## Explain Tables

Table 99 (Page 2 of 2). EXPLAIN\_OBJECT Table

Column Name	Data Type	Nullable?	Key?	Description
CLUSTER	DOUBLE	No	No	Degree of data clustering with the index. If $\geq 1$ , this is the CLUSTERRATIO. If $\geq 0$ and $< 1$ , this is the CLUSTERFACTOR. Set to -1 for a table, table function, or if this statistic is not available.
NLEAF	INTEGER	No	No	Number of leaf pages this index object's values occupy. Set to -1 for a table, table function, or if this statistic is not available.
NLEVELS	INTEGER	No	No	Number of index levels in this index object's tree. Set to -1 for a table, table function, or if this statistic is not available.
FULLKEYCARD	INTEGER	No	No	Number of distinct full key values contained in this index object. Set to -1 for a table, table function, or if this statistic is not available.
OVERFLOW	INTEGER	No	No	Total number of overflow records in the table. Set to -1 for an index, table function, or if this statistic is not available.

Table 100. Possible OBJECT\_TYPE Values

Value	Description
IX	Index
TA	Table
TF	Table Function

## EXPLAIN\_OPERATOR Table

The EXPLAIN\_OPERATOR table contains all the operators needed to satisfy the SQL statement by the SQL compiler.

For the definition of this table, see "EXPLAIN\_OPERATOR Table Definition" on page 953.

Table 101 (Page 1 of 2). EXPLAIN\_OPERATOR Table

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	CHAR(8)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	CHAR(8)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	CHAR(8)	No	FK	Schema, or qualifier, of source of Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	SMALLINT	No	FK	Statement number within package to which this explain information is related.
SECTNO	SMALLINT	No	FK	Section number within package to which this explain information is related.
OPERATOR_ID	SMALLINT	No	No	Unique ID for this operator within this query.
OPERATOR_TYPE	CHAR(6)	No	No	Descriptive label for the type of operator.

## Explain Tables

Table 101 (Page 2 of 2). EXPLAIN\_OPERATOR Table

Column Name	Data Type	Nullable?	Key?	Description
TOTAL_COST	DOUBLE	No	No	Estimated cumulative total cost (in instructions) of executing the chosen access plan up to and including this operator.
IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of executing the chosen access plan up to and including this operator.
CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in instructions) of executing the chosen access plan up to and including this operator.
FIRST_ROW_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the first row for the access plan up to and including this operator. This value includes any initial overhead required.
RE_TOTAL_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the next row for the chosen access plan up to and including this operator.
RE_IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of fetching the next row for the chosen access plan up to and including this operator.
RE_CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in timerons) of fetching the next row for the chosen access plan up to and including this operator.
COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost (in TCP/IP frames) of executing the chosen access plan up to and including this operator.
FIRST_COMM_COST	DOUBLE	No	No	Estimated cumulative communications cost (in TCP/IP frames) of fetching the first row for the chosen access plan up to and including this operator. This value includes any initial overhead required.
NODES_USED	CLOB(64K)	Yes	No	Cumulative list of nodes involved in executing the chosen access plan up to and including this operator.

Table 102 (Page 1 of 2). OPERATOR\_TYPE Values

Value	Description
DELETE	Delete
FETCH	Fetch
FILTER	Filter rows
GENROW	Generate Row
GRPBY	Group By
HSJOIN	Hash Join
INSERT	Insert
IXAND	Dynamic Bitmap Index ANDing
IXSCAN	Index Scan
MSJOIN	Merge Scan Join
NLJOIN	Nested loop Join
RETURN	Result
RIDSCN	Row Identifier (RID) Scan

## Explain Tables

Table 102 (Page 2 of 2). OPERATOR\_TYPE Values

Value	Description
SORT	Sort
TBSCAN	Table Scan
TEMP	Temporary Table Construction
TQ	Table Queue
UNION	Union
UNIQUE	Duplicate Elimination
UPDATE	Update

### EXPLAIN\_PREDICATE Table

The EXPLAIN\_PREDICATE table identifies which predicates are applied by a specific operator.

For the definition of this table, see “EXPLAIN\_PREDICATE Table Definition” on page 954.

Table 103 (Page 1 of 2). EXPLAIN\_PREDICATE Table

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	CHAR(8)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	CHAR(8)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	CHAR(8)	No	FK	Schema, or qualifier, of source of Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	SMALLINT	No	FK	Statement number within package to which this explain information is related.
SECTNO	SMALLINT	No	FK	Section number within package to which this explain information is related.
OPERATOR_ID	SMALLINT	No	No	Unique ID for this operator within this query.
PREDICATE_ID	SMALLINT	No	No	Unique ID for this predicate for the specified operator.
HOW_APPLIED	CHAR(5)	No	No	How predicate is being used by the specified operator.

## Explain Tables

Table 103 (Page 2 of 2). EXPLAIN\_PREDICATE Table

Column Name	Data Type	Nullable?	Key?	Description
WHEN_EVALUATED	CHAR(3)	No	No	Indicates when the subquery used in this predicate is evaluated.  Possible values are:  <b>blank</b> This predicate does not contain a subquery. <b>EAA</b> The subquery used in this predicate is evaluated at application (EAA). That is, it is re-evaluated for every row processed by the specified operator, as the predicate is being applied. <b>EAO</b> The subquery used in this predicate is evaluated at open (EAO). That is, it is re-evaluated only once for the specified operator, and its results are re-used in the application of the predicate for each row. <b>MUL</b> There is more than one type of subquery in this predicate.
RELOP_TYPE	CHAR(2)	No	No	The type of relational operator used in this predicate.
SUBQUERY	CHAR(1)	No	No	Whether or not a data stream from a subquery is required for this predicate. There may be multiple subquery streams required.  Possible values are: <b>N</b> No subquery stream is required <b>Y</b> One or more subquery streams is required
FILTER_FACTOR	DOUBLE	No	No	The estimated fraction of rows that will be qualified by this predicate.
PREDICATE_TEXT	CLOB(64K)	Yes	No	The text of the predicate as recreated from the internal representation of the SQL statement.  Null if not available.

Table 104. Possible HOW\_APPLIED Values

Value	Description
JOIN	Used to join tables
RESID	Evaluated as a residual predicate
SARG	Evaluated as a sargable predicate for index or data page
START	Used as a start condition
STOP	Used as a stop condition

Table 105 (Page 1 of 2). Possible RELOP\_TYPE Values

Value	Description
blanks	Not Applicable
EQ	Equals
GE	Greater Than or Equal
GT	Greater Than

## Explain Tables

Table 105 (Page 2 of 2). Possible RELOP\_TYPE Values

Value	Description
IN	In list
LE	Less Than or Equal
LK	Like
LT	Less Than
NE	Not Equal
NL	Is Null
NN	Is Not Null

### EXPLAIN\_STATEMENT Table

The EXPLAIN\_STATEMENT table contains the text of the SQL statement as it exists for the different levels of Explain information. The original SQL statement as entered by the user is stored in this table along with the version used (by the optimizer) to choose an access plan to satisfy the SQL statement. The latter version may bear little resemblance to the original as it may have been rewritten and/or enhanced with additional predicates as determined by the SQL Compiler.

For the definition of this table, see “EXPLAIN\_STATEMENT Table Definition” on page 955.

Table 106 (Page 1 of 3). EXPLAIN\_STATEMENT Table

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	CHAR(8)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	CHAR(8)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	CHAR(8)	No	PK, FK	Schema, or qualifier, of source of Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK	Level of Explain information for which this row is relevant. Valid values are: <b>O</b> Original Text (as entered by user) <b>P</b> PLAN SELECTION
STMTNO	SMALLINT	No	PK	Statement number within package to which this explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
SECTNO	SMALLINT	No	PK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at runtime. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.



## Explain Tables

Table 106 (Page 2 of 3). EXPLAIN\_STATEMENT Table

Column Name	Data Type	Nullable?	Key?	Description
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.
STATEMENT_TYPE	CHAR(2)	No	No	Descriptive label for type of query being explained. Possible values are: <b>S</b> Select <b>D</b> Delete <b>DC</b> Delete where current of cursor <b>I</b> Insert <b>U</b> Update <b>UC</b> Update where current of cursor
UPDATABLE	CHAR(1)	No	No	Indicates if this statement is considered updatable. This is particularly relevant to SELECT statements which may be determined to be potentially updatable. Possible values are: <b>'</b> Not applicable (blank) <b>N</b> No <b>Y</b> Yes
DELETABLE	CHAR(1)	No	No	Indicates if this statement is considered deletable. This is particularly relevant to SELECT statements which may be determined to be potentially deletable. Possible values are: <b>'</b> Not applicable (blank) <b>N</b> No <b>Y</b> Yes
TOTAL_COST	DOUBLE	No	No	Estimated total cost (in timerons) of executing the chosen access plan for this statement; set to 0 (zero) if EXPLAIN_LEVEL is 0 (original text) since no access plan has been chosen at this time.
STATEMENT_TEXT	CLOB(64K)	No	No	Text or portion of the text of the SQL statement being explained. The text shown for the Plan Selection level of Explain has been reconstructed from the internal representation and is SQL-like in nature; that is, the reconstructed statement is not guaranteed to follow correct SQL syntax.
SNAPSHOT	BLOB(10M)	Yes	No	Snapshot of internal representation for this SQL statement at the Explain_Level shown.  This column is intended for use with DB2 Visual Explain. Column is set to null if EXPLAIN_LEVEL is 0 (original statement) since no access plan has been chosen at the time that this specific version of the statement is captured.

## Explain Tables

Table 106 (Page 3 of 3). EXPLAIN\_STATEMENT Table

Column Name	Data Type	Nullable?	Key?	Description
QUERY_DEGREE	INTEGER	No	No	Indicates the degree of intra-partition parallelism at the time of Explain invocation. For the original statement, this contains the directed degree of intra-partition parallelism. For the PLAN SELECTION, this contains the degree of intra-partition parallelism generated for the plan to use.

### EXPLAIN\_STREAM Table

The EXPLAIN\_STREAM table represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN\_OBJECT table. The operators involved in a data stream are to be found in the EXPLAIN\_OPERATOR table.

For the definition of this table, see “EXPLAIN\_STREAM Table Definition” on page 956.

Table 107 (Page 1 of 2). EXPLAIN\_STREAM Table

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	CHAR(8)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	CHAR(8)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	CHAR(8)	No	FK	Schema, or qualifier, of source of Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	SMALLINT	No	FK	Statement number within package to which this explain information is related.
SECTNO	SMALLINT	No	FK	Section number within package to which this explain information is related.
STREAM_ID	SMALLINT	No	No	Unique ID for this data stream within the specified operator.
SOURCE_TYPE	CHAR(1)	No	No	Indicates the source of this data stream: <b>O</b> Operator <b>D</b> Data Object
SOURCE_ID	SMALLINT	No	No	Unique ID for the operator within this query that is the source of this data stream. Set to -1 if SOURCE_TYPE is 'D'.
TARGET_TYPE	CHAR(1)	No	No	Indicates the target of this data stream: <b>O</b> Operator <b>D</b> Data Object
TARGET_ID	SMALLINT	No	No	Unique ID for the operator within this query that is the target of this data stream. Set to -1 if TARGET_TYPE is 'D'.
OBJECT_SCHEMA	CHAR(8)	Yes	No	Schema to which the affected data object belongs. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.

## Explain Tables

Table 107 (Page 2 of 2). EXPLAIN\_STREAM Table

Column Name	Data Type	Nullable?	Key?	Description
OBJECT_NAME	VARCHAR(18)	Yes	No	Name of the object that is the subject of data stream. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.
STREAM_COUNT	DOUBLE	No	No	Estimated cardinality of data stream.
COLUMN_COUNT	SMALLINT	No	No	Number of columns in data stream.
PREDICATE_ID	SMALLINT	No	No	If this stream is part of a subquery for a predicate, the predicate ID will be reflected here, otherwise the column is set to -1.
COLUMN_NAMES	CLOB(64K)	Yes	No	This column contains the names and ordering information of the columns involved in this stream.  These names will be in the format of:  NAME1 (A)+NAME2 (D)+NAME3+NAME4  Where (A) indicates a column in ascending order, (D) indicates a column in descending order, and no ordering information indicates that either the column is not ordered or ordering is not relevant.
PMID	SMALLINT	No	No	Partitioning map ID.
SINGLE_NODE	CHAR(5)	Yes	No	Indicates if this data stream is on a single or multiple partitions:  <b>MULT</b> On multiple partitions <b>COOR</b> On coordinator node <b>HASH</b> Directed using hashing <b>RID</b> Directed using the row ID <b>FUNC</b> Directed using a function (PARTITION() or NODENUMBER()) <b>CORR</b> Directed using a correlation value
PARTITION_COLUMNS	CLOB(64K)	Yes	No	List of columns this data stream is partitioned on.

### Table Definitions for Explain Tables

The Explain tables must be created before Explain can be invoked. The following definitions specify how to create the necessary Explain tables:

- “EXPLAIN\_ARGUMENT Table Definition” on page 950
- “EXPLAIN\_INSTANCE Table Definition” on page 951
- “EXPLAIN\_OBJECT Table Definition” on page 952
- “EXPLAIN\_OPERATOR Table Definition” on page 953
- “EXPLAIN\_PREDICATE Table Definition” on page 954
- “EXPLAIN\_STATEMENT Table Definition” on page 955
- “EXPLAIN\_STREAM Table Definition” on page 956

Alternately, create them by using the sample command line processor input script provided in the EXPLAIN.DDL file located in the 'misc' subdirectory of the 'sqllib' directory. Connect to the database where the Explain tables are required. Then issue the command: db2 -tf EXPLAIN.DDL and the tables will be created.

## Explain Tables

### EXPLAIN\_ARGUMENT Table Definition

```
CREATE TABLE EXPLAIN_ARGUMENT ( EXPLAIN_REQUESTER CHAR(8) NOT NULL,  
EXPLAIN_TIME TIMESTAMP NOT NULL,  
SOURCE_NAME CHAR(8) NOT NULL,  
SOURCE_SCHEMA CHAR(8) NOT NULL,  
EXPLAIN_LEVEL CHAR(1) NOT NULL,  
STMTNO SMALLINT NOT NULL,  
SECTNO SMALLINT NOT NULL,  
OPERATOR_ID SMALLINT NOT NULL,  
ARGUMENT_TYPE CHAR(8) NOT NULL,  
ARGUMENT_VALUE VARCHAR(30) NOT NULL,  
FOREIGN KEY (EXPLAIN_REQUESTER,  
EXPLAIN_TIME,  
SOURCE_NAME,  
SOURCE_SCHEMA,  
EXPLAIN_LEVEL,  
STMTNO,  
SECTNO)  
REFERENCES EXPLAIN_STATEMENT  
ON DELETE CASCADE )
```

### EXPLAIN\_INSTANCE Table Definition

```

CREATE TABLE EXPLAIN_INSTANCE (
    EXPLAIN_REQUESTER CHAR(8) NOT NULL,
    EXPLAIN_TIME       TIMESTAMP NOT NULL,
    SOURCE_NAME        CHAR(8) NOT NULL,
    SOURCE_SCHEMA      CHAR(8) NOT NULL,
    EXPLAIN_OPTION     CHAR(1) NOT NULL,
    SNAPSHOT_TAKEN     CHAR(1) NOT NULL,
    DB2_VERSION        CHAR(7) NOT NULL,
    SQL_TYPE           CHAR(1) NOT NULL,
    QUERYOPT           INTEGER NOT NULL,
    BLOCK              CHAR(1) NOT NULL,
    ISOLATION          CHAR(2) NOT NULL,
    BUFPAGE            INTEGER NOT NULL,
    AVG_APPLS          INTEGER NOT NULL,
    SORTHEAP           INTEGER NOT NULL,
    LOCKLIST           INTEGER NOT NULL,
    MAXLOCKS           SMALLINT NOT NULL,
    LOCKS_AVAIL        INTEGER NOT NULL,
    CPU_SPEED          DOUBLE NOT NULL,
    REMARKS            VARCHAR(254),
    DBHEAP             INTEGER NOT NULL,
    COMM_SPEED         DOUBLE NOT NULL,
    PARALLELISM        CHAR(2) NOT NULL,
    PRIMARY KEY (EXPLAIN_REQUESTER,
                 EXPLAIN_TIME,
                 SOURCE_NAME,
                 SOURCE_SCHEMA))

```

## Explain Tables

### EXPLAIN\_OBJECT Table Definition

```
CREATE TABLE EXPLAIN_OBJECT ( EXPLAIN_REQUESTER CHAR(8) NOT NULL,
                               EXPLAIN_TIME        TIMESTAMP NOT NULL,
                               SOURCE_NAME         CHAR(8) NOT NULL,
                               SOURCE_SCHEMA       CHAR(8) NOT NULL,
                               EXPLAIN_LEVEL       CHAR(1) NOT NULL,
                               STMTNO             SMALLINT NOT NULL,
                               SECTNO             SMALLINT NOT NULL,
                               OBJECT_SCHEMA       CHAR(8) NOT NULL,
                               OBJECT_NAME        VARCHAR(18) NOT NULL,
                               OBJECT_TYPE        CHAR(2) NOT NULL,
                               CREATE_TIME        TIMESTAMP,
                               STATISTICS_TIME    TIMESTAMP,
                               COLUMN_COUNT       SMALLINT NOT NULL,
                               ROW_COUNT          INTEGER NOT NULL,
                               WIDTH              INTEGER NOT NULL,
                               PAGES              INTEGER NOT NULL,
                               DISTINCT           CHAR(1) NOT NULL,
                               TABLESPACE_NAME   VARCHAR(18),
                               OVERHEAD           DOUBLE NOT NULL,
                               TRANSFER_RATE     DOUBLE NOT NULL,
                               PREFETCHSIZE      INTEGER NOT NULL,
                               EXTENTSIZE        INTEGER NOT NULL,
                               CLUSTER           DOUBLE NOT NULL,
                               NLEAF             INTEGER NOT NULL,
                               NLEVELS          INTEGER NOT NULL,
                               FULLKEYCARD       INTEGER NOT NULL,
                               OVERFLOW          INTEGER NOT NULL,
                               FOREIGN KEY (EXPLAIN_REQUESTER,
                                             EXPLAIN_TIME,
                                             SOURCE_NAME,
                                             SOURCE_SCHEMA,
                                             EXPLAIN_LEVEL,
                                             STMTNO,
                                             SECTNO)
                               REFERENCES EXPLAIN_STATEMENT
                               ON DELETE CASCADE )
```

### EXPLAIN\_OPERATOR Table Definition

```

CREATE TABLE EXPLAIN_OPERATOR ( EXPLAIN_REQUESTER CHAR(8) NOT NULL,
                                EXPLAIN_TIME TIMESTAMP NOT NULL,
                                SOURCE_NAME CHAR(8) NOT NULL,
                                SOURCE_SCHEMA CHAR(8) NOT NULL,
                                EXPLAIN_LEVEL CHAR(1) NOT NULL,
                                STMTNO SMALLINT NOT NULL,
                                SECTNO SMALLINT NOT NULL,
                                OPERATOR_ID SMALLINT NOT NULL,
                                OPERATOR_TYPE CHAR(6) NOT NULL,
                                TOTAL_COST DOUBLE NOT NULL,
                                IO_COST DOUBLE NOT NULL,
                                CPU_COST DOUBLE NOT NULL,
                                FIRST_ROW_COST DOUBLE NOT NULL,
                                RE_TOTAL_COST DOUBLE NOT NULL,
                                RE_IO_COST DOUBLE NOT NULL,
                                RE_CPU_COST DOUBLE NOT NULL,
                                COMM_COST DOUBLE NOT NULL,
                                FIRST_COMM_COST DOUBLE NOT NULL,
                                NODES_USED CLOB(64K) NOT LOGGED,
                                FOREIGN KEY (EXPLAIN_REQUESTER,
                                             EXPLAIN_TIME,
                                             SOURCE_NAME,
                                             SOURCE_SCHEMA,
                                             EXPLAIN_LEVEL,
                                             STMTNO,
                                             SECTNO)
                                REFERENCES EXPLAIN_STATEMENT
                                ON DELETE CASCADE )

```

## Explain Tables

### EXPLAIN\_PREDICATE Table Definition

```
CREATE TABLE EXPLAIN_PREDICATE ( EXPLAIN_REQUESTER CHAR(8)      NOT NULL,
EXPLAIN_TIME      TIMESTAMP      NOT NULL,
SOURCE_NAME       CHAR(8)        NOT NULL,
SOURCE_SCHEMA     CHAR(8)        NOT NULL,
EXPLAIN_LEVEL     CHAR(1)        NOT NULL,
STMTNO           SMALLINT       NOT NULL,
SECTNO           SMALLINT       NOT NULL,
OPERATOR_ID       SMALLINT       NOT NULL,
PREDICATE_ID      SMALLINT       NOT NULL,
HOW_APPLIED       CHAR(5)        NOT NULL,
WHEN_EVALUATED    CHAR(3)        NOT NULL,
RELOP_TYPE        CHAR(2)        NOT NULL,
SUBQUERY          CHAR(1)        NOT NULL,
FILTER_FACTOR     DOUBLE         NOT NULL,
PREDICATE_TEXT    CLOB(64K)     NOT LOGGED,
FOREIGN KEY (EXPLAIN_REQUESTER,
EXPLAIN_TIME,
SOURCE_NAME,
SOURCE_SCHEMA,
EXPLAIN_LEVEL,
STMTNO,
SECTNO)
REFERENCES EXPLAIN_STATEMENT
ON DELETE CASCADE )
```



## Explain Tables

### EXPLAIN\_STATEMENT Table Definition

```
CREATE TABLE EXPLAIN_STATEMENT ( EXPLAIN_REQUESTER CHAR(8) NOT NULL,
EXPLAIN_TIME TIMESTAMP NOT NULL,
SOURCE_NAME CHAR(8) NOT NULL,
SOURCE_SCHEMA CHAR(8) NOT NULL,
EXPLAIN_LEVEL CHAR(1) NOT NULL,
STMTNO SMALLINT NOT NULL,
SECTNO SMALLINT NOT NULL,
QUERYNO INTEGER NOT NULL,
QUERYTAG CHAR(20) NOT NULL,
STATEMENT_TYPE CHAR(2) NOT NULL,
UPDATABLE CHAR(1) NOT NULL,
DELETABLE CHAR(1) NOT NULL,
TOTAL_COST DOUBLE NOT NULL,
STATEMENT_TEXT CLOB(64K) NOT NULL NOT LOGGED,
SNAPSHOT BLOB(10M) NOT LOGGED,
QUERY_DEGREE INTEGER NOT NULL,
PRIMARY KEY (EXPLAIN_REQUESTER,
EXPLAIN_TIME,
SOURCE_NAME,
SOURCE_SCHEMA,
EXPLAIN_LEVEL,
STMTNO,
SECTNO),
FOREIGN KEY (EXPLAIN_REQUESTER,
EXPLAIN_TIME,
SOURCE_NAME,
SOURCE_SCHEMA)
REFERENCES EXPLAIN_INSTANCE
ON DELETE CASCADE )
```

## Explain Tables

### EXPLAIN\_STREAM Table Definition

```
CREATE TABLE EXPLAIN_STREAM ( EXPLAIN_REQUESTER CHAR(8) NOT NULL,  
EXPLAIN_TIME TIMESTAMP NOT NULL,  
SOURCE_NAME CHAR(8) NOT NULL,  
SOURCE_SCHEMA CHAR(8) NOT NULL,  
EXPLAIN_LEVEL CHAR(1) NOT NULL,  
STMTNO SMALLINT NOT NULL,  
SECTNO SMALLINT NOT NULL,  
STREAM_ID SMALLINT NOT NULL,  
SOURCE_TYPE CHAR(1) NOT NULL,  
SOURCE_ID SMALLINT NOT NULL,  
TARGET_TYPE CHAR(1) NOT NULL,  
TARGET_ID SMALLINT NOT NULL,  
OBJECT_SCHEMA CHAR(8),  
OBJECT_NAME VARCHAR(18),  
STREAM_COUNT DOUBLE NOT NULL,  
COLUMN_COUNT SMALLINT NOT NULL,  
PREDICATE_ID SMALLINT NOT NULL,  
COLUMN_NAMES CLOB(64K) NOT LOGGED,  
PMID SMALLINT NOT NULL,  
SINGLE_NODE CHAR(5),  
PARTITION_COLUMNS CLOB(64K) NOT LOGGED,  
FOREIGN KEY (EXPLAIN_REQUESTER,  
EXPLAIN_TIME,  
SOURCE_NAME,  
SOURCE_SCHEMA,  
EXPLAIN_LEVEL,  
STMTNO,  
SECTNO)  
REFERENCES EXPLAIN_STATEMENT  
ON DELETE CASCADE )
```

## Appendix L. Explain Register Values

This appendix describes the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values with each other and with the PREP and BIND commands.

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact in the following way for dynamic SQL.

Table 108. Interaction of Explain Special Register Values for Dynamic SQL

EXPLAIN MODE values	EXPLAIN SNAPSHOT values		
	NO	YES	EXPLAIN
NO	<ul style="list-style-type: none"> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>
YES	<ul style="list-style-type: none"> <li>Explain Snapshot taken.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated</li> <li>Explain Snapshot taken</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated</li> <li>Explain Snapshot taken</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>
EXPLAIN	<ul style="list-style-type: none"> <li>Explain Snapshot taken</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated</li> <li>Explain Snapshot taken</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated</li> <li>Explain Snapshot taken</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>

The CURRENT EXPLAIN MODE special register interacts with the EXPLAIN bind option in the following way for dynamic SQL.

Table 109. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE

EXPLAIN Bind option values	EXPLAIN MODE values		
	NO	YES	ALL
NO	<ul style="list-style-type: none"> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL</li> <li>Explain tables populated for dynamic SQL</li> <li>Results of query returned.</li> </ul>
YES	<ul style="list-style-type: none"> <li>Explain tables populated for dynamic SQL</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL</li> <li>Explain tables populated for dynamic SQL</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL</li> <li>Explain tables populated for dynamic SQL</li> <li>Results of query returned.</li> </ul>
EXPLAIN	<ul style="list-style-type: none"> <li>Explain tables populated for dynamic SQL</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL</li> <li>Explain tables populated for dynamic SQL</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL</li> <li>Explain tables populated for dynamic SQL</li> <li>Results of query not returned (Dynamic statements not executed).</li> </ul>

The CURRENT EXPLAIN SNAPSHOT special register interacts with the EXPLSNAP bind option in the following way for dynamic SQL.

Table 110. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT

EXPLSNAP Bind option values	EXPLAIN SNAPSHOT values		
	NO	YES	ALL
NO	<ul style="list-style-type: none"> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL</li> <li>Explain Snapshot taken for dynamic SQL</li> <li>Results of query returned.</li> </ul>

Table 110. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT

EXPLSNAP Bind option values	EXPLAIN SNAPSHOT values		
	NO	YES	ALL
YES	<ul style="list-style-type: none"> <li>• Explain Snapshot taken for dynamic SQL</li> <li>• Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>• Explain Snapshot taken for static SQL</li> <li>• Explain Snapshot taken for dynamic SQL</li> <li>• Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>• Explain Snapshot taken for static SQL</li> <li>• Explain Snapshot taken for dynamic SQL</li> <li>• Results of query returned.</li> </ul>
EXPLAIN	<ul style="list-style-type: none"> <li>• Explain Snapshot taken for dynamic SQL</li> <li>• Results of query not returned (Dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>• Explain Snapshot taken for static SQL</li> <li>• Explain Snapshot taken for dynamic SQL</li> <li>• Results of query not returned (Dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>• Explain Snapshot taken for static SQL</li> <li>• Explain Snapshot taken for dynamic SQL</li> <li>• Results of query not returned (Dynamic statements not executed).</li> </ul>



---

## Appendix M. Recursion Example: Bill of Materials

Bill of materials (BOM) applications are a common requirement in many business environments. To illustrate the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by the part. For this example, create the table as follows.

```
CREATE TABLE PARTLIST
(PART VARCHAR(8),
 SUBPART VARCHAR(8),
 QUANTITY INTEGER);
```

In order to give query results for this example, assume the PARTLIST table is populated with the following values.

PART	SUBPART	QUANTITY
-----	-----	-----
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	14	8
07	12	8

---

### Example 1: Single Level Explosion

The first example is called single level explosion. It answers the question, "What parts are needed to build the part identified by '01'?" The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

## Recursion Example: Bill of Materials

```
WITH RPL (PART, SUBPART, QUANTITY) AS
  ( SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
    FROM PARTLIST ROOT
    WHERE ROOT.PART = '01'
  UNION ALL
    SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
    FROM RPL PARENT, PARTLIST CHILD
    WHERE PARENT.SUBPART = CHILD.PART
  )
SELECT DISTINCT PART, SUBPART, QUANTITY
FROM RPL
ORDER BY PART, SUBPART, QUANTITY;
```

The above query includes a common table expression, identified by the name *RPL*, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (fullselect) of the UNION, referred to as the *initialization fullselect*, gets the direct children of part '01'. The FROM clause of this fullselect refers to the source table and will never refer to itself (*RPL* in this case). The result of this first fullselect goes into the common table expression *RPL* (Recursive PARTLIST). As in this example, the UNION must always be a UNION ALL.

The second operand (fullselect) of the UNION uses *RPL* to compute subparts of subparts by having the FROM clause refer to the common table expression *RPL* and the source table with a join of a part from the source table (child) to a subpart of the current result contained in *RPL* (parent). The result goes back to *RPL* again. The second operand of UNION is then used repeatedly until no more children exist.

The SELECT DISTINCT in the main fullselect of this query ensures the same part/subpart is not listed more than once.

The result of the query is as follows:

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	12	8
07	14	8



## Recursion Example: Bill of Materials

Observe in the result that from part '01' we go to '02' which goes to '06' and so on. Further, notice that part '06' is reached twice, once through '01' directly and another time through '02'. In the output, however, its subcomponents are listed only once (this is the result of using a SELECT DISTINCT) as required.

It is important to remember that with recursive common table expressions it is possible to introduce an *infinite loop*. In this example, an infinite loop would be created if the search condition of the second operand that joins the parent and child tables was coded as:

```
PARENT.SUBPART = CHILD.SUBPART
```

This example of causing an infinite loop is obviously a case of not coding what is intended. However, care should also be exercised in determining what to code so that there is a definite end of the recursion cycle.

The result produced by this example query could be produced in an application program without using a recursive common table expression. However, this approach would require starting of a new query for every level of recursion. Furthermore, the application needs to put all the results back in the database to order the result. This approach complicates the application logic and does not perform well. The application logic becomes even harder and more inefficient for other bill of material queries, such as summarized and indented explosion queries.

---

### Example 2: Summarized Explosion

The second example is a summarized explosion. The question posed here is, what is the total quantity of each part required to build part '01'. The main difference from the single level explosion is the need to aggregate the quantities. The first example indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of the subparts are needed to build part '01'.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
(
  SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
  FROM PARTLIST ROOT
  WHERE ROOT.PART = '01'
  UNION ALL
  SELECT PARENT.PART, CHILD.SUBPART, PARENT.QUANTITY*CHILD.QUANTITY
  FROM RPL PARENT, PARTLIST CHILD
  WHERE PARENT.SUBPART = CHILD.PART
)
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total QTY Used"
FROM RPL
GROUP BY PART, SUBPART
ORDER BY PART, SUBPART;
```

In the above query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name *RPL*, shows the aggregation of the quantity. To find out how much of a subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different

## Recursion Example: Bill of Materials

places, it requires another final aggregation. This is done by the grouping over the common table expression *RPL* and using the SUM column function in the select list of the main fullselect.

The result of the query is as follows:

PART	SUBPART	Total Qty Used
01	02	2
01	03	3
01	04	4
01	05	14
01	06	15
01	07	18
01	08	40
01	09	44
01	10	140
01	11	140
01	12	294
01	13	150
01	14	144

Looking at the output, consider the line for subpart '06'. The total quantity used value of 15 is derived from a quantity of 3 directly for part '01' and a quantity of 6 for part '02' which is needed 2 times by part '01'.

---

### Example 3: Controlling Depth

The question may come to mind, what happens when there are more levels of parts in the table than you are interested in for your query? That is, how is a query written to answer the question, "What are the first two levels of parts needed to build the part identified by '01'?" For the sake of clarity in the example, the level is included in the result.

```
WITH RPL (LEVEL, PART, SUBPART, QUANTITY) AS
(
  SELECT 1,          ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
  FROM PARTLIST ROOT
  WHERE ROOT.PART = '01'
  UNION ALL
  SELECT PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
  FROM RPL PARENT, PARTLIST CHILD
  WHERE PARENT.SUBPART = CHILD.PART
  AND PARENT.LEVEL < 2
)
SELECT PART, LEVEL, SUBPART, QUANTITY
FROM RPL;
```

This query is similar to example 1. The column *LEVEL* was introduced to count the levels from the original part. In the initialization fullselect, the value for the *LEVEL* column is initialized to 1. In the subsequent fullselect, the level from the parent is incremented by 1. Then to control the number of levels in the result, the second fullselect

## Recursion Example: Bill of Materials

includes the condition that the parent level must be less than 2. This ensures that the second fullselect only processes children to the second level.

The result of the query is:

PART	LEVEL	SUBPART	QUANTITY
01		1 02	2
01		1 03	3
01		1 04	4
01		1 06	3
02		2 05	7
02		2 06	6
03		2 07	6
04		2 08	10
04		2 09	11
06		2 12	10
06		2 13	10

## Recursion Example: Bill of Materials

---

## Appendix N. Exception Tables

Exception tables are user-created tables that mimic the definition of the tables that are specified to be checked using SET CONSTRAINTS with the IMMEDIATE CHECKED option. They are used to store copies of the rows that violate constraints in the tables being checked.

The exception tables used with LOAD are identical to the ones used here. They can therefore be reused during checking with the SET CONSTRAINTS statement.

---

### Rules for Creating an Exception Table

The rules for creating an exception table are as follows:

1. The first “n” columns of the exception table are the same as the columns of the table being checked. All column attributes including name, type and length should be identical.
2. All the columns of the exception table must be free of any constraints and triggers. Constraints include referential integrity, check constraints as well as unique index constraints that could cause errors on insert.
3. The “(n+1)” column of the exception table is an optional TIMESTAMP column. This serves to identify successive invocations of checking by the SET CONSTRAINTS statement on the same table, if the rows within the exception table have not been deleted before issuing the SET CONSTRAINTS statement to check the data.
4. The “(n+2)” column should be of type CLOB(32K) or larger. This column is optional but recommended, and will be used to give the names of the constraints that the data within the row violates. If this column is not provided (as could be warranted if, for example, the original table had the maximum number of columns allowed), then only the row where the constraint violation was detected is copied.
5. The exception table should be created with both “(n+1)” and the “(n+2)” columns.
6. There is no enforcement of any particular name for the above additional columns. However, the type specification must be exactly followed.
7. No additional columns are allowed.
8. If the original table has DATALINK columns, the corresponding columns in the exception table should specify NO LINK CONTROL. This ensures that a file is not linked when a row (with DATALINK column) is inserted and an access token is not generated when rows are selected from the exception table.
9. It should also be noted that users invoking SET CONSTRAINTS to check the data must have INSERT privilege on the exception tables.

The information in the “message” column will be according to the following structure :

Table 111. Exception Table Message Column Structure

Field number	Contents	Size	Comments
1	Number of constraint violations	5 characters	Right justified padded with '0'
2	Type of first constraint violation	1 character	'K' - Check Constraint violation 'F' - Foreign Key violation 'I' - Unique Index violation (See Note) 'L' - DATALINK load violation
3	Length of constraint/index ID <sup>a</sup> / DLVDESC <sup>b</sup>	5 characters	Right justified padded with '0'
4	Constraint name/index ID <sup>a</sup> / DLVDESC <sup>b</sup>	length from the previous field	
5	Separator	3 characters	<space><colon><space>
6	Type of next constraint violation	1 character	'K' - Check Constraint violation 'F' - Foreign Key violation 'I' - Unique Index violation 'L' - DATALINK load violation
7	Length of constraint/index name/ DLVDESC	5 characters	Right justified padded with '0'
8	Constraint name/ Index name/ DLVDESC	length from the previous field	
.....	.....	.....	Repeat Field 5 through 8 for each violation

**Note:**

- In DB2 Version 5.2, unique index violations will not occur with checking using SET CONSTRAINTS. This will be reported, however, when running LOAD if the FOR EXCEPTION option is chosen. LOAD on the other hand will not report check constraint and foreign key violations in the exception tables in DB2 Version 5.2.
- <sup>a</sup> To retrieve an index ID from the catalog views, use a select statement. For example, if field 4 is 1234, then  
SELECT INDSHEMA, INDNAME FROM SYSCAT.INDEXES WHERE IID=1234.
- <sup>b</sup>DLVDESC is a DATALINK Load Violation DESCriptor described below.

Table 112 (Page 1 of 2). DATALINK Load Violation DESCriptor (DLVDESC)

Field number	Contents	Size	Comments
1	Number of violating DATALINK columns	4 characters	Right justified padded with '0'
2	DATALINK column number of the first violating column	4 characters	Right justified padded with '0'
2	DATALINK column number of the second violating column	4 characters	Right justified padded with '0'
.....	.....	.....	Repeat for each violating column number

---

Table 112 (Page 2 of 2). DATALINK Load Violation DESCriptor (DLVDESC)

---

Field number	Contents	Size	Comments
--------------	----------	------	----------

---

**Note:**

- DATALINK column number is COLNO in SYSCAT.COLUMNS for the appropriate table.
- 

---

## Handling Rows in the Exception Tables

The information in the exception tables can be processed in any manner desired. The rows could be used to correct the data and re-insert the rows into the original tables.

If there are no INSERT triggers on the original table, transfer the corrected rows by issuing an INSERT statement with a subquery on the exception table.

If there are INSERT triggers and you wish to complete the load with the corrected rows from exception tables without firing the triggers, the following ways are suggested:

- Design the INSERT triggers to be fired depending on the value in a column defined explicitly for the purpose.
- Unload the data from the exception tables and append them using LOAD. In this case if we re-check the data, it should be noted that in DB2 Version 5.2 the constraint violation checking is not confined to the appended rows only.
- Save the trigger text from the relevant catalog table. Then drop the INSERT trigger and use INSERT to transfer the corrected rows from the exception tables. Finally recreate the trigger using the saved information.

In DB2 Version 5.2, no explicit provision is made to prevent the firing of triggers when inserting rows from the exception tables.

Only one violation per row will be reported for unique index violations.

If values with long string or LOB data types are in the table, the values will not be inserted into the exception table in case of unique index violation.

---

## Querying the Exception Tables

The message column structure in an exception table is a concatenated list of constraint names, lengths and delimiters as described earlier. You may wish to write a query on this information.

For example, let's write a query to obtain a list of all the violations, repeating each row with only the constraint name along with it. Let us assume that our original table T1 had two columns C1 and C2. Assume also, that the corresponding exception table E1 has columns C1, C2 pertaining to those in T1 and MSGCOL as the message column. The following query (using recursion) will list one constraint name per row (repeating the row for more than one violation):

```

WITH IV (C1, C2, MSGCOL, CONSTNAME, I, J) AS
  (SELECT C1, C2, MSGCOL,
    CHAR(SUBSTR(MSGCOL, 12,
      INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
    1,
    15+INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
  FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
    CHAR(SUBSTR(MSGCOL, J+6,
      INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
    I+1,
    J+9+INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
  FROM IV
  WHERE I < INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
  ) SELECT C1, C2, CONSTNAME FROM IV;

```

If we want all the rows that violated a particular constraint, we could extend this query as follows:

```

WITH IV (C1, C2, MSGCOL, CONSTNAME, I, J) AS
  (SELECT C1, C2, MSGCOL,
    CHAR(SUBSTR(MSGCOL, 12,
      INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
    1,
    15+INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
  FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
    CHAR(SUBSTR(MSGCOL, J+6,
      INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
    I+1,
    J+9+INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
  FROM IV
  WHERE I < INTEGER(DECIMAL (VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
  ) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTNAME = 'constraintname';

```

To obtain all the Check Constraint violations, one could execute the following:



```

WITH IV (C1, C2, MSGCOL, CONSTNAME, CONSTTYPE, I, J) AS
  (SELECT C1, C2, MSGCOL,
    CHAR(SUBSTR(MSGCOL, 12,
      INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))))),
    CHAR(SUBSTR(MSGCOL, 6, 1)),
    1,
    15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
  FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
    CHAR(SUBSTR(MSGCOL, J+6,
      INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))))),
    CHAR(SUBSTR(MSGCOL, J, 1)),
    I+1,
    J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
  FROM IV
  WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTTYPE = 'K';

```



---

### Appendix O. Japanese and Traditional-Chinese EUC Considerations

Extended Unix Code (EUC) for Japanese and Traditional-Chinese defines a set of encoding rules that can support from 1 to 4 character sets. In some cases, such as Japanese EUC (eucJP) and Traditional-Chinese EUC (eucTW), a character may be encoded using more than two bytes. Use of such an encoding scheme has implications when used as the code page of the database server or the database client. The key considerations involve the following:

- expansion or contraction of strings when converting between EUC code pages and double-byte code pages
- use of Universal Character Set-2 (UCS-2) as the code page for graphic data stored in a database server defined with the eucJP (Japanese) or eucTW (Traditional-Chinese) code pages.

With the exception of these considerations, the use of EUC is consistent with the double-byte character set (DBCS) support. Throughout this book (and others), references to *double-byte* have been changed to *multi-byte* to reflect support for encoding rules that allow for character representations that require more than 2 bytes. Detailed considerations for support of Japanese and Traditional-Chinese EUC are included here, organized in the same way as the contents of the chapters of this book. This information should be considered by anyone using SQL with an EUC database server or an EUC database client and used in conjunction with application development information in the *Application Programming Guide*

---

### Language Elements

#### Characters

Each multi-byte character is considered a *letter* with the exception of the double-byte blank character which is considered a *special character*.

#### Tokens

Multi-byte lowercase alphabetic letters are not folded to uppercase. This differs from the single byte lowercase alphabetic letters in tokens which are generally folded to uppercase.

#### Identifiers

##### SQL Identifiers

Conversion between a double-byte code page and an EUC code page may result in the conversion of double-byte characters to multi-byte characters encoded with more than 2 bytes. As a result, an identifier that fits the length maximum in the double-byte code page may exceed the length in the EUC code page. Selecting identifiers for this type of

## Japanese and Traditional-Chinese EUC Considerations

environment must be done carefully to avoid expansion beyond the maximum identifier length.

### Data Types

#### Character Strings

In an MBCS database, character strings may contain a mixture of characters from a single-byte character set (SBCS) and from multi-byte character sets (MBCS). When using such strings, operations may provide different results if they are character based (treat the data as characters) or byte based (treat the data as bytes). Check the function or operation description to determine how mixed strings are processed.

#### Graphic Strings

A graphic string is defined as a sequence of double-byte character data. In order to allow Japanese or Traditional-Chinese EUC data to be stored in graphic columns, EUC characters are encoded in UCS-2. Characters that are not double-byte characters under all supported encoding schemes (for example, PC or EBCDIC DBCS) should not be used with graphic columns. The results of using other than double-byte characters may result in replacement by substitution characters during conversion. Retrieval of such data will not return the same value as was entered. Refer to the *Application Programming Guide* programming language sections for details on handling graphic data in host variables.

### Assignments and Comparisons

#### String Assignments

Conversion of a string is performed prior to the assignment. In cases involving an eucJP/eucTW code page and a DBCS code page, a character string may become longer (DBCS to eucJP/eucTW) or shorter (eucJP/eucTW to DBCS). This may result in errors on storage assignment and truncation on retrieval assignment. When the error on storage assignment is due to expansion during conversion, SQLSTATE 22524 is returned instead of SQLSTATE 22001.

Similarly, assignments involving graphic strings may result in the conversion of a UCS-2 encoded double-byte character to a substitution character in a PC or EBCDIC DBCS code page for characters that do not have a corresponding double-byte character. Assignments that replace characters with substitution characters will indicate this by setting the SQLWARN10 field of the SQLCA to 'W'.

In cases of truncation during retrieval assignment involving multi-byte character strings, the point of truncation may be part of a multi-byte character. In this case, each byte of the character fragment is replaced with a single-byte blank. This means that more than one single-byte blank may appear at the end of a truncated character string.

#### String Comparisons

String comparisons are performed on a byte basis. Character strings also use the collating sequence defined for the database. Graphic strings do not use the collating

## Japanese and Traditional-Chinese EUC Considerations

sequence and, in an eucJP or eucTW database, are encoded using UCS-2. Thus, the comparison of two mixed character strings may have a different result from the comparison of two graphic strings even though they contain the same characters. Similarly, the resulting sort order of a mixed character column and a graphic column may be different.

### Rules for Result Data Types

The resulting data type for character strings is not affected by the possible expansion of the string. For example, a union of two CHAR operands will still be a CHAR. However, if one of the character string operands will be converted such that the maximum expansion makes the length attribute the largest of the two operands, then the resulting character string length attribute is affected. For example, consider the result expressions of a CASE expression that have data types of VARCHAR(100) and VARCHAR(120). Assume the VARCHAR(100) expression is a mixed string host variable (that may require conversion) and the VARCHAR(120) expression is a column in the eucJP database. The resulting data type is VARCHAR(200) since the VARCHAR(100) is doubled to allow for possible conversion. The same scenario without the involvement of an eucJP or eucTW database would have a result type of VARCHAR(120).

Notice that the doubling of the host variable length is based on the fact that the database server is Japanese EUC or Traditional-Chinese EUC. Even if the client is also eucJP or eucTW, the doubling is still applied. This allows the same application package to be used by double-byte or multi-byte clients.

### Rules for String Conversions

The types of operations listed in the corresponding section of the SQL Reference may convert operands to either the application or the database code page.

If such operations are done in a mixed code page environment that includes Japanese or Traditional-Chinese EUC, expansion or contraction of mixed character string operands may occur. Therefore the resulting data type has a length attribute that accommodates the maximum expansion, if possible. In the cases where there are restrictions on the length attribute of the data type, the maximum allowed length for the data type is used. For example in an environment where maximum growth is double, a VARCHAR(200) host variable is treated as if it is a VARCHAR(400), but CHAR(200) host variable is treated as if it is a CHAR(254). A run-time error may occur when conversion is performed if the converted string would exceed the maximum length for the data type. For example, the union of CHAR(200) and CHAR(10) would have a result type of CHAR(254). When the value from the left side of the UNION is converted, if more than 254 characters are required, an error occurs.

In some cases, allowing for the maximum growth for conversion will cause the length attribute to exceed a limit. For example, UNION only allows columns up to 254 bytes. Thus, a query with a union that included a host variable in the column list (call it :hv1) that was a DBCS mixed character string defined as a varying length character string 128 bytes long, would set the data type to VARCHAR(256) resulting in an error preparing the query, even though the query in the application does not appear to have any columns greater than 254. In a situation where the actual string is not likely to cause expansion beyond 254 bytes the following can be used to prepare the statement.

## Japanese and Traditional-Chinese EUC Considerations

```
SELECT CAST(:hv1 CONCAT ' ' AS VARCHAR(254)), C2 FROM T1
UNION
SELECT C1, C2 FROM T2
```

The concatenation of the null string with the host variable will force the conversion to occur before the cast is done. This query can be prepared in the DBCS to eucJP/eucTW environment although a truncation error may occur at run-time.

This technique (null string concat with cast) can be used to handle the similar 254 byte limit for SELECT DISTINCT or use of the column in ORDER BY or GROUP BY clauses.

## Constants

### Graphic String Constants

A graphic string constant, in the case of a Japanese or Traditional-Chinese EUC client, may contain single or multi-byte characters (like a mixed character string). The string should not contain more than 2000 characters. It is recommended that only characters that convert to double-byte characters in all related PC and EBCDIC double-byte code pages be used in graphic constants. A graphic string constant in an SQL statement is converted from the client code page to the double-byte encoding at the database server. For a Japanese or Traditional-Chinese EUC server, the constant is converted to UCS-2, the double-byte encoding used for graphic strings. For a double-byte server, the constant is converted from the client code page to the DBCS code page of the server.

## Functions

The design of user-defined functions should consider the impact of supporting Japanese or Tradition-Chinese EUC on the parameter data types. One part of function resolution considers the data types of the arguments to a function call. Mixed character string arguments involving a Japanese or Traditional-Chinese EUC client may require additional bytes to specify the argument. This may require that the data type change to allow the increased length. For example, it may take 4001 bytes to represent a character string in the application (a LONG VARCHAR) that fits into a VARCHAR(4000) string at the server. If a function signature is not included that allows the argument to be a LONG VARCHAR, function resolution will fail to find a function.

Some functions exist that do not allow long strings for various reasons. Use of LONG VARCHAR or CLOB arguments with such functions will not succeed. For example, LONG VARCHAR as the second argument of the built-in POSSTR function, will fail function resolution (SQLSTATE 42884).

## Expressions

### With the Concatenation Operator

The potential expansion of one of the operands of concatenation may cause the data type and length of concatenated operands to change when in an environment that includes a Japanese or Traditional-Chinese EUC database server. For example, with an

## Japanese and Traditional-Chinese EUC Considerations

EUC server where the value from a host variable may double in length, consider the following example.

```
CHAR200 CONCAT :char50
```

The column *CHAR200* is of type CHAR(200). The host variable *char50* is defined as CHAR(50). The result type for this concatenation operation would normally be CHAR(250). However, given an eucJP or eucTW database server, the assumption is that the string may expand to double the length. Hence *char50* is treated as a CHAR(100) and the resulting data type is VARCHAR(300). Note that even though the result is a VARCHAR, it will always have 300 bytes of data including trailing blanks. If the extra trailing blanks are not desired, define the host variable as VARCHAR(50) instead of CHAR(50).

### Predicates

#### LIKE Predicate

For a LIKE predicate involving mixed character strings in an EUC database:

- single-byte underscore represents any one single-byte character
- single-byte percent represents a string of zero or more characters (single-byte or multi-byte characters).
- double-byte underscore represents any one multi-byte character
- double-byte percent represents a string of zero or more characters (single-byte or multi-byte characters).

The escape character must be one single-byte character or one double-byte character.

Note that use of the underscore character may produce different results depending on the code page of the LIKE operation. For example, Katakana characters in Japanese EUC are multi-byte characters (CS2) but in the Japanese DBCS code page they are single-byte characters. A query with the single-byte underscore in the *pattern-expression* would return occurrences of Katakana character in the position of the underscore from a Japanese DBCS server. However, the same rows from the equivalent table in a Japanese EUC server would not be returned, since the Katakana characters will only match with a double-byte underscore.

For a LIKE predicate involving graphic strings in an EUC database:

- the character used for underscore and percent must map to the underscore and percent character respectively
- underscore represents any one UCS-2 character
- percent represents a string of zero or more UCS-2 characters.

## Japanese and Traditional-Chinese EUC Considerations

---

### Functions

#### LENGTH

The processing of this function is no different for mixed character strings in an EUC environment. The value returned is the length of the string in the code page of the argument. When using this function to determine the length of a value, careful consideration should be given to how the length is used. This is especially true for mixed string constants since the length is given in bytes, not characters. For example, the length of a mixed string column in a DBCS database returned by the LENGTH function may be less than the length of the retrieved value of that column on an eucJP or eucTW client due to the conversion of some DBCS characters to multi-byte eucJP or eucTW characters.

#### SUBSTR

The SUBSTR function operates on mixed character strings on a byte basis. The resulting string may therefore include fragments of multi-byte characters at the beginning or end of the resulting string. No processing is provided to detect or process fragments of characters.

#### TRANSLATE

The TRANSLATE function supports mixed character strings including multi-byte characters. The corresponding characters of the *to-string-exp* and the *from-string-exp* must have the same number of bytes and cannot end with part of a multi-byte character.

The *pad-char-exp* must result in a single-byte character when the *char-string-exp* is a character string. Since TRANSLATE is performed in the code page of the *char-string-exp*, the *pad-char-exp* may be converted from a multi-byte character to a single-byte character.

A *char-string-exp* that ends with part of a multi-byte character will not have those bytes translated.

#### VARGRAPHIC

The VARGRAPHIC function on a character string operand in a Japanese or Traditional-Chinese EUC code page returns a graphic string in the UCS-2 code page.

- Single-byte characters are converted first to their corresponding double-byte character in the code set to which they belong (eucJP or eucTW). Then, they are converted to the corresponding UCS-2 representation. If there is no double-byte representation, the character is converted to the double-byte substitution character defined for that code set before being converted to UCS-2 representation.
- Characters from eucJP that are Katakana (eucJP CS2) are actually single byte characters in some encoding schemes. They are thus converted to corresponding double-byte characters in eucJP or to the double-byte substitution character before converting to UCS-2.
- Multi-byte characters are converted to their UCS-2 representations.



### Statements

#### CONNECT

The processing of a successful **CONNECT** statement returns information in the **SQLCA** that is important when the possibility exists for applications to process data in an environment that includes a Japanese or Traditional-Chinese EUC code page at the client or server. The **SQLERRD(1)** field gives the maximum expansion of a mixed character string when converted from the application code page to the database code page. The **SQLERRD(2)** field gives the maximum expansion of a mixed character string when converted from the database code page to the application code page. The value is positive if expansion could occur and negative if contraction could occur. If the value is negative, the value is always -1 since the worst case is that no contraction occurs and the full length of the string is required after conversion. Positive values may be as large as 2, meaning that in the worst case, double the string length may be required for the character string after conversion.

The code page of the application server and the application client are also available in the **SQLERRMC** field of the **SQLCA**.

#### PREPARE

The data types determined for untyped parameter markers (as described in Table 22 on page 675) are not changed in an environment that includes Japanese or Traditional-Chinese EUC. As a result, it may be necessary in some cases to use typed parameter markers to provide sufficient length for mixed character strings in eucJP or eucTW. For example, consider an insert to a **CHAR(10)** column. Preparing the statement:

```
INSERT INTO T1 (CH10) VALUES (?)
```

would result in a data type of **CHAR(10)** for the parameter marker. If the client was eucJP or eucTW, more than 10 bytes may be required to represent the string to be inserted but the same string in the DBCS code page of the database is not more than 10 bytes. In this case, the statement to prepare should include a typed parameter marker with a length greater than 10. Thus, preparing the statement:

```
INSERT INTO T1 (CH10) VALUES (CAST(? AS VARCHAR(20))
```

would result in a data type of **VARCHAR(20)** for the parameter marker.

## Japanese and Traditional-Chinese EUC Considerations

---

## Appendix P. How the DB2 Library Is Structured

The DB2 Universal Database library consists of SmartGuides, online help, and books. This section describes the information that is provided, and how to access it.

To access product information online, you can use the Information Center. You can view task information, DB2 books, troubleshooting information, sample programs, and DB2 information on the Web. See "Information Center" on page 990 for details.

---

### SmartGuides

SmartGuides help you complete some administration tasks by taking you through each task one step at a time. SmartGuides are available through the Control Center. The following table lists the SmartGuides.

**Note:** Not all SmartGuides are available for the partitioned database environment.

SmartGuide	Helps you to...	How to Access...
<b>Add Database</b>	Catalog a database on a client workstation.	From the Client Configuration Assistant, click on <b>Add</b> .
<b>Create Database</b>	Create a database, and perform some basic configuration tasks.	From the Control Center, click with the right mouse button on the <b>Databases</b> icon and select <b>Create-&gt;New</b> .
<b>Performance Configuration</b>	Tune the performance of a database by updating configuration parameters to match your business requirements.	From the Control Center, click with the right mouse button on the database you want to tune and select <b>Configure performance</b> .
<b>Backup Database</b>	Determine, create, and schedule a backup plan.	From the Control Center, click with the right mouse button on the database you want to backup and select <b>Backup-&gt;Database using SmartGuide</b> .
<b>Restore Database</b>	Recover a database after a failure. It helps you understand which backup to use, and which logs to replay.	From the Control Center, click with the right mouse button on the database you want to restore and select <b>Restore-&gt;Database using SmartGuide</b> .
<b>Create Table</b>	Select basic data types, and create a primary key for the table.	From the Control Center, click with the right mouse button on the <b>Tables</b> icon and select <b>Create-&gt;Table using SmartGuide</b> .
<b>Create Table Space</b>	Create a new table space.	From the Control Center, click with the right mouse button on the <b>Table spaces</b> icon and select <b>Create-&gt;Table space using SmartGuide</b> .

---

## Online Help

Online help is available with all DB2 components. The following table describes the various types of help. You can also access DB2 information through the Information Center. For information see “Information Center” on page 990.

Type of Help	Contents	How to Access...
<b>Command Help</b>	Explains the syntax of commands in the command line processor.	From the command line processor in interactive mode, enter:  <i>? command</i>  where <i>command</i> is a keyword or the entire command.  For example, <b>? catalog</b> displays help for all the CATALOG commands, while <b>? catalog database</b> displays help for the CATALOG DATABASE command.
<b>Control Center Help</b>	Explains the tasks you can perform in a window or notebook. The help includes prerequisite information you need to know, and describes how to use the window or notebook controls.	From a window or notebook, click on the <b>Help</b> push button or press the F1 key.
<b>Message Help</b>	Describes the cause of a message, and any action you should take.	From the command line processor in interactive mode, enter:  <i>? XXXnnnnn</i>  where <i>XXXnnnnn</i> is a valid message identifier.  For example, <b>? SQL30081</b> displays help about the SQL30081 message.  To view message help one screen at a time, enter:  <i>? XXXnnnnn   more</i>  To save message help in a file, enter:  <i>? XXXnnnnn &gt; filename.ext</i>  where <i>filename.ext</i> is the file where you want to save the message help.
<b>SQL Help</b>	Explains the syntax of SQL statements.	From the command line processor in interactive mode, enter:  <b>help statement</b>  where <i>statement</i> is an SQL statement.  For example, <b>help SELECT</b> displays help about the SELECT statement.

---

Type of Help	Contents	How to Access...
<b>SQLSTATE Help</b>	Explains SQL states and class codes.	From the command line processor in interactive mode, enter:  <i>? sqlstate</i> or <i>? class-code</i>  where <i>sqlstate</i> is a valid five-digit SQL state and the <i>class-code</i> is first two digits of the SQL state.  For example, <b>? 08003</b> displays help for the 08003 SQL state, while <b>? 08</b> displays help for the 08 class code.

## DB2 Books

The table in this section lists the DB2 books. They are divided into two groups:

**Cross-platform books** These books contain the common DB2 information for UNIX-based and Intel-based platforms.

**Platform-specific books** These books are for DB2 on a specific platform. For example, for DB2 on OS/2, on Windows NT, and on the UNIX-based platforms, there are separate *Quick Beginnings* books.

Most books are available in HTML and PostScript format, and in hardcopy that you can order from IBM. The exceptions are noted in the table.

If you want to read the English version of the books, they are always provided in the directory that contains the English documentation.

You can obtain DB2 books and access information in a variety of different ways:

<b>View</b>	See "Viewing Online Books" on page 987.
<b>Search</b>	See "Searching Online Books" on page 988.
<b>Print</b>	See "Printing the PostScript Books" on page 988.
<b>Order</b>	See "Ordering the Printed DB2 Books" on page 989.

Book Name	Book Description	Form Number File Name
<b>Cross-Platform Books</b>		
<i>Administration Getting Started</i>	Introduces basic DB2 database administration concepts and tasks, and walks you through the primary administrative tasks.	S10J-8154 db2k0x50
<i>Administration Guide</i>	Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.	S10J-8157 db2d0x51
<i>API Reference</i>	Describes the DB2 application programming interfaces (APIs) and data structures you can use to manage your databases. Explains how to call APIs from your applications.	S10J-8167 db2b0x51

<b>Book Name</b>	<b>Book Description</b>	<b>Form Number</b> <b>File Name</b>
<i>CLI Guide and Reference</i>	Explains how to develop applications that access DB2 databases using the DB2 Call Level Interface, a callable SQL interface that is compatible with the Microsoft ODBC specification.	S10J-8159 db2l0x50
<i>Command Reference</i>	Explains how to use the command line processor, and describes the DB2 commands you can use to manage your database.	S10J-8166 db2n0x51
<i>DB2 Connect Enterprise Edition Quick Beginnings</i>	Provides planning, migrating, installing, configuring, and using information for DB2 Connect Enterprise Edition. Also contains installation and setup information for all supported clients.	S10J-7888 db2cyx51
<i>DB2 Connect Personal Edition Quick Beginnings</i>	Provides planning, installing, configuring, and using information for DB2 Connect Personal Edition.	S10J-8162 db2c1x51
<i>DB2 Connect User's Guide</i>	Provides concepts, programming and general using information about the DB2 Connect products.	S10J-8163 db2c0x51
<i>DB2 Connectivity Supplement</i>	Provides setup and reference information for customers who want to use DB2 for AS/400, DB2 for OS/390, DB2 for MVS, or DB2 for VM as DRDA Application Requesters with DB2 Universal Database servers, and customers who want to use DRDA Application Servers with DB2 Connect (formerly DDCS) application requesters.  <b>Note:</b> Available in HTML and PostScript formats only.	No form number db2h1x51
<i>Embedded SQL Programming Guide</i>	Explains how to develop applications that access DB2 databases using embedded SQL, and includes discussions about programming techniques and performance considerations.	S10J-8158 db2a0x50
<i>Glossary</i>	Provides a comprehensive list of all DB2 terms and definitions.  <b>Note:</b> Available in HTML format only.	No form number db2t0x50
<i>Installing and Configuring DB2 Clients</i>	Provides installation and setup information for all DB2 Client Application Enablers and DB2 Software Developer's Kits.  <b>Note:</b> Available in HTML and PostScript formats only.	No form number db2iyx51
<i>Master Index</i>	Contains a cross reference to the major topics covered in the DB2 library.  <b>Note:</b> Available in PostScript format and hardcopy only.	S10J-8170 db2w0x50
<i>Messages Reference</i>	Lists messages and codes issued by DB2, and describes the actions you should take.	S10J-8168 db2m0x51

<b>Book Name</b>	<b>Book Description</b>	<b>Form Number File Name</b>
<i>DB2 Replication Guide and Reference</i>	Provides planning, configuring, administering, and using information for the IBM Replication tools supplied with DB2.	S95H-0999 db2e0x52
<i>Road Map to DB2 Programming</i>	Introduces the different ways your applications can access DB2, describes key DB2 features you can use in your applications, and points to detailed sources of information for DB2 programming.	S10J-8155 db2u0x50
<i>SQL Getting Started</i>	Introduces SQL concepts, and provides examples for many constructs and tasks.	S10J-8156 db2y0x50
<i>SQL Reference</i>	Describes SQL syntax, semantics, and the rules of the language. Also includes information about release-to-release incompatibilities, product limits, and catalog views.	S10J-8165 db2s0x51
<i>System Monitor Guide and Reference</i>	Describes how to collect different kinds of information about your database and the database manager. Explains how you can use the information to understand database activity, improve performance, and determine the cause of problems.	S10J-8164 db2f0x50
<i>Troubleshooting Guide</i>	Helps you determine the source of errors, recover from problems, and use diagnostic tools in consultation with DB2 Customer Service.	S10J-8169 db2p0x50
<i>What's New</i>	Describes the new features, functions, and enhancements in DB2 Universal Database, Version 5.2, including information about Java-based tools.	S04L-6230 db2q0x51
<b>Platform-Specific Books</b>		
<i>Building Applications for UNIX Environments</i>	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a UNIX system.	S10J-8161 db2axx51
<i>Building Applications for Windows and OS/2 Environments</i>	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a Windows or OS/2 system.	S10J-8160 db2a1x50
<i>DB2 Personal Edition Quick Beginnings</i>	Provides planning, installing, migrating, configuring, and using information for DB2 Universal Database Personal Edition on OS/2, Windows 95, and the Windows NT operating systems.	S10J-8150 db2i1x50
<i>DB2 SDK for Macintosh Building Your Applications</i>	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a Macintosh system.  <b>Note:</b> Available in PostScript format and hardcopy for DB2 Version 2.1.2 only.	S50H-0528 sqla7x02
<i>DB2 SDK for SCO OpenServer Building Your Applications</i>	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a SCO OpenServer system.  <b>Note:</b> Available for DB2 Version 2.1.2 only.	S89H-3242 sqla9x02

Book Name	Book Description	Form Number File Name
<i>DB2 SDK for SINIX Building Your Applications</i>	Provides environment setup information and step-by-step instructions to compile, link, and run DB2 applications on a SINIX system.  <b>Note:</b> Available in PostScript format and hardcopy for DB2 Version 2.1.2 only.	S50H-0530 sqla8x00
<i>Quick Beginnings for OS/2</i>	Provides planning, installing, migrating, configuring, and using information for DB2 Universal Database on OS/2. Also contains installing and setup information for all supported clients.	S10J-8147 db2i2x50
<i>Quick Beginnings for UNIX</i>	Provides planning, installing, configuring, migrating, and using information for DB2 Universal Database on UNIX-based platforms. Also contains installing and setup information for all supported clients.	S10J-8148 db2ixx51
<i>Quick Beginnings for Windows NT</i>	Provides planning, installing, configuring, migrating, and using information for DB2 Universal Database on the Windows NT operating system. Also contains installing and setup information for all supported clients.	S10J-8149 db2i6x50
<i>DB2 Extended Enterprise Edition for UNIX Quick Beginnings</i>	Provides planning, installing, configuring, and using information for DB2 Universal Database Extended Enterprise Edition for UNIX.  This book supercedes the <b>DB2 Extended Enterprise Edition Quick Beginnings for AIX</b> book, and is suitable for use with all versions of DB2 Extended Enterprise Edition that run on UNIX-based platforms.	S99H-8314 db2v3x51
<i>DB2 Extended Enterprise Edition for Windows NT Quick Beginnings</i>	Provides planning, installing, configuring, and using information for DB2 Universal Database Extended Enterprise Edition for Windows NT.	S09L-6713 db2v6x51

**Notes:**

1. The character in the sixth position of the file name indicates the language of a book. For example, the file name db2d0e50 indicates that the *Administration Guide* is in English. The following letters are used in the file names to indicate the language of a book:

Language	Identifier	Language	Identifier
Brazilian Portuguese	B	Japanese	J
Bulgarian	U	Korean	K
Czech	X	Norwegian	N
Danish	D	Polish	P
English	E	Russian	R
Finnish	Y	Simp. Chinese	C
French	F	Slovenia	L
German	G	Spanish	Z
Greek	A	Swedish	S
Hungarian	H	Trad. Chinese	T



Italian

I

Turkish

M

2. For late breaking information that could not be included in the DB2 books:
  - On UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/%L directory, where %L is the locale name and DB2DIR is:
    - /usr/lpp/db2\_05\_00 on AIX
    - /opt/IBMDB2/V5.0 on HP-UX, Solaris, SCO UnixWare 7, and SGI.
  - On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed.

## Viewing Online Books

The manuals included with this product are in Hypertext Markup Language (HTML) softcopy format. Softcopy format enables you to search or browse the information, and provides hypertext links to related information. It also makes it easier to share the library across your site.

You can use any HTML Version 3.2-compliant browser to view the online books.

To view online books:

- If you are running DB2 administration tools, use the Information Center. See “Information Center” on page 990 for details.
- Use the open file function of your Web browser. The page you open contains descriptions of and links to DB2 books:
  - On UNIX-based platforms, open the following page:  
`file:/INSTHOME/sql1lib/doc/%L/html/index.htm`  
where %L is the locale name.
  - On other platforms, open the following page:  
`sql1lib\doc\html\index.htm`

The path is located on the drive where DB2 is installed.

You can also open the page by double-clicking on the **DB2 Online Books** icon. Depending on the system you are using, the icon is in the main product folder or the Windows Start menu.

**Note:** The **DB2 Online Books** icon is only available if you do not install the Information Center.

## Setting up a Document Server

By default the DB2 information is installed on your local system. This means that each person who needs access to the DB2 information must install the same files. To have the DB2 information stored in a single location, use the following instructions:

1. Copy all files and sub-directories from \sql1lib\doc\html on your local system to a web server. Each book has its own sub-directory containing all the necessary

HTML and GIF files that make up the book. Ensure that the directory structure remains the same.

2. Configure the web server to look for the files in the new location. For information, see **Setting up DB2 Online Documentation on a Web Server** at:

<http://www.software.ibm.com/data/pubs/papers/db2html.html>

3. If you are using the Java version of the Information Center, you can specify a base URL for all HTML files. You should use the URL for the list of books.
4. Once you are able to view the book files, you should bookmark commonly viewed topics such as:
  - List of books
  - Tables of contents of frequently used books
  - Frequently referenced articles like the **ALTER TABLE** topic
  - Search form.

For information about setting up a search, see the *What's New* book.

## Searching Online Books

To search for information in the HTML books, you can do the following:

- Click on **Search the DB2 Books** at the bottom of any page in the HTML books. Use the search form to find a specific topic.
- Click on **Index** at the bottom of any page in an HTML book. Use the Index to find a specific topic in the book.
- Display the Table of Contents or Index of the HTML book, and then use the find function of the Web browser to find a specific topic in the book.
- Use the bookmark function of the Web browser to quickly return to a specific topic.
- Use the search function of the Information Center to find specific topics. See "Information Center" on page 990 for details.

## Printing the PostScript Books

If you prefer to have printed copies of the manuals, you can decompress and print PostScript versions. For the file name of each book in the library, see the table in "DB2 Books" on page 983.

**Note:** Specify the full path name for the file you intend to print.

On OS/2 and Windows platforms:

1. Copy the compressed PostScript files to a hard drive on your system. The files have a file extension of .exe and are located in the `x:\doc\language\books\ps` directory, where `x`: is the letter representing the CD-ROM drive and `language` is the two-character country code that represents your language (for example, EN for English).
2. Decompress the file that corresponds to the book that you want. The result from this step is a printable PostScript file with a file extension of .ps.

3. Ensure that your default printer is a PostScript printer capable of printing Level 1 (or equivalent) files.
4. Enter the following command from a command line:

```
print filename.psz
```

On UNIX-based platforms:

1. Mount the CD-ROM. Refer to your *Quick Beginnings* manual for the procedures to mount the CD-ROM.
2. Change to `/cdrom/doc/%L/ps` directory on the CD-ROM, where `/cdrom` is the mount point of the CD-ROM and `%L` is the name of the desired locale. The manuals will be installed in the previously-mentioned directory with file names ending with `.ps.Z`.
3. Decompress and print the manual you require using the following command:

- For AIX:

```
zcat filename | qprt -P PSprinter_queue
```

- For HP-UX, Solaris, or SCO UnixWare 7:

```
zcat filename | lp -d PSprinter_queue
```

- For Silicon Graphics IRIX and SINIX:

```
zcat < filename | lp -d PSprinter_queue
```

where *filename* is the name of the full path name and extension of the compressed PostScript file and *PSprinter\_queue* is the name of the PostScript printer queue.

For example, to print the English version of *Quick Beginnings for UNIX* on AIX, you can use the following command:

```
zcat /cdrom/doc/en/ps/db2ixe50.ps.Z | qprt -P ps1
```

## Ordering the Printed DB2 Books

You can order the printed DB2 manuals either as a set, or individually. There are three sets of books available. The form number for the entire set of DB2 books is SB0F-8915-00. The form number for the set of books updated for Version 5.2 is SB0F-8921-00. The form number for the books listed under the heading "Cross-Platform Books" is SB0F-8914-00.

**Note:** These form numbers only apply if you are ordering books that are printed in the English language.

You can also order books individually by the form number listed in "DB2 Books" on page 983. To order printed versions, contact your IBM authorized dealer or marketing representative, or phone 1-800-879-2755 in the United States or 1-800-IBM-4YOU in Canada.

---

## Information Center

The Information Center provides quick access to DB2 product information. You must install the DB2 administration tools to obtain the Information Center.

Depending on your system, you can access the Information Center from the:

- Main product folder
- Toolbar in the Control Center
- Windows Start menu
- Help menu of the Control Center
- **db2ic** command.

The Information Center provides the following kinds of information. Click on the appropriate tab to look at the information:

<b>Tasks</b>	Lists tasks you can perform using DB2.
<b>Reference</b>	Lists DB2 reference information, such as keywords, commands, and APIs.
<b>Books</b>	Lists DB2 books.
<b>Troubleshooting</b>	Lists categories of error messages and their recovery actions.
<b>Sample Programs</b>	Lists sample programs that come with the DB2 Software Developer's Kit. If the Software Developer's Kit is not installed, this tab is not displayed.
<b>Web</b>	Lists DB2 information on the World Wide Web. To access this information, you must have a connection to the Web from your system.

When you select an item in one of the lists, the Information Center launches a viewer to display the information. The viewer might be the system help viewer, an editor, or a Web browser, depending on the kind of information you select.

The Information Center provides some search capabilities so you can look for specific topics, and filter capabilities to limit the scope of your searches.

For a full text search, follow the **Search DB2 Books** link in each HTML file, or use the search feature of the help viewer.

The HTML search server is usually started automatically. If a search in the HTML information does not work, you may have to start the search server via its icon on the Windows or OS/2 desktop.

Refer to the release notes if you experience any other problems when searching the HTML information.

---

## Appendix Q. Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the

IBM Director of Licensing,  
IBM Corporation,  
500 Columbus Avenue,  
Thornwood, NY, 10594  
USA.

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 Canada Limited  
Department 071  
1150 Eglinton Ave. East  
North York, Ontario  
M3C 1H7  
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

---

## Trademarks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries:

ACF/VTAM	MVS/ESA
ADSTAR	MVS/XA
AISPO	NetView
AIX	OS/400
AIXwindows	OS/390
AnyNet	OS/2
APPN	PowerPC
AS/400	QMF
CICS	RACF
C Set++	RISC System/6000
C/370	SAA
DATABASE 2	SP
DatagLANce	SQL/DS
DataHub	SQL/400
DataJoiner	S/370
DataPropagator	System/370
DataRefresher	System/390
DB2	SystemView
Distributed Relational Database Architecture	VisualAge
DRDA	VM/ESA
Extended Services	VSE/ESA
FFST	VTAM
First Failure Support Technology	WIN-OS/2
IBM	
IMS	
Lan Distance	

---

## Trademarks of Other Companies

The following terms are trademarks or registered trademarks of the companies listed:

C-bus is a trademark of Corollary, Inc.

HP-UX is a trademark of Hewlett-Packard.

Java, HotJava, Solaris, Solstice, and Sun are trademarks of Sun Microsystems, Inc.

Microsoft, Windows, Windows NT, Visual Basic, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

SCO is a trademark of The Santa Cruz Operation.

SINIX is a trademark of Siemens Nixdorf.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, or service names, which may be denoted by a double asterisk (\*\*), may be trademarks or service marks of others.

---

## Appendix R. Contacting IBM

This section lists ways you can get more information from IBM.

If you have a technical problem, please take the time to review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. Depending on the nature of your problem or concern, this guide will suggest information you can gather to help us to serve you better.

For information or to order any of the DB2 Universal Database products contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

### Telephone

If you live in the U.S.A., call one of the following numbers:

- 1-800-237-5511 to learn about available service options.
- 1-800-IBM-CALL (1-800-426-2255) or 1-800-3IBM-OS2 (1-800-342-6672) to order products or get general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, see Appendix A of the IBM Software Support Handbook. You can access this document by accessing the following page:

<http://www.ibm.com/support/>

then performing a search using the keyword "handbook."

Note that in some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.

### World Wide Web

<http://www.software.ibm.com/data/> <http://www.software.ibm.com/data/db2/library/>

The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more. The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information. (Note that this information may be in English only.)

### Anonymous FTP Sites

<ftp.software.ibm.com>

Log on as anonymous. In the directory `/ps/products/db2`, you can find demos, fixes, information, and tools concerning DB2 and many related products.

### Internet Newsgroups

<comp.databases.ibm-db2>, <bit.listserv.db2-l>

These newsgroups are available for users to discuss their experiences with DB2 products.

**CompuServe**

**GO IBMDB2** to access the IBM DB2 Family forums

All DB2 products are supported through these forums.

To find out about the IBM Professional Certification Program for DB2 Universal Database, go to <http://www.software.ibm.com/data/db2/db2tech/db2cert.html>



---

## Index

### Special Characters

? (question mark) 626

*See also* parameter marker

\* (asterisk)

in subselect column names 317

naming columns, use in select 317

### A

ABS or ABSVAL function 156

detailed format description 186

values and arguments, rules for 186

ACOS function 156

detailed format description 187

values and arguments, rules for 187

ADD clause in ALTER TABLE 384

ADD column clause, order of processing 394

alias

comment descriptions, adding to catalog 418

CREATE ALIAS statement 446

deleting, using DROP statement 611

description 11, 51

TABLE\_NAME function 289

TABLE\_SCHEMA function 291

ALIAS clause

COMMENT ON statement 419

DROP statement 612

alias-name 48

ALL clause

SELECT statement, use in 317, 329

ALL in quantified predicate 137

ALL option

comparison, set operator, effect on 351

ALL PRIVILEGES clause

GRANT statement (table or view) 653

REVOKE statement, table and view privileges 697

ALTER BUFFERPOOL statement 375, 376

ALTER clause

GRANT statement (table or view) 653

REVOKE statement, removing privilege for 698

ALTER NODEGROUP statement 377, 379

ALTER TABLE statement 398

authorization required, summary 380

examples of usage 396

syntax diagram 384

ALTER TABLESPACE statement 399, 402

ALTER TYPE (Structured) statement 403, 404

ALTER VIEW statement 406

authorization required, summary 405

syntax diagram 405

ambiguous cursor 598

ambiguous reference, error conditions for 102

AND truth table 153

ANY in quantified predicate 137

application process, definition of 19

application program

concurrency 19

uses SQLDA 763

application requester, overview 24

application server

overview 24

role of in connections 25

arguments of COALESCE

result data type 82

arithmetic

AVG function, operation of 171

columns, adding values (SUM) 183

constants

definition of 88

NOT NULL, required attribute for 88

date operations, rules for 126

datetime, SQL rules for 124

decimal operations, scale and precision

formulas 122

decimal value, precision and scale 59

decimal values from numeric expressions 213

distinct type operands 122

expressions, adding values (SUM) 183

floating point operands

rules and precision values 122

with integers, result of 122

floating point values from numeric expressions 229,

273

floating point, range and precision 59

integer

large integer, range and precision 59

small integer, range and precision 59

integer values, returning from expressions 192, 242

maximum value, finding 178

minimum value, finding 180

operators, summary of results 120

- arithmetic (*continued*)
  - parameter marker, syntax and operations 674
  - remote use of, conversions, overview 34
  - small integer values, returning from expressions 282
  - STDDEV function, operation of 182
  - time operations, rules for 126, 127
  - timestamp operations, rules for 127
  - unary minus sign, effect on operand 120
  - unary plus sign, effect on operand 120
  - VARIANCE function, operation of 184
- AS clause
  - CREATE VIEW statement 584
  - in SELECT clause 317, 319
  - ORDER BY clause 358
- ASC clause
  - CREATE INDEX statement 505
  - of select-statement 359
- ASCII function 156
  - detailed format description 188
  - values and arguments, rules for 188
- ASIN function 156
  - detailed format description 189
  - values and arguments, rules for 189
- Assembler application host variable 631
- assigning a string to a column, rules for 72
- assignments
  - character strings to datetime columns, rules for 70
  - DATALINK type 75
  - datetime to character string value 75
  - datetime values, rules for 75
  - fragmenting a MBCS character, rules for 74
  - mixed character string blank padding 74
  - mixed character string to host variables 74
  - mixed character string truncation 74
  - numbers 72
  - reference type 77
  - retrieval 73
  - storage 73, 131
  - strings, basic rules for 72
  - user-defined type 77
- asterisk (\*)
  - in COUNT 173
  - in COUNT\_BIG 174
  - in subselect column names 317
- ATAN function 156
  - detailed format description 190
  - values and arguments, rules for 190
- ATAN2 function 156
  - detailed format description 191

- ATAN2 function (*continued*)
  - values and arguments, rules for 191
- attribute-name 48
  - in dereference operation 134
- authority level
  - authorization name, syntax rules for 48
- authorization
  - definition 38
  - granting control on database operations 641
  - granting control on index 644
  - granting create on schema 649
  - public control on index 644
  - public create on schema 649
- authorization ID, overview of 51
- authorization-name
  - restrictions governing 48
  - use of in Grant and Revoke 51
- AVG function 156
- AVG function, detailed description 171

## B

- base table 10
- basic operations in SQL 70
- basic predicate, detailed format 136
- BEGIN DECLARE SECTION statement 407, 408
  - authorization required 407
  - invocation rules for 407
- BETWEEN predicate, detailed format diagram 140
- big integers 59
- BIGINT data type 530
  - description 59
  - precision 59
  - range 59
- BIGINT function 156
- BIGINT function, integer values from expressions 192
- binary integer, as data type 54
- Binary Large Object 55
  - See also* BLOB
- BINDADD parameter, GRANT...ON DATABASE statement 641
- binding 646
  - bound statement, overview of 25
  - data retrieval, role in optimizing 7
  - revoking all privileges 692
- bit data
  - BLOB string 55
  - definition 57
- blank 46

- blanks
  - definition of 45
- BLOB
  - data type 531
  - scalar function description 193
  - string 55
- BLOB function 156
- bound statement, use of 25
- buffer insert 664
- buffer pool
  - deleting, using DROP statement 611
  - description 40
  - extended storage use 375, 450
  - page size 450
  - setting size 375, 450
- bufferpool
  - naming conventions 48
- BUFFERPOOL clause
  - ALTER TABLESPACE statement 401
  - CREATE TABLESPACE statement 564
  - DROP statement 612
- built-in function 155
  - See also* function description 110
- byte length values, list for data types 246

**C**

- caching
  - EXECUTE statement 627
- call level interface 8
- CALL statement 409, 415
- cancelling a unit of work 702
- CASCADE delete rule 542
  - description 15
- case
  - expression 129
- case sensitive identifiers, SQL 47
- CAST
  - expression as operand 131
  - NULL as operand 131
  - parameter marker as operand 131
- CAST specification 131
- casting
  - between data types 67
  - reference types 68
  - user-defined types 67
- catalog
  - adding comments on tables, views, columns 418
  - COMMENT ON, detailed syntax 418

- catalog views
  - BUFFERPOOLNODES 777
  - BUFFERPOOLS 776
  - CHECKS 778
  - COLAUTH 779
  - COLCHECKS 780
  - COLDIST 781
  - COLUMNS 782
  - CONSTDEP 784
  - DATATYPES 785
  - DBAUTH 786
  - definition 19
  - EVENTMONITORS 787
  - EVENTS 788
  - FUNCPARMS 789
  - FUNCTIONS 790
  - INDEXAUTH 793
  - INDEXES 794
  - KEYCOLUSE 797
  - NODEGROUPDEF 798
  - NODEGROUPS 799
  - overview 773
  - PACKAGEAUTH 800
  - PACKAGEDEP 801
  - PACKAGES 802
  - PARTITIONMAPS 805
  - PROCEDURE PARAMETERS 807
  - PROCEDURES 806
  - read-only 774, 832
  - REFERENCES 808
  - SCHEMAAUTH 809
  - SCHEMATA 810
  - STATEMENTS 811
  - SYSSTAT.COLUMNS 824
  - SYSSTAT.FUNCTIONS 825
  - SYSSTAT.INDEXES 827
  - SYSSTAT.TABLES 830
  - TABAUTH 812
  - TABCONST 814
  - TABLES 815
  - TABLESPACES 818
  - TRIGDEP 819
  - TRIGGERS 820
  - updatable 774
  - VIEWDEP 821
  - VIEWS 822
- catalog views (structured types)
  - ATTRIBUTES 835
  - CHECKS 836
  - COLCHECKS 837

- catalog views (structured types) *(continued)*
  - COLUMNS 838
  - CONSTDEP 841
  - DATATYPES 842
  - FUNCPARMS 843
  - FUNCTIONS 844
  - HIERARCHIES 847
  - INDEXES 848
  - KEYCOLUSE 851
  - OBJSTAT.TABLES 861
  - overview 831
  - PACKAGEDEP 852
  - REFERENCES 853
  - TABCONST 854
  - TABLES 855
  - TRIGDEP 858
  - TRIGGERS 859
  - updatable 832
  - VIEWDEP 860
- CEIL or CEILING function 156
- CEIL or CEILINGfunction
  - detailed format description 194
  - values and arguments, rules for 194
- CHAR
  - function description 195
- CHAR function 157
- CHAR VARYING data type 531
- character conversion
  - character set 35
  - code page 35
  - code point 36
  - encoding scheme 36
  - rules for assignments 73
  - rules for comparison 79
  - rules for operations combining strings 85
  - rules when comparing strings 85
- CHARACTER data type 531
- Character Large Object 55
  - See also* CLOB string
- character set 35
- character string
  - arithmetic operators, prohibited use of 120
  - as data type 54
  - assignment, overview 72
  - bit data
    - definition 57
  - BLOB string representation 193
  - CLOB 55
  - comparisons, rules for 78
  - constants, range and precision 89
  - character string *(continued)*
    - detailed description 56
    - double byte character string, returning 308
    - empty, compared to null value 56
    - equality, collating sequence examples 78
    - equality, definition of 78
    - fixed length 54
    - fixed length, description 56
    - hexadecimal constant 90
    - mixed data 57
    - POSSTR scalar function 265
    - returning from host variable name 299
    - SBCS data, definition 57
    - SQL statement string, rules for creating 631
    - SQL statement, execution as 631
    - translating string syntax 299
    - VARCHAR scalar function, using 307
    - VARGRAPHIC scalar function, using 308
    - varying length 54
    - varying length, description 56
- CHARACTER VARYING data type 531
- characters, SQL, range of 45
- CHECK clause in CREATE VIEW statement 587
- check constraint
  - ALTER TABLE statement 388, 390
  - CREATE TABLE statement 545
  - INSERT statement 664
- check pending state 16, 709
- CHR function 157
  - detailed format description 200
  - values and arguments, rules for 200
- CL\_SCHED sample table 864
- client/server
  - server name, description of 50
- CLOB data type 532
- CLOB function 157
  - detailed format description 201
  - values and arguments, rules for 201
- CLOB string 55
- CLOSE statement 416, 417
- closed state of cursor 670
- CLUSTER clause
  - CREATE INDEX statement 506
- COALESCE
  - function description 202
- COALESCE function 157
  - code page 35
  - code point 36
  - collating sequence, string comparison, rules for 78

## column

- adding to a table, ALTER TABLE 384
- adding values (SUM) 183
- adding with ALTER TABLE statement 380
- adding, privileges for, granting 653
- ambiguous name reference, error conditions 102
- averaging a column set of values (AVG) 171
- BASIC predicate, use in matching strings 136
- BETWEEN predicate, use in matching strings 140
- column name, qualified, conditions for 103
- column name, unqualified, conditions for 103
- comment descriptions, adding to catalog 418
- constraint name, FOREIGN KEY, rules 542
- definition of 10
- DISTINCT keyword, queries, role of 170
- EXISTS predicate, use in matching strings 142
- fixed length character strings, attributes 56
- GROUP BY, use in limiting in SELECT clause 318
- grouping column name, use in GROUP BY 329
- HAVING clause, search names, rules for 336
- HAVING, use in limiting in SELECT clause 318
- IN predicate, fullselect, values returned 143
- index key, column-name, use in 505
- inserting values, INSERT statement 662
- LIKE predicate, use in matching strings 146
- maximum value, finding 178
- minimum value, finding 180
- naming conventions 48
- naming conventions, applications of
  - in CREATE INDEX statement 98
  - in CREATE TABLE statement 98
  - in expressions 98
  - in GROUP BY or ORDER BY statements 98
- nested table expression, use of 103
- null values in result columns, rules for 319
- null values, ALTER TABLE, prevention of 384
- qualified column names, rules for 98
- result data, expression type, table of 320
- scalar fullselect, use of 103
- searching using WHERE clause 328
- SELECT clause, select list notation 317
- standard deviation of a column set of values (STDDEV) 182
- string assignment, basic rules for 72
- subquery, use of 102
- undefined name reference, error conditions 102
- updating row values, UPDATE statement 740
- variance of a column set of values (VARIANCE) 184
- varying length character strings, attributes 56

## COLUMN clause

- COMMENT ON statement 419
- column function, arguments for 155
- column name
  - in ORDER BY clause 358
  - rules for 48
- column name qualification in the COMMENT ON statement 98
- column name, uses for 98
- column options
  - CREATE TABLE statement 534
- column-name
  - in INSERT statement 662
- combining grouping sets 333
- comment
  - SQL static statements, use in 374
- comment in catalog table 418
- COMMENT ON statement 418, 425
- comments
  - host language, format for 47
  - SQL, format for 47
- commit processing
  - locks, relation to uncommitted changes 20
- COMMIT statement 426, 427
- common table expression 356
  - description 19
  - recursive 357
- common-table-expression
  - select-statement 356
- comparing a value with a collection 140
- comparing LONG VARCHAR strings, restricted use of 80
- comparing two predicates, truth conditions 136, 152
- comparison
  - compatibility rules 70
  - compatibility rules, data types, summary 70
  - datetime values, rules for 81
  - graphic strings, rules for 80
  - LONG VARCHAR, restricted use of 80
  - numbers, rules for 78
  - reference type 82
  - SBCS/MBCS, rules for 79
  - strings, rules for 78
  - user-defined type 81
- compatibility
  - data types 70
  - data types, summary 70
  - rules 70
  - rules for operation types 70

- composite column value 332
- composite key 12
- Compound SQL statement 431
  - combining statements into a block 428
- CONCAT function
  - detailed format description 203
  - values and arguments, rules for 203
- CONCAT or || function 157
- concatenation
  - distinct type 120
  - operator 117
  - result data type 119
  - result length 119
- concurrency
  - application 19
  - prevention of
    - LOCK TABLE statement 666
  - tables with NOT LOGGED INITIALLY parameter, restriction 549
- CONNECT parameter, GRANT...ON DATABASE statement 641
- CONNECT statement
  - disconnecting from current server 437
  - implicit connect 432
  - IMPLICIT connect, diagram of state transitions 27
  - information on application server, getting 437
  - information on setting a new password 438
  - non-IMPLICIT connect, diagram of state transitions 29
  - overview 25
  - with no operand, returning information 437
- CONNECT statement (Type 1) 432, 438
- CONNECT statement (Type 2) 439, 445
- CONNECT TO statement
  - successful connection, detailed description 434, 439
  - unsuccessful connection, detailed description 436, 440
- connected state 32
- connection states
  - application process 31
  - distributed unit of work 30
  - remote unit of work 26
- constants
  - character string, range and precision 89
  - decimal 89
  - floating-point, rules for 89
  - hexadecimal 90
  - integer, definition of 89
  - with user-defined types 91
- constants, overview of 88
- constraint
  - comment descriptions, adding to catalog 418
  - Explain tables 935
  - referential 12, 14
  - table check 12, 16
  - unique 12, 13
- CONSTRAINT clause
  - COMMENT ON statement 420
- constraints
  - adding or dropping, ALTER TABLE 380
- container
  - CREATE TABLESPACE statement 561
  - description 40
- container-clause
  - CREATE TABLESPACE statement 562
- CONTINUE clause in WHENEVER statement 750
- CONTROL clause
  - GRANT statement (table or view) 654
- CONTROL clause in GRANT statement, revoking 698
- CONTROL parameter
  - revoking privileges for packages 692
- conversion
  - character string to executable SQL 631
  - datetime to character string variable 75
  - integer to decimal, mixed expression, rules 121
- conversion rules
  - for assignments 73
  - for comparison 79
  - for operations combining strings 85
  - for string comparisons 85
- conversions
  - CHAR, returning converted datetime values 195
  - character string to timestamp 295
  - DBCS from mixed SBCS and DBCS 308
  - decimal values from numeric expressions 213
  - double byte character string, returning 308
  - floating point values from numeric expressions 229, 273
  - numeric, scale and precision, summary 72
- correlated reference 328
- correlated reference, use in nested table expression 103
- correlated reference, use in scalar fullselect 103
- correlated reference, use in subquery 102, 103
- correlation name
  - FROM clause, subselect, rules for use 321
- correlation-name
  - detailed description 49
  - in SELECT clause, syntax diagram 317

- correlation-name (*continued*)
  - qualified reference of column name 99
- correlation-name, rules for 99
- COS function 157
  - detailed format description 204
  - values and arguments, rules for 204
- COT function 157
  - detailed format description 205
  - values and arguments, rules for 205
- COUNT function 157
  - detailed format description 173
  - values and arguments, rules for 173
- COUNT\_BIG function 157, 174
  - detailed format description 174
  - values and arguments, rules for 174
- CREATE ALIAS statement 446, 448
- CREATE BUFFERPOOL statement 449, 451
- CREATE DISTINCT TYPE statement 452, 457
- CREATE EVENT MONITOR statement 458, 466
- CREATE FUNCTION (External Scalar) statement 468
- CREATE FUNCTION (External Table) statement 484
- CREATE FUNCTION (Sourced Scalar) statement 503
- CREATE FUNCTION (Sourced) statement 497
- CREATE FUNCTION statement 467, 483, 496
- CREATE INDEX statement 504, 507
  - column-name, rules for key creation 505
- CREATE NODEGROUP statement 508, 510
- CREATE SCHEMA statement 519, 521
- CREATE TABLE statement 522, 558
  - syntax diagram 523
- CREATE TABLESPACE statement 559, 567
- CREATE TRIGGER statement 568, 577
- CREATE TYPE (Structured) statement 578, 581
- CREATE VIEW statement 582, 594
- CREATE VIEW statement, definition of 10
- CREATETAB parameter, GRANT...ON DATABASE statement 641
- creating a database, granting authority for 642
- cross tabulation rows 332
- CS (cursor stability) isolation level 23, 885
- CUBE 332
  - examples of 342
- current connection state 32
- CURRENT DATE special register 91
- CURRENT DEGREE special register 91
  - SET CURRENT DEGREE statement 716
- CURRENT EXPLAIN MODE special register 92
  - SET CURRENT EXPLAIN MODE statement 718
- CURRENT EXPLAIN SNAPSHOT special register 93
  - SET CURRENT EXPLAIN SNAPSHOT statement 720
- CURRENT FUNCTION PATH special register 94
  - SET CURRENT FUNCTION PATH statement 731
  - SET CURRENT PATH statement 731
  - SET PATH statement 731
- CURRENT NODE special register 94
- CURRENT PATH special register 94
  - SET CURRENT FUNCTION PATH statement 731
  - SET CURRENT PATH statement 731
  - SET PATH statement 731
- CURRENT QUERY OPTIMIZATION special register 95
  - SET CURRENT QUERY OPTIMIZATION statement 724
- CURRENT REFRESH AGE special register 95
- CURRENT SCHEMA special register 96
- CURRENT SERVER special register 96
- CURRENT SQLID special register 96
- CURRENT TIME special register 96
- CURRENT TIMESTAMP special register 97
- CURRENT TIMEZONE special register 97
- cursor
  - See also* DECLARE CURSOR statement
  - active set, associated with 668
  - ambiguous 598
  - closed state, pre-conditions for 670
  - closing, CLOSE statement 416
  - current row 638
  - declaring, SQL statement syntax for 595
  - defining 595
  - deleting, search condition details 601
  - location in table, results of FETCH 637
  - moving position, using FETCH 637
  - opening a cursor, OPEN statement 668
  - positions for open 638
  - preparing for application use 668
  - program usage, rules for 597
  - read-only status, conditions for 597
  - result table, relation to 595
  - terminating for unit of work, ROLLBACK 702
  - unit of work, conditional states of 595
  - updatability, determining 597
  - WITH HOLD lock clause, COMMIT statement, effect 426
- cursor stability 23, 885
- cursor-name, definition of 49

**D**

- data integrity
  - concurrent updates, preventing, LOCK TABLE 666
  - point of consistency, example of 20

- data representation considerations 34
- data structure
  - column, definition of 10
  - constants
    - character string, rules for 89
    - decimal, rules for 89
    - floating point, rules for 89
    - graphic string (DBCS), rules for 89
    - integer, rules for 89
  - date syntax and range 60
  - index, derived values of 11
  - numeric data, overview 59
  - packed decimal 770
  - row, definition of 10
  - time syntax and range 60
  - value, definition of 10
  - values
    - data types 54
    - sources 54
- data type 87
  - abstract 403, 578
  - ALTER TYPE (Structured) statement 403
  - character string 56
  - CREATE TYPE (Structured) statement 578
  - datalink 63
  - datetime 60
  - distinct 64, 452
  - overview 54
  - partition compatibility 87
  - reference 65
  - result column data, SELECT, table of 320
  - result columns 320
  - row 403, 578
  - structured 65, 403, 578
  - TYPE\_ID function 302
  - TYPE\_NAME function 303
  - TYPE\_SCHEMA function 304
  - user-defined 64
- data types
  - casting between 67
  - promotion 66
- database access
  - authority to access database, granting 641
- database administration privilege 39
- database identifier in SQL 47
- database managed space 39
  - See also DMS table space
- database management
  - control, granting authority, SQL statement for 641
  - DBADM creation authority, granting 642
- database management (*continued*)
  - saving changes, COMMIT statement 426
  - switching tasks, COMMIT statement 426
- database manager
  - catalog views
    - overview of 19
  - distributed relational database, use in 24
  - limits 753
  - SQL, interpretation of 7
- database manager limits 755
- database manager page size specific limits 757
- database-containers
  - CREATE TABLESPACE statement 562
- datalink
  - building 227
  - extracting comment 220
  - extracting complete URL 222
  - extracting file server 226
  - extracting linktype 221
  - extracting path and file name 223, 224
  - extracting scheme 225
  - INSERT statement 664
- datalink type
  - description 63
- date
  - arithmetic operations 125
  - CHAR, use of in format conversion 195
  - day durations, finding from range (DAYS) 211
  - day, returning from value (DAY function) 207
  - duration, format of 123
  - month, returning from datetime value 258
  - strings 61
  - value to date, format conversion (DATE) 206
  - WEEK scalar function, using 310
  - year, using in expressions 311
- DATE data type 532
- DATE function 157
- DATE function, returning dates from values 206
- datetime
  - arithmetic operations 124
  - data types
    - description 60
    - string representation 60, 61
  - format
    - EUR, ISO, JIS, LOCAL, USA 61
  - limits 754
  - VARCHAR scalar function, using 307
- datetime format 61
- DAY function 158



- DAY function, returning day part of values 207
- DAYNAME function 158
  - detailed format description 208
  - values and arguments, rules for 208
- DAYOFMONTH function 158
- DAYOFWEEK function 158
  - detailed format description 209
  - values and arguments, rules for 209
- DAYOFYEAR function 158
  - detailed format description 210
  - values and arguments, rules for 210
- DAYS function 158
- DAYS function, returning integer durations 211
- DB2 library
  - books 983
  - Information Center 990
  - language identifier for books 986
  - late breaking information 987
  - online help 982
  - ordering printed books 989
  - printing PostScript books 988
  - searching online books 988
  - setting up document server 987
  - SmartGuides 981
  - structure of 981
  - viewing online books 987
- db2nodes.cfg
  - ALTER NODEGROUP 378
  - CONNECT (Type 1) 438
  - CREATE NODEGROUP 508
  - CURRENT NODE 94
  - NODENUMBER function 260
- DBADM parameter, GRANT...ON DATABASE statement 642
- DBCLOB data type 532
- DBCLOB function 158
  - detailed format description 212
  - values and arguments, rules for 212
- DBCLOB string 55
- decimal
  - arithmetic formulas, scale and precision 122
  - constants, range and precision 89
  - data type, overview 59
  - implicit decimal point 59
  - numbers 59
  - packed decimal 59
- decimal conversion from integer, summary 72
- DECIMAL function, returning decimal values 213
- DECIMAL or DEC function 158
- decimal, as data type 54
- declaration
  - inserting into a program 659
- DECLARE
  - BEGIN DECLARE SECTION statement 407
  - END DECLARE SECTION statement 624
- DECLARE CURSOR statement 595, 598
  - authorization, conditions for 595
  - program usage, notes for 597
- decrementing a date, rules for 125
- decrementing a time, rules for 127
- default value
  - column
    - ALTER TABLE statement 385
    - CREATE TABLE statement 534
- DEGREES function 159
  - detailed format description 216
  - values and arguments, rules for 216
- deletable
  - view 589
- DELETE clause
  - GRANT statement (table or view) 654
  - REVOKE statement, revoking privilege for 698
- delete rule for referential constraint 16
- DELETE statement 599, 603
  - authorization, searched or positioned format 599
- delete-connected table 16
- deleting SQL objects 611
- delimited identifier in SQL 47
- delimited identifier, SQL statement 47
- delimiter tokens, definition of 46
- DEPARTMENT sample table 864
- dependency
  - of objects on each other 618
- dependent row 14
- dependent table 14
- DEREF function 159
  - reference type 217
- dereference operation 134
  - attribute-name operand 134
  - scoped-ref-expression 134
- DESC clause
  - CREATE INDEX statement 505
  - of select-statement 359
- descendent row 14
- descendent table 14
- DESCRIBE statement 604, 607
  - prepared statements, destruction conditions 605
- DESCRIPTOR
  - host variables, parameter substitution list 626

- descriptor-name 49
  - in FETCH statement 638
- diagnostic string
  - in RAISE\_ERROR function 270
  - in SIGNAL SQLSTATE statement 738
- DIFFERENCE function 159
  - detailed format description 218
  - values and arguments, rules for 218
- DIGITS function 159, 219
- digits, range of 46
- dirty read 885
- DISCONNECT statement 608, 610
- DISTINCT clause
  - of subselect 317
- DISTINCT keyword
  - AVG function, relation to 171
  - column function 170
  - COUNT function, relationship to 173
  - COUNT\_BIG function, relationship to 174
  - MAX function, restriction for 178
  - MIN function 180
  - STDDEV function, relation to 182
  - SUM function 183
  - VARIANCE function, relation to 184
- DISTINCT keyword, overview 170
- distinct type
  - as arithmetic operands 122
  - comparison 81
  - concatenation 120
  - constants 91
  - CREATE DISTINCT TYPE statement 452
  - description 49, 64
  - DROP statement 611
- DISTINCT TYPE clause
  - COMMENT ON statement 423
  - DROP statement 617
- distributed relation database, definition 24
- distributed relational database
  - application requester, overview 24
  - application server, overview 24
  - data representation considerations 34
  - environment, illustration of 24
  - remote unit of work, overview 26
  - requester-server protocols, overview 24
- distributed relational database architecture (DRDA) 24
- DLCOMMENT function 159
- DLCOMMENT function, extracting comment from
  - DATALINK value 220
- DLLINKTYPE function 159
  - extracting linktype from
    - DATALINK value 221
- DLURLCOMPLETE function 159
- DLURLCOMPLETE function, extracting complete URL
  - from DATALINK value 222
- DLURLPATH function 159
- DLURLPATH function, extracting path and file name
  - from DATALINK value 223
- DLURLPATHONLY function 159
- DLURLPATHONLY function, extracting path and file
  - name from DATALINK value 224
- DLURLSCHEME function 159
- DLURLSCHEME function, extracting scheme from
  - DATALINK value 225
- DLURLSERVER function 159
- DLURLSERVER function, extracting file server from
  - DATALINK value 226
- DLVALUE function 159
- DLVALUE function, building a DATALINK value 227
- DMS table space
  - CREATE TABLESPACE statement 562
  - description 39
- dormant connection state 32
- DOUBLE
  - CHAR, use of in format conversion 195
  - double byte character string (DBCS), returning 308
- DOUBLE data type 530
  - precision 59
  - range 59
- DOUBLE function 159
- DOUBLE function, double precision conversion 229
- DOUBLE or DOUBLE\_PRECISION function 159
- DOUBLE PRECISION data type 531
- double precision float data type 530
- double-byte character
  - truncated during assignment 74
- Double-Byte Character Large Object 55
  - See also* DBCLOB string
- double-precision floating-point 59
- DRDA (Distributed Relational Database
  - Architecture) 24
- DROP CHECK clause of ALTER TABLE statement 391
- DROP CONSTRAINT clause of ALTER TABLE
  - statement 391
- DROP FOREIGN KEY clause 391
- DROP PARTITIONING KEY clause of ALTER TABLE
  - statement 391
- DROP PRIMARY KEY clause 391
- DROP statement 611, 623

- DROP UNIQUE clause 391
- duration
  - adding, results of 125
  - date, format of 123
  - labeled 123
  - subtracting, results of 125
  - time, format of 124
  - timestamp 124
- durations 123
- dynamic select
  - host variables, restrictions on 372
  - parameter markers, usage in 372
- dynamic SQL 8, 763
- DECLARE CURSOR statement, usage in 372
  - definition of 7
  - description, preparation methods 370
  - execution 371
  - FETCH statement, usage in 372
  - OPEN statement, usage in 372
  - preparation 371
  - PREPARE statement, execution of 673
  - PREPARE statement, usage in 372
  - prepared statement information, using
    - DESCRIBE 604
  - preparing and executing, commands for 7
  - SQLDA used with 763

## E

- embedded SQL statement
  - executing character strings, EXECUTE
    - IMMEDIATE 631
- embedded SQL, requirements overview 370
- EMP\_ACT sample table 867
- EMP\_PHOTO sample table 869
- EMP\_RESUME sample table 869
- EMPLOYEE sample table 865
- empty character string 56
- encoding scheme 36
- END DECLARE SECTION statement 624, 625
- erasing the sample database 864
- error
  - closes cursor 670
  - during FETCH 638
  - during UPDATE, 744
  - return code, language overview 373
- errors
  - executing triggers 574
- escape character in SQL 47

- ESCAPE clause
  - LIKE predicate 148
- EUC considerations 973
- EUR 61
  - See also* datetime format
- European (EUR) date format 61
- European (EUR) time format 61
- evaluation order, expressions 128
- event monitor
  - CREATE EVENT MONITOR statement 458
  - description 18
  - DROP statement 611
  - EVENT\_MON\_STATE function 231
  - name description 49
  - SET EVENT MONITOR STATE statement 729
- EVENT\_MON\_STATE function 159, 231
- EXCEPT clause of fullselect 351
- except-on-nodes-clause
  - CREATE BUFFERPOOL statement 450
- exception tables 712
  - SET CONSTRAINTS statement 712
  - structure 967
- EXCLUSIVE
  - IN EXCLUSIVE MODE 432
- exclusive locks 22
- EXCLUSIVE option, LOCK TABLE statement 666
- executable statement, methods overview 369
- executable statement, processing summary 370
- EXECUTE IMMEDIATE statement 632
  - detailed instructions for 631
  - embedded usage, detailed description 371
  - use in dynamic SQL 7
- EXECUTE statement 630
  - detailed instructions for 626
  - embedded usage, detailed description 371
  - use in dynamic SQL 7
- executing, revoking package privileges 692
- execution
  - package, necessary privileges, granting 646
- EXISTS predicate, detailed format description 142
- EXP function 159
  - detailed format description 232
  - values and arguments, rules for 232
- EXPLAIN statement 633, 636
- EXPLAIN\_ARGUMENT table 935
- EXPLAIN\_ARGUMENT table definition 950
- EXPLAIN\_INSTANCE table 939
- EXPLAIN\_INSTANCE table definition 951
- EXPLAIN\_OBJECT table 940

- EXPLAIN\_OBJECT table definition 952
- EXPLAIN\_OPERATOR table 942
- EXPLAIN\_OPERATOR table definition 953
- EXPLAIN\_PREDICATE table 944
- EXPLAIN\_PREDICATE table definition 954
- EXPLAIN\_STATEMENT table 946
- EXPLAIN\_STATEMENT table definition 955
- EXPLAIN\_STREAM table 948
- EXPLAIN\_STREAM table definition 956
- explainable statement
  - definition 633
- exposed name, correlation-name, FROM clause 100
- expression
  - case 129
  - CAST specification 131
  - concatenation operator 117
  - datetime operands, summary of 123
  - decimal operands 121
  - dereference operation 134
  - floating-point operands, rules for 122
  - format and rules 117
  - grouping-expression, use in GROUP BY 329
  - in CAST specification 131
  - in DIGITS function 219
  - in ORDER BY clause 359
  - in SELECT clause, syntax diagram 317
  - in subselect 317
  - integer operands 121
  - operators, mathematical, listing 117
  - precedence of operation 128
  - sign of, values 117
  - string 117
  - substitution operators, listing 117
  - with arithmetic operators 120
  - without operators 117
- extended character set 45
- extended storage 375, 450
- external function
  - description 111

## F

- FETCH statement 637, 639
  - cursor prerequisites for executing 637
- file reference variables
  - BLOB 108
  - CLOB 108
  - DBCLOB 108
- FLOAT data type 59, 530
- FLOAT function 160
- FLOAT function, double precision conversion 233
- floating point numbers
  - as data type 54
  - precision 59
  - range 59
- floating-point constants 89
- floating-point to decimal conversion 72
- FLOOR function 160
  - detailed format description 234
  - values and arguments, rules for 234
- FOR BIT DATA clause
  - CREATE TABLE statement 531
- FOR FETCH ONLY clause
  - select-statement 362
- FOR READ ONLY clause
  - select-statement 362
- foreign key 12, 14
  - adding or dropping, ALTER TABLE 380
  - constraint name, conventions for 542
  - view, referential constraints in 11
- FOREIGN KEY clause
  - CASCADE clause, propagation summary 544
  - constraint name, conventions for 542
  - CREATE TABLE statement 542
  - delete rule, conventions for 544
  - multiple paths, consequences of using 544
  - RESTRICT clause, prohibition 544
  - SET NULL clause, operation of 544
- fragments in SUBSTR function, warning 288
- FREE LOCATOR statement 640
- FROM clause
  - correlation name, use of, example 100
  - exposed and non-exposed names, explanation 100
  - PREPARE statement 673
  - subselect syntax 321
- FROM clause in DELETE statement 600
- FROM clause, use in correlation-name, example 99
- fullselect
  - examples of 352
  - multiple operations, order of execution 351
  - ORDER BY clause 358
  - subquery role, search condition, overview 102
  - table-reference 322
  - used in CREATE VIEW statement 586
- fullselect, detailed syntax 350
- function 110, 155, 174, 185
  - arguments 155
  - built-in 110
  - column 161, 170
  - AVG 156

function (*continued*)

column (*continued*)

- AVG, options and results 171
- COUNT 157, 173
- COUNT\_BIG 157, 174
- COUNT\_BIG, values returned 174
- COUNT, values returned 173
- MAX 178
- MAX, values returned 178
- MIN 162, 180
- STDDEV 164, 182
- STDDEV, options and results 182
- SUM 165, 183
- VAR, options and results 184
- VARIANCE or VAR 167, 184
- VARIANCE, options and results 184
- comment descriptions, adding to catalog 418
- description 110, 155
- expression 155
- external
  - description 111
- HOUR 160
- INTEGER or INT 160
- LCASE 161
- LN 161
- LOG 161
- MICROSECOND 162
- MIDNIGHT\_SECONDS 162
- MONTH 162
- MONTHNAME 162
- name description 49
- nesting 185
- resolution 112
- scalar 161, 162
  - ABS or ABSVAL 156, 186
  - ACOS 156, 187
  - ASCII 156, 188
  - ASIN 156, 189
  - ATAN 156, 190
  - ATAN2 156, 191
  - AVG 171
  - BIGINT 156, 192
  - BIGINT, returning integer values 192
  - BLOB 156, 193
  - CEIL or CEILING 156, 194
  - CHAR 157, 195
  - CHAR, use in datetime conversion 195
  - CHR 157, 200
  - CLOB 157, 201
  - COALESCE 157, 202
  - CONCAT 203

function (*continued*)

scalar (*continued*)

- CONCAT or || 157
- COS 157, 204
- COT 157, 205
- DATE 157, 206
- DATE, returning dates from values 206
- DAY 158, 207
- DAY, returning day part of value 207
- DAYNAME 158, 208
- DAYOFMONTH 158
- DAYOFWEEK 158, 209
- DAYOFYEAR 158, 210
- DAYS 158, 211
- DAYS, returning integer durations 211
- DBCLOB 158, 212
- DECIMAL or DEC 158, 213
- DECIMAL, returning decimal equivalents 213
- definition 185
- DEGREES 159, 216
- DEREF 159, 217
- DIFFERENCE 159, 218
- DIGITS 159, 219
- DLCOMMENT 159, 220
- DLCOMMENT, extracting comment from DATALINK value 220
- DLINKTYPE 159, 221
- DLINKTYPE, extracting linktype from DATALINK value 221
- DLURLCOMPLETE 159, 222
- DLURLCOMPLETE, extracting complete URL from DATALINK value 222
- DLURLPATH 159, 223
- DLURLPATH, extracting path and file name from DATALINK value 223
- DLURLPATHONLY 159, 224
- DLURLPATHONLY, extracting path and file name from DATALINK value 224
- DLURLSCHEME 159, 225
- DLURLSCHEME, extracting scheme from DATALINK value 225
- DLURLSERVER 159, 226
- DLURLSERVER, extracting file server from DATALINK value 226
- DLVALUE 159, 227
- DLVALUE, building a DATALINK value 227
- DOUBLE 159
- DOUBLE or DOUBLE\_PRECISION 159, 229
- DOUBLE, returning floating point values 229
- EVENT\_MON\_STATE 159, 231
- EVENT\_MON\_STATE, returning event monitor states 231

function (*continued*)

scalar (*continued*)

EXP 159, 232  
FLOAT 160, 233  
FLOAT, returning floating point values 233  
FLOOR 160, 234  
GENERATE\_UNIQUE 160, 235  
GRAPHIC 160, 237  
GROUPING 160, 176  
HEX 160, 238  
HOUR 240  
HOUR, returning hour part of values 240  
INSERT 241  
INTEGER or INT 242  
INTEGER, returning integer values 242  
JULIAN\_DAY 243  
LCASE 244  
LEFT 245  
LENGTH 161, 246  
LENGTH, length values from expressions 246  
LN 247  
LOCATE 161, 248  
LOG 249  
LOG10 250  
LONG\_VARCHAR 161, 251  
LONG\_VARGRAPHIC 161, 252  
LTRIM 161, 253  
MICROSECOND 254  
MICROSECOND, returning microsecond part of values 254  
MIDNIGHT\_SECONDS 255  
MINUTE 256  
MINUTE, returning minute part of values 256  
MOD 257  
MONTH 258  
MONTH, returning month part of values 258  
MONTHNAME 259  
NODENUMBER 162, 260  
NULLIF 162, 262  
PARTITION 162, 263  
POSSTR 163, 265  
POWER 163, 267  
QUARTER 163, 268  
RADIANS 163, 269  
RAISE\_ERROR 163, 270  
RAND 163, 272  
REAL 163, 273  
REAL, returning floating point values 273  
REPEAT 163, 274  
REPLACE 163, 275  
restrictions, overview of 185

function (*continued*)

scalar (*continued*)

RIGHT 163, 276  
ROUND 164, 277  
RTRIM 164, 278  
SECOND 164, 279  
SECOND, returning second from values 279  
SIGN 164, 280  
SIN 164, 281  
SMALLINT 164, 282  
SMALLINT, returning small integer values 282  
SOUNDEX 164, 283  
SPACE 164, 284  
SQRT 164, 285  
SUBSTR 165, 286  
SUBSTR, returning substring from string 286  
TABLE\_NAME 165, 289  
TABLE\_SCHEMA 165, 291  
TAN 165, 293  
TIME 165, 294  
TIME, using time in an expression 294  
TIMESTAMP 165, 295  
TIMESTAMP\_ISO 165, 297  
TIMESTAMP, returning timestamp from values 295  
TIMESTAMPDIFF 166, 298  
TRANSLATE 166, 299  
TRUNC or TRUNCATE 166, 301  
TYPE\_ID 166, 302  
TYPE\_NAME 166, 303  
TYPE\_SCHEMA 166, 304  
UCASE 166, 305  
user-defined 312  
VALUE 166, 306  
VALUE, returning non-null result 306  
VARCHAR 167, 307  
VARGRAPHIC 167, 308  
WEEK 167, 310  
YEAR 167, 311  
YEAR, returning values based on year 311  
signature 111  
sourced  
description 111  
user-defined 110  
FUNCTION clause  
COMMENT ON statement 420  
function path 64  
See also SQL path

## G

- GENERATE\_UNIQUE function 160, 235
  - detailed format description 235
- GO TO clause
  - WHENEVER statement 750
- grand total row 333
- GRANT
  - CONTROL ON INDEX 644
  - CREATE ON SCHEMA 649
  - Database Authorities 641
  - Package Privileges 646
  - Table Privileges 652, 658
  - View Privileges 652, 658
- GRANT (Schema Privileges) statement 649, 651
- grant statement
  - authorization name, use in 51
- GRAPHIC data type
  - for CREATE TABLE 532
- GRAPHIC function 160
  - detailed format description 237
  - values and arguments, rules for 237
- graphic string
  - returning from host variable name 299
  - translating string syntax 299
- graphic string, as data type
  - fixed length 54
  - varying length 54
- graphic strings
  - fixed length, description 58
  - varying length, description 58
- GROUP BY clause
  - of subselect, rules and syntax 329
  - results with subselect 319
- group-by-clause, rules and syntax 329
- GROUPING function 160, 176
- grouping sets
  - examples of 342
- grouping-expression 329
- grouping-sets 330

## H

- hash partitioning 41
- hashing on partition keys 548
- HAVING clause
  - of subselect, use of search conditions 336
  - results with subselect 319
- held connection state 32
- HEX

- HEX (*continued*)
  - function 238
    - hexadecimal 238
- HEX function 160
- host identifier
  - definition 48
  - in a host variable 49
  - SQL statement 47
- host variable
  - active set, linking with cursor 668
  - assigning values from a row 705, 748
  - BLOB 107
  - CLOB 107
  - DBCLOB 107
  - declaration rules, related to cursor 596
  - description 105
  - description of 49
  - embedded SQL statements, end declaration 624
  - embedded statements, usage in 370
  - embedded use, BEGIN DECLARE SECTION, rules 407
  - EXECUTE IMMEDIATE statement 631
  - FETCH statement, identifying 637
  - host identifier in 49
  - indicator variable, uses of 106
  - inserting in rows, INSERT statement 662
  - PREPARE statement 674
  - REXX applications, special case 407
  - statement string, restricted listing, PREPARE statement 674
  - substitution for parameter markers 626
  - syntax, diagram of 105
- host-identifier
  - in host variable 106
- host-label 750
- HOUR function 160
- HOUR function, returning hour part of values 240

## I

- identifiers
  - limits 753
- identifiers in SQL
  - description 47
  - host identifiers, syntax for 47
  - long identifiers, definition 47
  - ordinary 47
  - short identifiers, definition 47
- IMMEDIATE
  - EXECUTE IMMEDIATE statement 631, 632

- implicit connect
  - CONNECT statement 432
- implicit decimal number 59
- implicit schema
  - GRANT (Database Authorities) statement 642
  - REVOKE (Database Authorities) statement 688
- IMPLICIT\_SCHEMA authority 9
- IN EXCLUSIVE MODE clause, LOCK TABLE statement 666
- IN predicate, detailed format description 143
- IN SHARE MODE clause, LOCK TABLE statement 666
- IN\_TRAY sample table 870
- INCLUDE clause
  - CREATE INDEX statement 505
- INCLUDE statement 659
- incompatibilities
  - description 891
- incrementing a date, rules for 125
- incrementing a time, rules for 127
- index
  - authorization ID, use in name 51
  - comment descriptions, adding to catalog 418
  - control, granting 654
  - control(to drop), granting, SQL statement for 644
  - correspondence to inserted row values, rules for 663
  - definition of 11
  - deleting, using DROP statement 611
  - primary key, use in matching 389
  - unique key, use in matching 389
  - uses of 11
  - view, relationship to 11
- INDEX clause
  - COMMENT ON statement 421
  - CREATE INDEX statement 504, 505
  - DROP statement 614
  - GRANT statement (table or view) 654
  - REVOKE statement, removing privileges for 698
- index name
  - primary key constraint 542
  - unique constraint 541
- index-name, qualified and unqualified naming 49
- indicator
  - variable 106, 631
- indicator variable
  - host variable, uses in declaring 106
- infix operators 120
- inoperative trigger
  - CREATE TRIGGER statement 574
- inoperative view
  - CREATE VIEW statement 590
- INSERT clause
  - GRANT statement (table or view) 654
  - REVOKE statement, removing privileges for 698
  - values, restrictions leading to failure 663
- INSERT function 160
  - detailed format description 241
  - values and arguments, rules for 241
- insert rule with referential constraint 15
- INSERT statement 661, 665
- insertable
  - view 590
- installing the sample database 863
- integer
  - in ORDER BY clause 358
- integer constants
  - definition of 89
  - syntax example 89
- INTEGER data type 530
  - description 59
  - precision 59
  - range 59
- INTEGER function, integer values from expressions 242
- INTEGER or INT function 160
- integer to decimal conversion, summary 72
- interactive entry of SQL statements 372
- interactive SQL 9
  - CLOSE, use in, example 9
  - DECLARE CURSOR, use in, example 9
  - DESCRIBE, use in, example 9
  - FETCH, use in, example 9
  - OPEN, use in, example 9
  - PREPARE, use in, example 9
  - SELECT statement, dynamic example 9
- interactive SQL, definition of 7
- intermediate result table 321, 328, 329, 336
- International Standards Organization (ISO) date format 61
- International Standards Organization (ISO) time format 61
- INTERSECT clause
  - duplicate rows, use of ALL, effect of 351
  - of fullselect, role in comparison 351
- INTO clause
  - DESCRIBE statement, SQLDA area name 604
  - FETCH statement, host variable substitution 637
  - FETCH statement, use in host variable 105
  - INSERT statement, naming table or view 662



INTO clause (*continued*)  
     PREPARE statement 673  
     restrictions on using, list of 662  
     SELECT INTO statement 705  
     SELECT INTO statement, use in host variable 105  
     values from applications programs 105  
     VALUES INTO statement 748  
 invoking SQL statements 369  
 IS clause  
     COMMENT ON statement 424  
 ISO 61  
     *See also* datetime format  
 ISO/ANSI standards  
     SQLCODE, use of SQL 373  
     SQLSTATE, use of SQL92 373  
 isolation level  
     comparison 885  
     cursor stability 23, 885  
     description 22  
     none 885  
     read stability 23, 885  
     repeatable read 22, 885  
     uncommitted read 24, 885

## J

Japanese Industrial Standard (JIS) date format 61  
 Japanese Industrial Standard (JIS) time format 61  
 JIS 61  
     *See also* datetime format  
 join  
     examples of 339  
     examples of a subselect 337  
     full outer 326  
     inner 326  
     left outer 326  
     partitioning key considerations 551  
     right outer 326  
     table collocation 42  
 joined-table 326  
     table-reference 322  
 JULIAN\_DAY function 160  
     detailed format description 243  
     values and arguments, rules for 243

## K

key  
     composite 12  
     foreign 12, 14

key (*continued*)  
     parent 14  
     partitioning 12  
     primary 12  
     unique 12, 13

## L

labeled duration, detailed description 123  
 labelled durations, in expressions, diagram  
     labelled duration values, listing 123  
 large integers 59  
 large object location, definition 55  
 LCASE function 161  
     detailed format description 244  
     values and arguments, rules for 244  
 LEFT function 161  
     detailed format description 245  
     values and arguments, rules for 245  
 length attributes of columns 56  
 LENGTH function 161  
 LENGTH function, length values from expressions 246  
 lengths of expressions, rules for 246  
 letters, range of 46  
 LIKE predicate, rules for 146  
 limits  
     database manager 755, 757  
     datetime 754  
     identifier 753  
     numeric 753  
     SQL 753  
     string 754  
 literals, overview of 88  
 LN function 161  
     detailed format description 247  
     values and arguments, rules for 247  
 LOB  
     locator, definition 55  
     string, definition 55  
 LOCAL 61  
     *See also* datetime format  
 LOCAL datetime format 61  
 LOCAL time format 61  
 LOCATE function 161  
     detailed format description 248  
     values and arguments, rules for 248  
 locator  
     definition 55  
     FREE LOCATOR statement 640

- locator variable
  - description 108
- LOCK TABLE statement 666, 667
- locking
  - COMMIT statement, effect on 426
  - definition of 20
  - LOCK TABLE statement 666
  - table rows and columns, restricting access 666
- locks
  - during UPDATE 744
  - exclusive 22
  - INSERT statement, default rules for 664
  - share 22
  - terminating for unit of work, ROLLBACK 702
  - update 22
- LOG function 161
  - detailed format description 249
  - values and arguments, rules for 249
- LOG10 function 161
  - detailed format description 250
  - values and arguments, rules for 250
- logging
  - creating table without initial logging 549
- logical operators, rules for search conditions 153
- long identifier, SQL statement, definition 47
- LONG VARCHAR data type
  - for CREATE TABLE 531
- LONG VARCHAR strings
  - attributes, summary 56
  - restrictions on usage 56
- LONG VARGRAPHIC strings
  - attributes, summary 58
  - restrictions on usage 58
- LONG\_VARCHAR function 161
  - detailed format description 251
  - values and arguments, rules for 251
- LONG\_VARGRAPHIC function 161
  - detailed format description 252
  - values and arguments, rules for 252
- LTRIM function
  - detailed format description 253
  - values and arguments, rules for 253

## M

- MANAGED BY clause
  - CREATE TABLESPACE statement 559
- MAX function 161
  - detailed format description 178
  - values and arguments, rules for 178

- MBCS (double-byte character set) data
  - within mixed data 57
- MICROSECOND function 162
- MICROSECOND function, returning microsecond from value 254
- MIDNIGHT\_SECONDS function 162
  - detailed format description 255
  - values and arguments, rules for 255
- MIN function 162
  - detailed format description 180
  - values and arguments, rules for 180
- MINUTE function 162
- MINUTE function, returning minute from value 256
- mixed data
  - See also* SBCS (single-byte character set) data
  - description 57
  - LIKE predicate 148
- MOD function
  - detailed format description 257
  - values and arguments, rules for 257
- MODE keyword, LOCK TABLE statement 666
- MONTH function 162
- MONTH function, returning month from value 258
- MONTHNAME function 162
  - detailed format description 259
  - values and arguments, rules for 259
- multi-byte character set (MBCS), support for 46
- multiple row VALUES clause
  - result data type 82

## N

- name
  - identifying columns in subselect 318
- name, use of in deleting a row 601
- names for columns, rules governing 48
- names in SQL, rules for, summary 48
- names, qualified column, rules for 98
- naming conventions in SQL 48
- nested table expression 323
- new unit of work, initiating 702
- NO ACTION delete rule 542
- node number of row, obtaining 260
- nodegroup
  - adding a node 377
  - adding a partition 377
  - creation 508
  - description 40
  - dropping a node 377
  - dropping a partition 377

- nodegroup (*continued*)
  - naming conventions 49
  - partitioning map created with 509
- NODEGROUP clause
  - COMMENT ON statement 421
  - CREATE BUFFERPOOL statement 450
  - DROP statement 614
- nodegroup name, syntax for 49
- nodegroups
  - comment descriptions, adding to catalog 418
- NODENUMBER function 162, 260
- non-exposed name, re. correlation-name, FROM clause 100
- nonexecutable statement
  - precompiler requirements summary 371
- nonexecutable statement, methods overview 369
- nonrepeatable read 885
- NOT FOUND clause
  - WHENEVER statement 750
- NOT NULL clause
  - CREATE TABLE statement 534
- NOT NULL, use in NULL predicate 151
- NULL
  - in CAST specification 131
  - keyword SET NULL delete rule
    - description 15
- NULL predicate, rules for 151
- null value in SQL
  - assignment, rules governing 71
  - column names in a result 319
  - in duplicate rows 317
  - in grouping-expressions, allowable uses 329
  - in result columns 319
  - specified by indicator variable 106
  - unknown condition 153
- null value in SQL, definition of 54
- NULLIF
  - function description 262
- NULLIF function 162
- numbers, summary of types 59
- numeric
  - assignments in SQL operations 72
  - comparisons, rules for 78
  - limits 753
- numeric data
  - data types, overview 59
- numeric data, remote conversions of 34

## O

- object identifier (OID) 546
  - CREATE TABLE statement 546
  - CREATE VIEW statement 584
- object table 101
- OF clause
  - CREATE VIEW statement 584
- OID column 546
  - See also* object identifier (OID)
- ON clause
  - CREATE INDEX statement 505
- ON TABLE clause
  - GRANT statement 655
  - REVOKE statement 698
- ON UPDATE clause 544
- on-nodes-clause
  - CREATE TABLESPACE statement 562, 563
- ONLY clause in DELETE statement 600
- ONLY clause in UPDATE statement 742
- OPEN statement 668, 672
- operand
  - string 117
- operands
  - datetime
    - date duration 123
    - labelled duration 123
    - time duration 123
  - decimal 121
  - decimal, rules governing 121
  - floating-point, rules for 122
  - integer 121
  - integer, rules governing 121
- operands of in list
  - result data type 82
- operation
  - assignment 70, 75
  - assignments, general description 70
  - comparison 78, 82
  - comparisons, general description 70
  - datetime, SQL rules for 124
  - dereference 134
- operators
  - arithmetic, summary of results 120
- OPTION clause
  - CREATE VIEW statement 587
- OR truth table 153
- ORDER BY clause
  - of select-statement 358
- order of evaluation, expressions 128

- order-by-clause 358
- ordinary identifier, SQL statement 47
- ordinary tokens, definition of 46
- ORG sample table 870
- outer join
  - joined-table 322, 326

## P

- package
  - access plan, relation to term 19
  - authority to create, granting 641
  - authorization ID, use in name 51
  - binding, overview of relationship 25
  - comment descriptions, adding to catalog 418
  - COMMIT statement, effect on cursor 426
  - definition of 19
  - deleting, using DROP statement 611
  - DROP FOREIGN KEY, effect on dependencies 395
  - DROP PRIMARY KEY, effect on dependencies 395
  - DROP UNIQUE key, effect on dependencies 395
  - naming conventions 49
  - necessary privileges, granting 646
  - plan, relation to term 19
  - revoking all privileges 692
  - validity and usage rules when revoking
    - privilege 699
- PACKAGE clause
  - COMMENT ON statement 421
  - DROP statement 615
- package name, syntax for 49
- packed decimal number, locating decimal point 59
- parameter marker
  - host variables in dynamic SQL 105
  - in CAST specification 131
  - in EXECUTE statement 626
  - in OPEN statement 669
  - in PREPARE statement 674
  - rules, syntax and operations 674
  - substitution for, OPEN statement 668
  - typed 675
  - untyped 675
  - usage in expressions, predicates and functions 675
- parent key 14
- parent row 14
- parent table 14
- parentheses
  - precedence of operation, use 128
- partial declustering 41
- partition compatibility
  - definition 87
- PARTITION function 162, 263
- partition number of row, obtaining 260
- partitioned relational database, definition 7
- partitioning data
  - compatibility table 88
  - description 41
  - hash partitioning 41
  - partial declustering 41
  - partition compatibility 87
  - partitioning map, definition 41
- partitioning key 12
  - adding or dropping, ALTER TABLE 380
  - ALTER TABLE statement 390
  - considerations 551
  - defining when creating table 548
  - purpose 41
- partitioning map
  - created with nodegroup 509
- partitioning map index of row, obtaining 263
- partitioning of data 7
- PCTFREE clause
  - CREATE INDEX statement 506
- performance
  - partitioning key recommendation 551
- phantom row 23, 885
- positional updating of columns by row 742
- POSSTR function 163
- POSSTR scalar function
  - description 265
- POWER function 163
  - detailed format description 267
  - values and arguments, rules for 267
- precedence
  - level operators, rules for 128
  - operation, order of evaluating 128
- precision of numbers
  - determined by SQLLEN variable 769
- precision-integer, DECIMAL function
  - default values for data types 213
- precision, as a numeric attribute 59
- precompiler
  - INCLUDE statement, trigger for 659
  - non-executable statements, usage overview 371
  - static SQL, use in Run-Time Service calls 8
- precompiling
  - including external text file 659
  - initiating and setting up SQLDA and SQLCA 659

- predicate
  - basic, detailed format, diagram 136
  - BETWEEN, detailed format diagram 140
  - description 135
  - EXISTS, detailed format description 142
  - IN, detailed format description 143
  - LIKE 146
  - NULL, detailed format, diagram 151
  - quantified, usage and rules 137
  - TYPE, detailed format, diagram 152
- prefix operator 120
- PREPARE statement 673, 681
  - embedded usage, detailed description 371
  - use in dynamic SQL 7
- prepared SQL statement 763
  - dynamically declaring, PREPARE statement 673
  - dynamically prepared by PREPARE 681
  - executing 626, 630
  - host variables, substitution of 626
  - information, obtaining with DESCRIBE 604
- prepared statement
  - OPEN statement, use in variable substitution 668
  - SQLDA provides information about 763
- primary key 12
  - adding or dropping, ALTER TABLE 380
  - adding, privileges for, granting 653
  - CREATE TABLE statement 540
  - dropping, privileges for, granting 653
- PRIMARY KEY clause
  - ALTER TABLE statement 389
  - CREATE TABLE statement 541
- privilege 697
- privileges
  - CONTROL privilege, overview of 38
  - database, effects of revoking 689, 696
  - DBADM, scope of 39
  - definition 38
  - index, effects of revoking 691
  - overview 38
  - package, effects of revoking 693
  - packages, validity rules when revoking 699
  - SYSADM, scope of 38
  - SYSCTRL, scope of 39
  - SYSMAINT, scope of 39
  - table or view, effects of revoking 700
  - views, cascading effects of revoking 699
- procedure
  - naming conventions 49
- PROCEDURE clause
  - COMMENT ON statement 422

- procedure name, syntax for 49
- PROJECT sample table 870
- promotion
  - of data types 66
- PUBLIC clause
  - GRANT statement 642, 645, 647, 650, 656
  - REVOKE statement 689, 690, 693, 696
  - REVOKE statement, removing privileges for 699

## Q

- qualified column names, rules for 98
- qualifier
  - reserved 881
- quantified predicate, detailed rules for 137
- QUARTER function 163
  - detailed format description 268
  - values and arguments, rules for 268
- query 315, 366
  - authorization IDs required for issuing 315
  - definition 315
  - recursive 357
    - example 961
- query (SQL)
  - subquery, use in WHERE clause 328
- question mark (?) 626
  - See also parameter marker

## R

- RADIANS function 163
  - detailed format description 269
  - values and arguments, rules for 269
- RAISE\_ERROR function 163, 270
- raising errors
  - RAISE\_ERROR function 270
  - SIGNAL SQLSTATE statement 738
- RAND function 163
  - detailed format description 272
  - values and arguments, rules for 272
- read stability 23, 885
- read-only
  - view 590
- read-only cursor
  - ambiguous 598
- REAL data type 530
  - precision 59
  - range 59
- REAL function 163

- REAL function, single precision conversion 273
- Record Blocking
  - locks to row data, INSERT statement 664
- recovery of applications 19
- recursion
  - example 961
  - query 357
- recursive common table expression 357
- reference type
  - comparison 82
  - DEREF function 217
  - description 65
- reference types
  - casting 68
- REFERENCES clause
  - GRANT statement 654
  - REVOKE statement, removing privileges for 698
- referential constraint 14
- referential cycle 14
- referential integrity 14
- REFRESH TABLE statement 682
  - REFRESH DEFERRED 682
  - REFRESH IMMEDIATE 682
- register 91
  - See also* special register
- relational database, definition 7
- RELEASE statement 684
- release to release incompatibilities
  - description 891
- release-pending connection state 32
- remote access
  - application server, role in 25
  - character strings, conversions 34
  - CONNECT statement
    - EXCLUSIVE MODE, dedicated connection 437
    - ON SINGLE NODE, dedicated connection 437
    - server information only, no operand 437
    - SHARE MODE, read-only for non-connector 437
  - IMPLICIT connect, diagram of state transitions 27
  - non-IMPLICIT connect, diagram of state transitions 29
  - numeric data, conversions 34
  - successful connection, detailed description 434
  - unsuccessful connection, detailed description 436
- remote execution of SQL 30
- remote unit of work, overview 26
- RENAME TABLE statement 685, 686
- REPEAT function 163
  - detailed format description 274
  - values and arguments, rules for 274
- repeatable read 22, 885
- REPLACE function 163
  - detailed format description 275
  - values and arguments, rules for 275
- reserved
  - qualifiers 881
  - schema names 881
  - words 881
- RESTRICT delete rule 542
  - description 15
- result columns of subselect 320
- result data type
  - arguments of COALESCE 82
  - multiple row VALUES clause 82
  - operands of in list 82
  - result expressions of CASE expression 82
  - set operator 82
- result expressions of CASE expression
- result data type 82
- result table 10
- result table, result from query 315
- return code
  - embedded statements, language instructions for 373
  - executable statements, usage summary 370
- REVOKE
  - CONTROL ON INDEX 690
  - CREATEIN ON SCHEMA 695
  - Database Authorities 687
  - DROPIN ON SCHEMA 695
  - Package Privileges 692
  - Table Privileges 697, 701
  - View Privileges 697, 701
- REVOKE (Schema Privileges) statement 695, 696
- revoke statement
  - authorization-name, use in 51
- REXX
  - END DECLARE SECTION, prohibition 624
- RIGHT function 163
  - detailed format description 276
  - values and arguments, rules for 276
- ROLLBACK
  - cursor, effect on 702
  - SQL statement, detailed usage instructions for 702
- rollback description 20
- ROLLBACK statement 703
  - detailed syntax instructions 702
- ROLLUP 331
  - examples of 342

- ROUND function 164
  - detailed format description 277
  - values and arguments, rules for 277
- row
  - as syntax component, diagram 317
  - assigning values to host variable, SELECT INTO 705
  - assigning values to host variable, VALUES INTO 748
  - COUNT function, values returned 173
  - COUNT\_BIG function, values returned 174
  - cursor, effect of closing on FETCH 416
  - cursor, FETCH statement, relation to 670
  - cursor, location in result table 596
  - definition of 10
  - deleting, privilege for, granting 654
  - deleting, SQL statement, details 599
  - dependent 14
  - descendent 14
  - exporting row data, privilege for, granting 655
  - FETCH request, cursor row selection 596
  - GROUP BY clause, result table from 329
  - GROUP BY, use in limiting in SELECT clause 318
  - HAVING clause, results from search, rules for 336
  - HAVING, use in limiting in SELECT clause 318
  - importing values, privilege for, granting 654
  - index, role of key 504
  - inserting into table or view 661
  - inserting values, INSERT statement 662
  - inserting, privilege for, granting 654
  - locks to row data, INSERT statement 664
  - locks, effect on cursor of WITH HOLD 596
  - parent 14
  - retrieving row data, privilege for, granting 655
  - search conditions, detailed syntax 153
  - SELECT clause, select list notation 317
  - self-referencing 14
  - UNIQUE clause, table index, effect on key 504
  - updating column values, UPDATE statement 740
  - values, insertion, restrictions leading to failure 663
- row fullselect
  - UPDATE statement 743
- RR (repeatable read) isolation level 22, 885
- RS (read stability) isolation level 23, 885
- RTRIM function 164
  - detailed format description 278
  - values and arguments, rules for 278
- run-time authorization ID 52
- Run-Time Services, static SQL support for 8

## S

- SALES sample table 871
- sample database
  - erasing 864
  - installing 863
- sample tables 863, 880
- SBCS (single-byte character set) data
  - within mixed data 57
- SBCS (single-byte character set) data, description 57
- scalar
  - INSERT 160
  - JULIAN\_DAY 160
  - LEFT 161
  - MINUTE 162
- scalar fullselect 117
- scalar function 185
  - See also* function
- scalar function, arguments for 155
- scale of data 763
  - comparisons in SQL, overview 78
  - conversion of numbers in SQL 72
  - determined by SQLLEN variable 765
  - in results of arithmetic operations 121
- scale of numbers
  - determined by SQLLEN variable 769
- scale-integer, DECIMAL function 213
- schema
  - controlling use of 9
  - CREATE SCHEMA statement 519
  - creating implicit schema, granting authority for 642
  - creating implicit schema, revoking authority for 688
  - definition of 9
  - privileges 10
- SCHEMA clause
  - COMMENT ON statement 423
  - DROP statement 616
- schema-name
  - description 49
  - reserved names 881
- schema-name, description of 50
- schemas
  - comment descriptions, adding to catalog 418
  - definition of 9
- scope
  - adding with ALTER TABLE statement 391
  - adding with ALTER VIEW statement 405
  - defining in CAST specification 132
  - defining with added column 387
  - defining with CREATE TABLE statement 539

- scope (*continued*)
  - defining with CREATE VIEW statement 585
  - definition of 65
  - dereference operation 134
- SCOPE clause
  - ALTER TABLE statement 387, 391
  - ALTER VIEW statement 405
  - CREATE TABLE statement 539
  - CREATE VIEW statement 585
  - in CAST specification 132
- scoped-ref-expression
  - in dereference operation 134
- search condition
  - AND, logical operator 153
  - description 153
  - HAVING clause, arguments and rules 336
  - NOT, logical operator 153
  - OR, logical operator 153
  - order of evaluation 153
  - using WHERE clause, rules for 328
  - with DELETE, row selection 601
  - with UPDATE, applying changes to a match 743
- SECOND function 164
- SECOND function, returning second from value 279
- security
  - CONNECT statement 437
- SELECT clause
  - DISTINCT keyword, use in 317
  - GRANT statement (table or view) 655
  - list notation, column reference 317
  - REVOKE statement, removing privileges for 698
- SELECT INTO statement 705, 706
- select list
  - application of, rules and syntax 319
  - description 317
  - notation rules and conventions 317
- SELECT statement
  - cursor, rules regarding parameter markers 596
  - dynamic invocation, execution overview 371
  - fullselect, detailed syntax 350
  - interactive invocation, limitations on 372
  - invoking, usage summary 370
  - result table, OPEN statement, relation to cursor 668
  - select-statement 355
  - static invocation, execution overview 371
  - subselect 316
  - VALUES clause 350
- select-statement
  - examples of 365
- self-referencing row 14
- self-referencing table 14
- sequence values
  - generating 235
- servername, description of 50
- SET clause
  - UPDATE statement, column names and values 742
- SET CONNECTION statement 707, 708
  - successful connection, detailed description 707
  - unsuccessful connection, detailed description 708
- SET CONSTRAINTS statement 709, 715
- SET CURRENT DEGREE statement 716, 717
- SET CURRENT EXPLAIN MODE statement 718, 719
- SET CURRENT EXPLAIN SNAPSHOT statement 720, 721
- SET CURRENT FUNCTION PATH statement 731
- SET CURRENT PATH statement 731
- SET CURRENT QUERY OPTIMIZATION statement 724, 726
- SET CURRENT SQLID statement 733
- SET DEFAULT delete rule
  - description 15
- SET EVENT MONITOR STATE statement 729, 730
- SET NULL delete rule 542
  - description 15
- set operator
  - EXCEPT, comparing differences only 351
  - INTERSECT, role of AND in comparisons 351
  - result data type 82
  - UNION, correspondence to OR 351
- SET PATH statement 731
- SET SCHEMA statement 733
- SET transition-variable statement 735, 737
- setting up document server 987
- SHARE
  - IN SHARE MODE 432
- share locks 22
- SHARE option, LOCK TABLE statement 666
- shift-in character
  - not truncated by assignments 74
- short identifier, SQL statement, definition 47
- SIGN function 164
  - detailed format description 280
  - values and arguments, rules for 280
- sign, as a numeric attribute 59
- SIGNAL SQLSTATE statement 738, 739
- SIN function 164
  - detailed format description 281
  - values and arguments, rules for 281



- single precision float data type 530
- single row select 705
- single-byte character set (SBCS) 57
  - See also SBCS (single-byte character set) data
- single-byte character set (SBCS), support for 46
- single-precision floating-point 59
- small integer
  - description 59
  - precision 59
  - range 59
- SMALLINT data type 530
  - description 59
  - precision 59
  - range 59
- SMALLINT function 164
- SMALLINT function, small integer values from
  - expressions 282
- SMS table space
  - CREATE TABLESPACE statement 561
  - description 39
- SOME in quantified predicate 137
- sorting
  - ordering of results 79
  - string comparisons 78
- SOUNDEX function 164
  - detailed format description 283
  - values and arguments, rules for 283
- sourced function
  - description 111
- space 46
- SPACE function 164
  - detailed format description 284
  - values and arguments, rules for 284
- spaces
  - rules governing 46
- special characters, range of 46
- special register 91
  - CURRENT DATE 91, 96
  - CURRENT DEGREE 91
  - CURRENT EXPLAIN MODE 92
  - CURRENT EXPLAIN SNAPSHOT 93
  - CURRENT FUNCTION PATH 94
  - CURRENT NODE 94
  - CURRENT PATH 94
  - CURRENT QUERY OPTIMIZATION 95
  - CURRENT REFRESH AGE 95
  - CURRENT SCHEMA 96
  - CURRENT SERVER 96
  - CURRENT SQLID 96
  - CURRENT TIME 96
  - special register (*continued*)
    - CURRENT TIMESTAMP 97
    - CURRENT TIMEZONE 97
    - interaction of Explain special registers 957
    - USER 98
- specific function
  - comment descriptions, adding to catalog 418
- SPECIFIC FUNCTION clause
  - COMMENT ON statement 421
- SPECIFIC PROCEDURE clause
  - COMMENT ON statement 423
- specific-name, description of 50
- specification
  - CAST 131
- SQL (Structured Query Language)
  - numbers 59
  - tokens 46
- SQL comments, static statements, rules for 374
- SQL identifiers
  - database identifier 47
  - long identifier 47
  - short identifier 47
- SQL path 64
  - CURRENT PATH special register 94
  - resolution 112
- SQL return code 373
- SQL statement
  - ALTER BUFFERPOOL 375, 376
  - ALTER NODEGROUP 377
  - ALTER TABLE 380, 398
  - ALTER TABLESPACE 399, 402
  - ALTER TYPE (Structured) 403, 404
  - ALTER VIEW 406
  - BEGIN DECLARE SECTION 407, 408
  - CALL 409, 415
  - CLOSE 416, 417
  - COMMENT ON 418, 425
  - COMMIT 426, 427
  - Compound SQL 428, 431
  - CONNECT (Type 1) 432, 438
  - CONNECT (Type 2) 439, 445
  - CONTINUE, response to exception 750
  - CREATE ALIAS 446, 448
  - CREATE BUFFERPOOL 449, 451
  - CREATE DISTINCT TYPE 452, 457
  - CREATE EVENT MONITOR 458, 466
  - CREATE FUNCTION 467, 483, 496
  - CREATE FUNCTION (External Scalar) 468
  - CREATE FUNCTION (External Table) 484
  - CREATE FUNCTION (Sourced Scalar) 503

SQL statement (*continued*)

- CREATE FUNCTION (Sourced) 497
- CREATE INDEX 504, 507
- CREATE NODEGROUP 508
- CREATE SCHEMA 519, 521
- CREATE TABLE 522, 558
- CREATE TABLESPACE 559, 567
- CREATE TRIGGER 568, 577
- CREATE TYPE (Structured) 578, 581
- CREATE VIEW 582, 594
- DECLARE CURSOR 595, 598
- DELETE 599, 603
- DESCRIBE 604, 607
- DISCONNECT 608, 610
- DROP 611, 623
- dynamic SQL, definition of 7
- END DECLARE SECTION 624, 625
- EXECUTE 626, 630
- EXECUTE IMMEDIATE 631, 632
- EXPLAIN 633, 636
- FETCH 637, 639
- FREE LOCATOR 640
- GRANT (Schema Privileges) 651
- GRANT (Schema Privileges) 649
- GRANT (Table Privileges) 652, 658
- GRANT (View Privileges) 652, 658
- immediate execution of dynamic SQL 7
- INCLUDE 659
- INSERT 661, 665
- interactive SQL, definition of 7
- LOCK TABLE 666, 667
- OPEN 668, 672
- pg=end.ALTER NODEGROUP 379
- pg=end.CREATE NODEGROUP 510
- PREPARE 673, 681
- preparing and executing dynamic SQL 7
- REFRESH TABLE 682
- RELEASE 684
- RENAME TABLE 685, 686
- REVOKE (Schema Privileges) 696
- REVOKE (Schema Privileges) 695
- REVOKE (Table Privileges) 697, 701
- REVOKE (View Privileges) 697, 701
- ROLLBACK 702, 703
- SELECT INTO 705, 706
- SET CONNECTION 707, 708
- SET CONSTRAINTS 709, 715
- SET CURRENT DEGREE 716, 717
- SET CURRENT EXPLAIN MODE 718, 719
- SET CURRENT EXPLAIN SNAPSHOT 720, 721

SQL statement (*continued*)

- SET CURRENT FUNCTION PATH 731
- SET CURRENT PATH 731
- SET CURRENT QUERY OPTIMIZATION 724, 726
- SET EVENT MONITOR STATE 729, 730
- SET PATH 731
- SET SCHEMA 733, 734
- SET transition-variable 735, 737
- SIGNAL SQLSTATE 738, 739
- specific-name, conventions for 50
- statement name, conventions for 50
- static SQL, definition of 7
- syntax conventions for 2
- UPDATE 740, 746
- VALUES INTO 748, 749
- WHENEVER 750, 751
- WITH HOLD, cursor attribute 595

SQL statement syntax

- case sensitive identifiers, rule for 47
- cursor-name, definition of 49
- escape character 47
- specific-name, conventions for 50
- statement name, conventions for 50

SQL syntax

- AVG function, results on column set 171
- basic predicate, detailed diagram 136
- BETWEEN predicate, rules for 140
- comparing two predicates, truth conditions 136, 152
- COUNT function, arguments and results 173
- COUNT\_BIG function, arguments and results 174
- data types, overview 54
- dates, detailed description 60
- DISTINCT keyword, queries, role of 170
- executable statements, embedded usage 371
- EXISTS predicate, detailed format description 142
- GENERATE\_UNIQUE function, arguments and results 235
- GROUP BY clause, use in subselect 329
- IN predicate, detailed format description 143
- LIKE predicate, rules for 146
- multiple operations, order of execution 351
- naming conventions, listing of, definitions 48
- non-executable statements, embedded usage 371
- null value, definition of 54
- scale of data in SQL 59
- search conditions, detailed formats and rules 153
- SELECT clause, detailed description 317
- SELECT statement, invocation methods 370
- STDDEV function, results on column set 182
- times, detailed description 60

SQL syntax (*continued*)

- TYPE predicate, detailed diagram 152
- values, overview 54
- VARIANCE function, results on column set 184
- WHERE clause, search conditions for 328

SQL92

- setting rules for dynamic SQL 733

SQLCA (SQL communication area) 759

- entry changed by UPDATE 744

SQLCA (SQL communication area) clause

- INCLUDE statement 659

SQLCA structure, overview 373

SQLCODE

- description 373
- return code values, table 373

SQLD field in SQLDA 763

- description 764

SQLDA

- host variable descriptions, OPEN statement 669
- prepared statement information, storing 673

SQLDA (SQL descriptor area) 763

- contents 763
- FETCH statement 638

SQLDA (SQL descriptor area) clause

- INCLUDE statement, specifying 659

SQLDA area, required variables for DESCRIBE 604

SQLDABC field in SQLDA 763

- description 764

SQLDAID field in SQLDA

- description 764

SQLDATALEN field in SQLDA

- description 767

SQLDATATYPE\_NAME field in SQLDA

- description 767

SQLERROR clause

- WHENEVER statement 750

SQLIND field in SQLDA 763

- description 765

SQLLEN field in SQLDA 763

- description 765

SQLLONGLEN field in SQLDA

- description 766

SQLN field in SQLDA 763

- description 764

SQLNAME field in SQLDA 763

- description 766

sqlstate

- description 373
- in RAISE\_ERROR function 270
- in SIGNAL SQLSTATE statement 738

sqlstate (*continued*)

- ISO/ANSI SQL92 standard, relation to 373

SQLTYPE field in SQLDA 763

- description 765

SQLVAR field in SQLDA 763

- base 765
- secondary 766

SQLWARNING clause

- WHENEVER statement 750

SQRT function 164

- detailed format description 285
- values and arguments, rules for 285

STAFF sample table 872

STAFFG sample table 874

standards

- setting rules for dynamic SQL 733

starting a new unit of work 702

statement string, PREPARE statement, rules for 674

statement string, rules for creating 631

statement-name, description of 50

states

- connection 32

static select 371

static SQL

- DECLARE CURSOR statement, usage in 371
- definition of 7
- FETCH statement, usage in 371
- invoking 370, 371
- OPEN statement, usage in 371
- source code, differences from dynamic SQL 7

statistics

- updating 823, 830

STDDEV function 164

STDDEV function, detailed description 182

storage

- backing out, unit of work, ROLLBACK 702

storage structures

- ALTER BUFFERPOOL statement 375
- ALTER TABLESPACE statement 399
- buffer pool 40
- CREATE BUFFERPOOL statement 449
- CREATE TABLESPACE statement 559
- description 39
- nodegroup 40
- table space 39

stored procedures

- CALL statement 409

string

- assignment
- conversion rules 73

- string (*continued*)
  - BLOB 55
  - CLOB 55
  - constant
    - character 89
    - hexadecimal 90
  - definition 35
  - expression 117
  - LOB 55
  - operand 117
- string limits 754
- Structured Query Language (SQL)
  - assignment operation, overview 70
  - basic operands, assignments and comparisons 70
  - character strings, overview of 56
  - characters, range of 45
  - comments, rules for 45
  - comparison operation, overview 70
  - constants, definition of 88
  - double byte character set (DBCS),
    - considerations 45
  - identifiers, definition of
    - delimited identifier, description 47
    - ordinary identifiers, description 47
  - spaces, definition of 45
  - tokens, definition of
    - delimiter tokens 45
    - ordinary tokens 45
  - values
    - data types for 54
    - overview 54
    - sources of 54
  - variable names used 48
- structured type
  - description 65
  - DROP statement 611
- sub-total rows 331
- subject table of trigger 17
- subquery
  - HAVING clause 336
  - in HAVING clause, execution of 336
  - in WHERE clause 328
- subquery, fullselect use as, search conditions 102
- subselect 316
  - definition 316
  - examples of 337
  - FROM clause, relation to subselect 316
  - sequence of operations, example 316
- SUBSTR function 165
- SUBSTR function, returning substring from string 286
- substrings
  - cautions and restrictions 288
  - length, defining 286
  - locating in string 286
  - start, setting 286
- SUM function 165
  - detailed format description 183
  - values and arguments, rules for 183
- SUMMARY table
  - in CREATE TABLE statement 527
- summary tables
  - REFRESH TABLE statement 682
- super-aggregate rows 332
- super-groups 331
- supertype
  - supertype name, conventions for 50
- supertype-name, description 50
- symetric super-aggregate rows 332
- synonym
  - CREATE ALIAS statement 446
  - DROP ALIAS statement 612
  - qualifying a column name 99
- syntax diagrams
  - description 2
- system administration privilege 38
- system control privilege 39
- system maintenance privilege 39
- system managed space 39
  - See also* SMS table space
- system-containers
  - CREATE TABLESPACE statement 561

## T

- table 7, 863
  - adding a column, ALTER TABLE 384
  - alias 446, 612
  - authorization for creating 522
  - authorization ID, use in name 51
  - base table 10
  - catalog views for use with structured types 831
  - catalog views on system tables 773
  - changing definition 380
  - collocation 42
  - comment descriptions, adding to catalog 418
  - common table expression 19
  - control privilege, granting 654
  - creating a table, granting authority for 641
  - creating, SQL statement instructions 522

table (*continued*)

- definition of 10
- deleting, using DROP statement 611
- dependent 14
- descendent 14
- designator, use to avoid ambiguity 101
- exception 712, 967
- exposed or non-exposed names, FROM clause 100
- foreign key 12
- FROM clause, subselect, naming conventions 321
- index creation, requirements of 504
- nested table expression, use of 103
- parent 14
- partitioning key 12
- partitioning map 41
- primary key 12
- privileges, granting 652
- qualifying a column name 99
- renaming, requirements of 685
- restricting shared access, LOCK TABLE statement 666
- result table 10
- revoking privileges for 697
- row, inserting 661
- sample 863
- scalar fullselect, use of 103
- schema 519
- self-referencing 14
- space 39, 449, 559, 617
- subquery, use of 102
- table name, conventions for 50
- table-reference 322
- temporary, OPEN statement, use of 670
- unique correlation names as table designators 103
- unique key 12
- updating by row and column, UPDATE statement 740

table check constraint

- description 16

TABLE clause

- COMMENT ON statement 423
- DROP statement 616

table expression

- common-table-expression 356
- description 18

table join

- partitioning key considerations 551

table space

- comment descriptions, adding to catalog 418
- deleting, using DROP statement 611

table space (*continued*)

- description 39
- identification
  - CREATE TABLE statement 547
- index
  - CREATE TABLE statement 547
- name description 50
- pagesize 561

table-name

- in ALTER TABLE statement 384
- in CREATE TABLE statement 527
- in FROM clause 321
- in LOCK TABLE statement 666
- in SELECT clause, syntax diagram 317

table-name, description 50

table-reference

- alias 323
- nested table expressions 323
- table-name 322
- view-name 322

TABLE\_NAME function 165

- alias 289

TABLE\_SCHEMA function 165

- alias 291

tables

- distributed relational database, use in 24

TABLESPACE clause

- COMMENT ON statement 423

tablespace-name, description 50

TAN function 165

- detailed format description 293
- values and arguments, rules for 293

temporary tables in OPEN 670

terminating

- unit of work 426, 702

terminating a unit of work 702

time

- arithmetic operations, rules for 126
- as data type 54
- CHAR, use of in format conversion 195
- duration, format of 124
- expression, using in 294
- hour values, using in an expression (HOUR) 240
- microsecond, returning from datetime value 254
- minute, returning from datetime value 256
- returning values based on time 294
- second, returning from datetime value 279
- strings 61
- timestamp
  - internal representation of 60
  - length of string 60

- time (*continued*)
  - timestamp, as data type 54
  - timestamp, returning from values 295
- time data type 60, 532
- TIME function 165
- TIME function, using time in an expression 294
- timestamp
  - arithmetic operations 127
  - as data type 54
  - data definition 60
  - duration 124
  - from GENERATE\_UNIQUE result 235
  - multi-byte character string (MBCS) restriction 63
  - string representation format 62
  - WEEK scalar function, using 310
- TIMESTAMP data type 532
- TIMESTAMP function 165
- TIMESTAMP function, returning from values 295
- TIMESTAMP\_ISO function 165
  - detailed format description 297
  - values and arguments, rules for 297
- TIMESTAMPDIFF function 166
  - detailed format description 298
  - values and arguments, rules for 298
- TO clause
  - GRANT statement 642, 644, 647, 650, 656
- tokens
  - as language element 45
  - delimiter tokens, definition of 46
  - ordinary tokens, definition of 46
  - spaces, rules governing 46
  - upper and lower case, support for 47
- transform
  - DROP statement 611
- transition tables in triggers 18
- transition variables in triggers 18
- TRANSLATE function 166
  - character string, using with 299
  - graphic string, using with 299
  - rules and restrictions 299
- translation table 299
- trigger
  - and constraints 887
  - CREATE TRIGGER statement 568
  - DROP statement 617
  - errors executing 574
  - Explain tables 935
  - inoperative 574
  - interactions 887
  - name description 50
- TRIGGER clause
  - COMMENT ON statement 423
- triggered SQL statement
  - SET transition-variable statement 735
  - SIGNAL SQLSTATE statement 738
- triggers
  - activation 17
  - activation time 17
  - cascading 18
  - comment descriptions, adding to catalog 418
  - description 17
  - event 17
  - granularity 17
  - INSERT statement 664
  - set of affected rows 17
  - subject table 17
  - triggered action 17
  - uses of 17
- TRUNC or TRUNCATE function 166
  - detailed format description 301
  - values and arguments, rules for 301
- truncation of numbers 72
- truth table 153
- truth valued logic, search conditions, rules for 153
- type
  - type name, conventions for 50
- TYPE clause
  - COMMENT ON statement 423
  - DROP statement 617
- TYPE predicate, detailed format 152
- type-name, description 50
- TYPE\_ID function 166
  - data type 302
- TYPE\_NAME function 166
  - data type 303
- TYPE\_SCHEMA function 166
  - data type 304
- typed-table
  - typed-table name, conventions for 50
- typed-table-name, description 50
- typed-view
  - typed-view name, conventions for 51
  - typed-view-name, description 51

## U

- UCASE function 166
  - detailed format description 305
  - values and arguments, rules for 305

- unary
  - minus sign, results of 120
  - plus sign, results of 120
- uncommitted changes, relation to locks 20
- uncommitted read 24, 885
- unconnected state 32
- undefined reference, error conditions for 102
- UNDER clause
  - CREATE VIEW statement 584
- UNION clause, role in comparison
  - of fullselect 351
- UNIQUE clause
  - ALTER TABLE statement 389
  - CREATE INDEX statement 504
  - CREATE TABLE statement 541
- unique constraint 12, 13
  - adding or dropping, ALTER TABLE 380
  - ALTER TABLE statement 388
  - CREATE TABLE statement 540
- unique correlation names as table designators 103
- unique key 12, 13
  - ALTER TABLE statement 388
  - CREATE TABLE statement 540
- unique values
  - generating 235
- unit of work
  - COMMIT 426
  - description 20
  - destroying prepared statements 681
  - initiating closes cursors 670
  - referring to prepared statements 673
  - ROLLBACK statement, effect of 702
  - terminating 426
  - terminating destroys prepared statements 681
  - terminating without saving changes 702
- unknown condition
  - null value 153
- updatable
  - view 590
- UPDATE clause
  - GRANT statement 655
  - REVOKE statement, removing privileges for 698
- update locks 22
- UPDATE statement 740, 746
  - row fullselect 743
- updating statistics 823, 830
- uppercase, folding to 47
- UR (uncommitted read) isolation level 24, 885
- USA 61
  - See also* datetime format

- USA date format 61
- USA time format 61
- USER special register 98
- user-defined data type
  - distinct-type-name
    - CREATE TABLE statement 533
- user-defined function 312
  - CREATE FUNCTION (External Scalar)
    - statement 468
  - CREATE FUNCTION (External Table)
    - statement 484
  - CREATE FUNCTION (Sourced) statement 497
  - CREATE FUNCTION statement 467
    - description 110
    - DROP statement 611
    - GRANT (Database Authorities) statement 642
    - REVOKE (Database Authorities) statement 688
- user-defined type
  - comment descriptions, adding to catalog 418
  - description 64
- user-defined types
  - casting 67
- USING clause
  - EXECUTE statement 626
  - FETCH statement 638
  - OPEN statement, listing host variables 668
- USING DESCRIPTOR 626
- USING DESCRIPTOR clause
  - EXECUTE statement 626
  - OPEN statement 669

## V

- VALUE function 166, 306
- value in SQL 54
- value, data definition of 10
- VALUES clause
  - fullselect 350
  - INSERT statement, loading one row 662
  - number of values, rules for 662
- VALUES INTO statement 748, 749
- VARCHAR
  - DOUBLE scalar function, using 229
  - function 307
- VARCHAR data type 531
- VARCHAR function 167
- VARCHAR strings
  - attributes, summary 56
  - restrictions on usage 56

- VARCHAR(26)
  - WEEK scalar function, using 310
- VARGRAPHIC
  - function 308
- VARGRAPHIC function 167
- VARGRAPHIC strings
  - attributes, summary 58
  - restrictions on usage 58
- VARIANCE function, detailed description 184
- VARIANCE or VAR function 167
- view
  - alias 446, 612
  - authorization ID, use in name 51
  - comment descriptions, adding to catalog 418
  - control privilege
    - granting 654
    - limitations on 654
  - creating 582
  - deletable 589
  - deleting, using DROP statement 611
  - description 10
  - exposed or non-exposed names, FROM clause 100
  - foreign key, referential constraints 11
  - FROM clause, subselect, naming conventions 321
  - index, relation to view 11
  - inoperative 590
  - insertable 590
  - preventing view definition loss, WITH CHECK OPTION 744
  - privileges, granting 652
  - qualifying a column name 99
  - read-only 590
  - revoking privileges for 697
  - row, inserting in viewed table 661
  - schema 519
  - updatable 590
  - updating rows by columns, UPDATE statement 740
  - validity and usage rules when revoking
    - privilege 699
  - view name, conventions for 51
  - WITH CHECK OPTION, effect on UPDATE 744
- VIEW clause
  - CREATE VIEW statement 582
  - DROP statement 618
- view-name
  - description of 51
  - in ALTER VIEW statement 405
  - in FROM clause 321
  - in SELECT clause, syntax diagram 317

## W

- warning return code 373
- WEEK
  - function 310
- WEEK function 167
- WHENEVER statement 750, 751
- WHENEVER statement, changing flow of control 371
- WHERE clause
  - DELETE statement, row selection 601
  - search function, subselect, rules for 328
  - UPDATE statement, conditional search 743
- WHERE CURRENT OF clause
  - DELETE statement, use of DECLARE CURSOR 601
  - UPDATE statement 743, 744
- wildcard character
  - LIKE predicate, values for 146
- WITH CHECK OPTION clause
  - CREATE VIEW statement 587
- WITH clause
  - CREATE VIEW statement 586
  - INSERT statement 663
- WITH common-table-expression 355
- WITH DEFAULT clause
  - ALTER TABLE statement 384
- WITH GRANT OPTION clause
  - GRANT statement 656
- WITH HOLD clause
  - DECLARE CURSOR statement 595
- WITH OPTIONS clause
  - CREATE VIEW statement 585
- WORK
  - in COMMIT statement 426
  - in ROLLBACK statement 702

## Y

- YEAR function 167
- YEAR function, using in expressions 311







Part Number: 04L9267

Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

S10J-8165-01



04L9267

