



iSeries

DB2 Universal Database for iSeries SQL Reference

Version 5





@server

iSeries

DB2 Universal Database for iSeries SQL Reference

Version 5

Contents

About DB2 UDB for iSeries SQL

Reference xiii

Standards Compliance	xiii
Who should read the SQL Reference book	xiii
Assumptions Relating to Examples of SQL	
Statements	xiv
How to Read Syntax Diagrams	xv
Conventions for Describing Mixed Data Values	xvii
SQL Accessibility	xvii
What's new for V5R2 in the SQL Reference book	xviii

Chapter 1. Concepts 1

Relational Database	1
Structured Query Language	3
Static SQL	3
Dynamic SQL	3
Extended Dynamic SQL	4
Interactive SQL	4
SQL Call Level Interface (CLI) and Open Database	
Connectivity (ODBC)	4
Java Database Connectivity (JDBC) and Embedded	
SQL for Java (SQLJ) Programs	4
Schemas	5
Tables	5
Keys	6
Primary Keys and Unique Keys	6
Referential Integrity	7
Check Constraints	9
Triggers	9
Indexes	11
Views	12
Aliases	13
Packages and Access Plans	13
Procedures	14
Catalog	15
Application Processes, Concurrency, and Recovery	15
Locking, Commit, and Rollback	17
Unit of Work	18
Rolling Back Work	19
Rolling back all changes	19
Rolling back selected changes using savepoints	19
Threads	20
Isolation Level	21
Comparison of Isolation Levels	24
Distributed Relational Database	25
Application Servers	26
CONNECT (Type 1) and CONNECT (Type 2)	27
Remote Unit of Work	27
Application-Directed Distributed Unit of Work	29
Data Representation Considerations	31
Character Conversion	32
Character Sets and Code Pages	33
Coded Character Sets and CCSIDs	34
Default CCSID	34
Sort Sequence	35

Authorization, Privileges and Object Ownership	36
Storage Structures	37

Chapter 2. Language Elements 39

Characters	39
Tokens	41
Identifiers	43
SQL Identifiers	43
System identifiers	43
Host Identifiers	44
Naming Conventions	45
SQL Path	53
Qualification of Unqualified Object Names	53
SQL Names and System Names: Special	
Considerations	55
Aliases	56
Authorization IDs and Authorization-Names	57
Example	58
Data Types	59
Nulls	61
Numbers	61
Character Strings	62
Character Encoding Schemes	63
Graphic Strings	64
Graphic Encoding Schemes	65
Binary Strings	65
Large Objects (LOBs)	65
Limitations on Use of Strings	67
Datetime Values	67
DataLink Values	72
Row ID Values	73
User-Defined Types	73
Promotion of Data Types	75
Casting Between Data Types	77
Assignments and Comparisons	80
Numeric Assignments	82
String Assignments	83
Datetime Assignments	85
DataLink Assignments	86
Row ID Assignments	87
Distinct Type Assignments	88
Numeric Comparisons	89
String Comparisons	89
Datetime Comparisons	91
Distinct Type Comparisons	91
Rules for Result Data Types	93
Numeric Operands	93
Character and Graphic String Operands	94
Binary String Operands	95
Datetime Operands	95
Distinct Type Operands	96
DataLink Operands	96
Conversion Rules for Operations That Combine	
Strings	97
Constants	99

Integer Constants	99
Floating-Point Constants	99
Decimal Constants	99
Character-String Constants	99
Graphic-String Constants	100
Binary-String Constants	102
Decimal Point	102
Delimiters	103
Special Registers	105
CURRENT DATE or CURRENT_DATE	105
CURRENT PATH, CURRENT_PATH, or CURRENT FUNCTION PATH	105
CURRENT SCHEMA	106
CURRENT SERVER or CURRENT_SERVER	106
CURRENT TIME or CURRENT_TIME	106
CURRENT TIMESTAMP or CURRENT_TIMESTAMP	107
CURRENT TIMEZONE or CURRENT_TIMEZONE	107
USER	107
Column Names	108
Qualified Column Names	108
Correlation Names	108
Column Name Qualifiers to Avoid Ambiguity	110
Column Name Qualifiers in Correlated References	111
Unqualified Column Names	112
References to Variables	114
References to Host Variables	114
Host Variables in Dynamic SQL	117
References to LOB Host Variables	117
References to LOB Locator Variables	117
References to LOB File Reference Variables	118
Host Structures	120
Host Structure Arrays	122
Functions	123
Types of Functions	123
Function resolution	124
Determining the Best Fit	125
Function Invocation	127
Expressions	129
Without Operators	129
With Arithmetic Operators	129
With the Concatenation Operator	131
Scalar Subselect	133
Datetime Operands and Durations	134
Datetime Arithmetic in SQL	134
Precedence of Operations	138
CASE Expressions	140
CAST Specification	143
Predicates	147
Basic Predicate	148
Quantified Predicate	149
BETWEEN Predicate	151
EXISTS Predicate	152
IN Predicate	153
LIKE Predicate	155
NULL Predicate	160
Search Conditions	161
Examples	162

Chapter 3. Built-In Functions	163
Column Functions	168
AVG	169
COUNT	170
COUNT_BIG	172
MAX	173
MIN	174
STDDEV_POP or STDDEV	175
SUM	176
VAR_POP or VARIANCE or VAR	177
Scalar Functions	178
Example	178
ABS	179
ACOS	180
ANTILOG	181
ASIN	182
ATAN	183
ATANH	184
ATAN2	185
BIGINT	186
BLOB	187
CEILING	189
CHAR	190
CHARACTER_LENGTH	195
CLOB	196
COALESCE	200
CONCAT	201
COS	202
COSH	203
COT	204
CURDATE	205
CURTIME	206
DATE	207
DAY	209
DAYOFMONTH	210
DAYOFWEEK	211
DAYOFWEEK_ISO	212
DAYOFYEAR	213
DAYS	214
DBCLOB	215
DECIMAL or DEC	217
DEGREES	220
DIFFERENCE	221
DIGITS	222
DLCOMMENT	223
DLLINKTYPE	224
DLURLCOMPLETE	225
DLURLPATH	226
DLURLPATHONLY	227
DLURLSCHEME	228
DLURLSERVER	229
DLVALUE	230
DOUBLE_PRECISION or DOUBLE	232
EXP	234
FLOAT	235
FLOOR	236
GRAPHIC	237
HASH	240
HEX	241
HOUR	242
IDENTITY_VAL_LOCAL	243

IFNULL	247
INTEGER or INT	248
JULIAN_DAY	250
LAND	251
LCASE	252
LEFT	253
LENGTH	255
LN	256
LNOT	257
LOCATE	258
LOG10	259
LOR	260
LOWER	261
LTRIM	262
MAX	263
MICROSECOND	264
MIDNIGHT_SECONDS	265
MIN	266
MINUTE	267
MOD	268
MONTH	270
NODENAME	271
NODENUMBER	272
NOW	273
NULLIF	274
PARTITION	275
PI	276
POSITION or POSSTR	277
POWER	279
QUARTER	280
RADIANS	281
RAND	282
REAL	283
ROUND	284
ROWID	286
RRN	287
RTRIM	288
SECOND	289
SIGN	290
SIN	291
SINH	292
SMALLINT	293
SOUNDEX	294
SPACE	295
SQRT	296
STRIP	297
SUBSTRING or SUBSTR	298
TAN	301
TANH	302
TIME	303
TIMESTAMP	304
TIMESTAMPDIFF	306
TRANSLATE	307
TRIM	309
TRUNCATE or TRUNC	311
UCASE	313
UPPER	314
VALUE	315
VARCHAR	316
VARGRAPHIC	320
WEEK	323

WEEK_ISO	324
XOR	325
YEAR	326
ZONED	327

Chapter 4. Queries 329

Authorization	329
subselect	330
select-clause	331
from-clause	335
where-clause	340
group-by-clause	341
having-clause	343
Examples of a subselect	344
fullselect	346
Rules for Columns	346
Examples of a fullselect	348
select-statement	349
common-table-expression	350
order-by-clause	351
fetch-first-clause	353
update-clause	354
read-only-clause	355
optimize-clause	356
isolation-clause	357
Examples of a select-statement	358

Chapter 5. Statements 361

How SQL Statements Are Invoked	365
Embedding a Statement in an Application Program	365
Dynamic Preparation and Execution	366
Static Invocation of a select-statement	366
Dynamic Invocation of a select-statement	367
Interactive Invocation	367
SQL Return Codes	367
SQLSTATE	368
SQLCODE	368
SQL Comments	368
ALTER TABLE	370
Invocation	370
Authorization	370
Syntax	371
Description	376
ADD <i>column-definition</i>	376
ALTER <i>column-alteration</i>	382
DROP COLUMN	384
ADD unique-constraint	384
ADD referential-constraint	385
ADD check-constraint	387
DROP	388
Notes	388
Cascaded Effects	390
Examples	392
BEGIN DECLARE SECTION	394
Invocation	394
Authorization	394
Syntax	394
Description	394
Examples	395

CALL	396	Syntax	440
Invocation	396	Description	442
Authorization	396	Notes	452
Syntax	396	Examples	453
Description	397	CREATE FUNCTION (External Table)	455
Notes	399	Invocation	455
Examples	401	Authorization	455
CLOSE	402	Syntax	456
Invocation	402	Description	458
Authorization	402	Notes	466
Syntax	402	Example	467
Description	402	CREATE FUNCTION (Sourced)	469
Notes	402	Invocation	469
Example	403	Authorization	469
COMMENT	404	Syntax	470
Invocation	404	Description	472
Authorization	404	Notes	475
Syntax	406	Examples	476
Description	408	CREATE FUNCTION (SQL Scalar)	478
Notes	412	Invocation	478
Examples	412	Authorization	478
COMMIT	413	Syntax	478
Invocation	413	Description	481
Authorization	413	Notes	484
Syntax	413	Example	486
Description	413	CREATE FUNCTION (SQL Table)	487
Notes	413	Invocation	487
Example	415	Authorization	487
CONNECT (Type 1)	416	Syntax	487
Invocation	416	Description	490
Authorization	416	Notes	493
Syntax	416	Example	495
Description	416	CREATE INDEX	496
Notes	417	Invocation	496
Examples	420	Authorization	496
CONNECT (Type 2)	421	Syntax	496
Invocation	421	Description	497
Authorization	421	Notes	498
Syntax	421	Examples	499
Description	421	CREATE PROCEDURE	500
Notes	423	Notes	500
Examples	424	CREATE PROCEDURE (External)	501
CREATE ALIAS	425	Invocation	501
Invocation	425	Authorization	501
Authorization	425	Syntax	501
Syntax	425	Description	504
Description	425	Notes	510
Notes	426	Example	511
Examples	427	CREATE PROCEDURE (SQL)	512
CREATE DISTINCT TYPE	428	Invocation	512
Invocation	428	Authorization	512
Authorization	428	Syntax	513
Syntax	428	Description	516
Description	429	Notes	518
Notes	431	Example	520
Examples	434	CREATE SCHEMA	521
CREATE FUNCTION	435	Invocation	521
Notes	435	Authorization	521
CREATE FUNCTION (External Scalar)	439	Syntax	521
Invocation	439	Description	522
Authorization	439	Notes	523

Examples	524	Example	607
CREATE TABLE	525	DECLARE VARIABLE	609
Invocation	525	Invocation	609
Authorization	525	Authorization	609
Syntax	526	Syntax	609
Description	532	Description	609
column-definition	532	Notes	610
LIKE	543	Example	610
as-subquery-clause	544	DELETE	612
copy-options	545	Invocation	612
unique-constraint	546	Authorization	612
referential-constraint	547	Syntax	613
check-constraint	548	Description	613
nodegroup-clause	549	DELETE Rules	615
Notes	550	Notes	615
Rules for System Name Generation	553	Examples	616
Examples	554	DESCRIBE	618
CREATE TRIGGER	556	Invocation	618
Invocation	556	Authorization	618
Authorization	556	Syntax	618
Syntax	558	Description	618
Description	559	Notes	620
Notes	564	Example	621
Examples	567	DESCRIBE TABLE	623
CREATE VIEW	569	Invocation	623
Invocation	569	Authorization	623
Authorization	569	Syntax	623
Syntax	570	Description	623
Description	570	Notes	625
Notes	573	Example	625
Examples	574	DISCONNECT	627
DECLARE CURSOR	576	Invocation	627
Invocation	576	Authorization	627
Authorization	576	Syntax	627
Syntax	577	Description	627
Description	577	Notes	628
Notes	579	Examples	628
Examples	581	DROP	629
DECLARE GLOBAL TEMPORARY TABLE	583	Invocation	629
Invocation	583	Authorization	629
Authorization	583	Syntax	630
Syntax	584	Description	632
Description	587	Note	637
column-definition	587	Examples	638
LIKE	592	END DECLARE SECTION	639
as-subquery-clause	593	Invocation	639
copy-options	594	Authorization	639
Notes	596	Syntax	639
Examples	597	Description	639
DECLARE PROCEDURE	598	Examples	639
Invocation	598	EXECUTE	640
Authorization	598	Invocation	640
Syntax	598	Authorization	640
Description	601	Syntax	640
Notes	606	Description	640
Example	606	Notes	641
DECLARE STATEMENT	607	Example	642
Invocation	607	EXECUTE IMMEDIATE	643
Authorization	607	Invocation	643
Syntax	607	Authorization	643
Description	607	Syntax	643

Description	643	Invocation	673
Note	644	Authorization	673
Example	644	Syntax.	674
FETCH	645	Description	675
Invocation	645	insert-multiple-rows	677
Authorization	645	INSERT Rules	677
Syntax.	645	Notes	678
Description	646	Examples.	679
single-fetch	647	LABEL	681
multiple-row-fetch.	647	Invocation	681
Notes	649	Authorization	681
Example	650	Syntax.	681
FREE LOCATOR	651	Description	682
Invocation	651	Notes	682
Authorization	651	Examples.	683
Syntax.	651	LOCK TABLE	684
Description	651	Invocation	684
Example	651	Authorization	684
GRANT (Distinct Type Privileges)	652	Syntax.	684
Invocation	652	Description	684
Authorization	652	Notes	685
Syntax.	652	Example	685
Description	652	OPEN	686
Note	653	Invocation	686
Example	654	Authorization	686
GRANT (Function or Procedure Privileges)	655	Syntax.	686
Invocation	655	Description	686
Authorization	655	Notes	687
Syntax.	655	Examples.	689
Description	657	PREPARE	691
Note	660	Invocation	691
Example	661	Authorization	691
GRANT (Package Privileges)	662	Syntax.	691
Invocation	662	Description	692
Authorization	662	Notes	694
Syntax.	662	Examples.	698
Description	662	RELEASE (Connection)	700
Note	663	Invocation	700
Example	663	Authorization	700
GRANT (Table or View Privileges)	664	Syntax.	700
Invocation	664	Description	700
Authorization	664	Notes	701
Syntax.	664	Examples.	701
Description	665	RELEASE SAVEPOINT	702
Notes	666	Invocation	702
Examples.	668	Authorization	702
HOLD LOCATOR	669	Syntax.	702
Invocation	669	Description	702
Authorization	669	Note	702
Syntax.	669	Example	702
Description	669	RENAME	703
Note	669	Invocation	703
Example	669	Authorization	703
INCLUDE	671	Syntax.	703
Invocation	671	Description	703
Authorization	671	Notes	704
Syntax.	671	Examples.	705
Description	671	REVOKE (Distinct Type Privileges)	706
Notes	672	Invocation	706
Example	672	Authorization	706
INSERT	673	Syntax.	706

Description	706	Invocation	747
Notes	707	Authorization	747
Example	707	Syntax	747
REVOKE (Function or Procedure Privileges)	708	Description	747
Invocation	708	Notes	748
Authorization	708	Example	749
Syntax	708	SET RESULT SETS	750
Description	710	Invocation	750
Notes	713	Authorization	750
Example	714	Syntax	750
REVOKE (Package Privileges)	715	Description	750
Invocation	715	Notes	751
Authorization	715	Example	752
Syntax	715	SET SCHEMA	753
Description	715	Invocation	753
Notes	716	Authorization	753
Example	716	Syntax	753
REVOKE (Table or View Privileges)	717	Description	753
Invocation	717	Notes	753
Authorization	717	Examples	754
Syntax	717	SET TRANSACTION	755
Description	717	Invocation	755
Notes	718	Authorization	755
Examples	719	Syntax	755
ROLLBACK	720	Description	755
Invocation	720	Notes	756
Authorization	720	Examples	757
Syntax	720	SET transition-variable	758
Description	720	Invocation	758
Notes	721	Authorization	758
Examples	723	Syntax	758
SAVEPOINT	724	Description	758
Invocation	724	Notes	759
Authorization	724	Examples	759
Syntax	724	SET variable	760
Description	724	Invocation	760
Note	725	Authorization	760
Example	725	Syntax	760
SELECT	726	Description	760
SELECT INTO	727	Notes	761
Invocation	727	Examples	761
Authorization	727	UPDATE	762
Syntax	727	Invocation	762
Description	728	Authorization	762
Note	728	Syntax	764
Examples	729	Description	765
SET CONNECTION	730	UPDATE Rules	768
Invocation	730	Notes	768
Authorization	730	Examples	770
Syntax	730	VALUES	771
Description	730	Invocation	771
Notes	731	Authorization	771
Example	731	Syntax	771
SET OPTION	733	Description	771
Invocation	733	Notes	771
Authorization	733	Examples	771
Syntax	733	VALUES INTO	773
Description	737	Invocation	773
Notes	746	Authorization	773
Examples	746	Syntax	773
SET PATH	747	Description	773

Notes	773	Notes	807
Examples	774	Examples	807
WHENEVER	776	LOOP statement	808
Invocation	776	Syntax	808
Authorization	776	Description	808
Syntax	776	Examples	808
Description	776	REPEAT statement	809
Notes	776	Syntax	809
Example	777	Description	809
		Example	809
Chapter 6. SQL Control Statements	779	RESIGNAL statement	811
References to SQL Parameters and Variables	781	Syntax	811
SQL procedure statement	782	Description	811
Syntax	782	Notes	812
Notes	783	Example	813
assignment-statement	784	RETURN statement	814
Syntax	784	Syntax	814
Description	784	Description	814
Notes	785	Notes	814
Example	785	Example	815
CALL statement	786	SIGNAL statement	816
Syntax	786	Syntax	816
Description	786	Description	816
Notes	786	Notes	817
Example	787	Example	818
CASE statement	788	WHILE statement	819
Syntax	788	Syntax	819
Description	788	Description	819
Notes	789	Example	819
Examples	789		
compound-statement	790	Appendix A. SQL Limits 821	
Syntax	790	Appendix B. SQL Communication	
Description	793	Area 825	
Notes	795	Field Descriptions	825
Examples	796	INCLUDE SQLCA Declarations	831
FOR statement	797		
Syntax	797	Appendix C. Characteristics of SQL	
Description	797	Statements 835	
Notes	798	Actions Allowed on SQL Statements	836
Example	798	SQL Statement Data Access Indication in Routines	838
GET DIAGNOSTICS statement	799	Considerations for Using Distributed Relational	
Syntax	799	Database	840
Description	799	CONNECT (Type 1) and CONNECT (Type 2)	
Notes	800	Differences	845
Example	800		
GOTO statement	802	Appendix D. SQL Descriptor Area	
Syntax	802	(SQLDA) 847	
Description	802	Field Descriptions in an SQLDA Header	849
Notes	802	Determining How Many SQLVAR Occurrences	
Example	802	are Needed	850
IF statement	804	Field Descriptions in an Occurrence of SQLVAR	852
Syntax	804	SQLTYPE and SQLLEN	854
Description	804	CCSID Values in SQLDATA or SQLNAME	856
Example	804	Unrecognized and Unsupported SQLTYPES	856
ITERATE statement	806	INCLUDE SQLDA Declarations	857
Syntax	806	For C and C++	857
Description	806	For COBOL	860
Example	806	For ILE COBOL	860
LEAVE statement	807		
Syntax	807		
Description	807		

For PL/I	861		SQLCOLPRIVILEGES	950
For ILE RPG/400	862		SQLCOLUMNS	951
Appendix E. CCSID Values.	863		SQLFOREIGNKEYS	956
Appendix F. DB2 UDB for iSeries			SQLPRIMARYKEYS	957
Catalog Views	881		SQLPROCEDURECOLS	958
Notes	882		SQLPROCEDURES	962
iSeries Catalog Tables and Views	884		SQLSCHEMAS	963
SYSCATALOGS	885		SQLSPECIALCOLUMNS	964
SYSCHKCST	887		SQLSTATISTICS	966
SYSCOLUMNS	888		SQLTABLEPRIVILEGES	967
SYSCST	897		SQLTABLES	968
SYSCSTCOL	898		SQLTYPEINFO	969
SYSCSTDEP	899		SQLUDTS	974
SYSFUNCS	900		ANS and ISO Catalog Views	976
SYSINDEXES	906		CHARACTER_SETS	977
SYSJARCONTENTS	907		CHECK_CONSTRAINTS	978
SYSJAROBJECTS	908		COLUMNS	979
SYSKEYCST	909		INFORMATION_SCHEMA_CATALOG_NAME	983
SYSKEYS	910		PARAMETERS	984
SYSPACKAGE	911		REFERENTIAL_CONSTRAINTS	988
SYSPARMS	913		ROUTINES	989
SYSPROCS	917		SCHEMATA	997
SYSREFCST	922		SQL_FEATURES	998
SYSRoutineDEP	923		SQL_LANGUAGES	999
SYSROUTINES	924		SQL_SIZING	1000
SYSTABLES	932		TABLE_CONSTRAINTS	1001
SYSTRIGCOL	934		TABLES	1002
SYSTRIGDEP	935		USER_DEFINED_TYPES	1003
SYSTRIGGERS	936		VIEWS	1007
SYSTRIGUPD	940		Appendix G. Reserved Words	1009
SYSTYPES	941		Bibliography	1011
SYSVIEWDEP	946		Index	1013
SYSVIEWS	948			
ODBC and JDBC Catalog Views	949			

About DB2 UDB for iSeries SQL Reference

This book defines Structured Query Language (SQL) as supported by DB2 Query Manager and SQL Development Kit. It contains reference information for the tasks of system administration, database administration, application programming, and operation. This manual includes syntax, usage notes, keywords, and examples for each of the SQL statements used on the system.

For more information about this guide, see the following sections:

- “Standards Compliance”
- “Who should read the SQL Reference book”
- “Assumptions Relating to Examples of SQL Statements” on page xiv
- “How to Read Syntax Diagrams” on page xv
- “Conventions for Describing Mixed Data Values” on page xvii
- “SQL Accessibility” on page xvii
- “What’s new for V5R2 in the SQL Reference book” on page xviii

Standards Compliance

DB2 UDB for iSeries Version 5 Release 2 complies with the following IBM and Industry SQL Standards:

- ISO (International Standards Organization) 9075: 1992, Database Language SQL - Entry Level
- ISO (International Standards Organization) 9075-4: 1996, Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM)
- ISO (International Standards Organization) 9075: 1999, Database Language SQL - Core
- ANSI (American National Standards Institute) X3.135-1992, Database Language SQL - Entry Level
- ANSI (American National Standards Institute) X3.135-4: 1996, Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM)
- ANSI (American National Standards Institute) X3.135-1999, Database Language SQL - Core
- *IBM SQL Reference Version 2, SC26-8416.*

For strict adherence to the standards, consider using the standards option. For more information, see SQLCURRULE in “SET OPTION” on page 733 and on the SQL precompiler commands.

Who should read the SQL Reference book

This book is intended for programmers who want to write applications that will use SQL to access an iSeries database.

It is assumed that you possess an understanding of system administration, database administration, or application programming for the iSeries server, as provided by the SQL Programming Concepts book and that you have some knowledge of the following:

- COBOL for iSeries

- ILE C compiler
- ILE C++ compiler
- ILE COBOL compiler
- Toolbox for Java or Developer Kit for Java
- ILE RPG compiler
- iSeries PL/I
- REXX
- RPG III (part of RPG for iSeries)
- Structured Query Language (SQL)

References in this book to RPG and COBOL refer to the RPG or COBOL language in general. References to COBOL for iSeries, ILE COBOL for iSeries, RPG for iSeries, or RPG III (part of RPG for iSeries) refer to specific elements of the product where they differ from each other.

This manual is a reference rather than a tutorial. It assumes you are already familiar with SQL programming. This manual also assumes that you will be writing applications solely for the iSeries server.

If you need more information about using SQL statements, statement syntax, and parameters, see the SQL Programming Concepts book.

If you are planning applications that are portable to other IBM environments, it will be necessary for you to refer to books for those environments in addition to this one (such as *IBM SQL Reference Version 2, SC26-8416*).

See the following sections for more details:

- “Assumptions Relating to Examples of SQL Statements”
- “How to Read Syntax Diagrams” on page xv

Assumptions Relating to Examples of SQL Statements

The examples of SQL statements shown in this guide are based on the sample tables in Appendix A of the SQL Programming Concepts book and assume the following:

- They are shown in the interactive SQL environment or written in COBOL. EXEC SQL and END-EXEC are used to delimit an SQL statement in a COBOL program. A description of how to use SQL statements in a COBOL program is provided in the SQL Programming with Host Languages book.
- Each SQL example is shown on several lines, with each clause of the statement on a separate line.
- SQL keywords are highlighted.
- Table names used in the examples are the sample tables provided in Appendix A of the SQL Programming Concepts book and use the schema CORPDATA. Table names that are not provided in that appendix should use schemas that you create. You can create a set of sample tables in your own schema by issuing the following SQL statement:

```
CALL QSYS.CREATE_SQL_SAMPLE ('your-schema-name')
```
- Calculated columns are enclosed in parentheses, ().
- The SQL naming convention is used.
- The APOST and APOSTSQL precompiler options are assumed (although they are not the default in COBOL). Character-string constants within SQL and host language statements are delimited by apostrophes (').

- A sort sequence of *HEX is used.

Whenever the examples vary from these assumptions, it is stated.

See also “Code disclaimer information”.

Code disclaimer information

This document contains programming examples.

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you "AS IS" without any warranties of any kind. The implied warranties of non-infringement, merchantability and fitness for a particular purpose are expressly disclaimed.

How to Read Syntax Diagrams

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

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

The ►— symbol indicates the beginning of a statement.

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

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

The —►◄ symbol indicates the end of a statement.

Syntax fragments start with the |— symbol and end with the —| symbol.

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



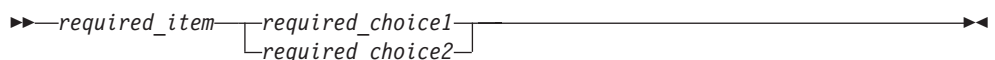
- Optional items appear below the main path.



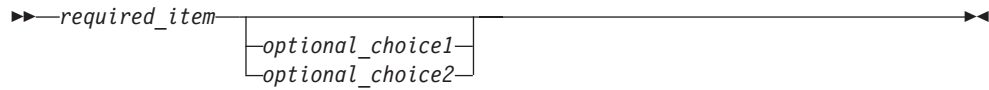
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



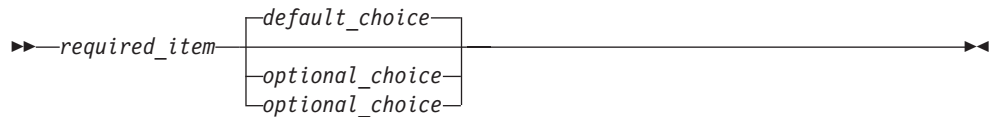
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



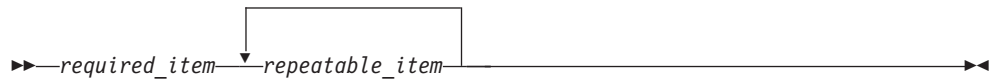
If choosing one of the items is optional, the entire stack appears below the main path.



If one of the items is the default, it 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.



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



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

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.
- The syntax diagrams only contain the preferred or standard keywords. If non-standard synonyms are supported in addition to the standard keywords, they are described the **Notes** sections instead of the syntax diagrams. For maximum portability, only the preferred or standard keywords should be used.

|
|
|
|

Conventions for Describing Mixed Data Values

When mixed data values are shown in the examples, the following conventions apply:


Convention	Meaning
S_0	Represents the EBCDIC <i>shift-out</i> control character (X'0E')
S_I	Represents the EBCDIC <i>shift-in</i> control character (X'0F')
<i>sbc</i> s-string	Represents a string of zero or more single-byte characters
<i>dbc</i> s-string	Represents a string of zero or more double-byte characters
'	Represents a DBCS apostrophe (EBCDIC X'427D')
G	Represents a DBCS G (EBCDIC X'42C7')

SQL Accessibility


IBM is committed to providing interfaces and documentation that are easily accessible to the disabled community. For general information on IBM's Accessibility support visit the Accessibility Center at <http://www.ibm.com/able>.



SQL accessibility support falls in two main categories.

- iSeries Navigator is graphical user interface to iSeries and DB2 UDB. For information about the Accessibility features supported in Windows graphical user interfaces, see Accessibility in the Windows Help Index.
- Online documentation, online help, and prompted SQL interfaces can be accessed by a Windows Reader program such as the IBM Home Page Reader. For information on the IBM Home Page Reader and other tools, visit the Accessibility Center .

The IBM Home Page Reader can be used to access all descriptive text in this book, all articles in the SQL Information Center, and all SQL messages. Due to the complex nature of SQL syntax diagrams, however, the reader will skip syntax diagrams. Two alternatives are provided for better ease of use:

- Interactive SQL and Query Manager
Interactive SQL and Query Manager are traditional file interfaces that provide prompting for SQL statements. These are part of the DB2 UDB Query Manager and SQL Development Kit. For more information about Interactive SQL and Query Manager, see the SQL Programming Concepts and Query Manager Use
-  SQL Assist
SQL Assist is a graphical user interface that provides a prompted interface to SQL statements. This is part of iSeries Navigator. For more information, see the iSeries Navigator online help and the Information Center.

What's new for V5R2 in the SQL Reference book

The major new features covered in this book include:

- ROWID data type and ROWID scalar function
- IDENTITY column attribute
- CREATE TABLE AS (subselect)
- DECLARE GLOBAL TEMPORARY tables
- User-defined Table Functions
- COMMIT ON RETURN procedures
- UNION in views
- Scalar subselect enhancements
- READ ONLY and READ WRITE in SET TRANSACTION
- ITERATE and nested Compound statements in SQL procedures, SQL functions, and SQL triggers
- Fullselect in derived tables and common table expressions
- Parameter markers in labeled durations
- Savepoints
- SET SCHEMA and SET SQLID
- HOLD LOCATOR
- ORDER BY expression not required in the select-list
- ORDER BY and FETCH FIRST n ROWS ONLY in derived tables and common table expressions
- Length of SQL statements increased to 64K
- Length of delimited column name identifiers increased
- SUBSTRING enhancements
- VARCHAR concatenation enhancement
- Debug of original source statements in SQL procedures, SQL functions, and SQL triggers
- Multiple relational databases on iSeries
- Standard and ODBC and JDBC catalog views
- C derived variables

Chapter 1. Concepts

DB2 UDB for &is; SQL Reference describes the following concepts:

- “Relational Database”
- “Structured Query Language” on page 3
- “Schemas” on page 5
- “Tables” on page 5
- “Keys” on page 6
- “Primary Keys and Unique Keys” on page 6
- “Referential Integrity” on page 7
- “Check Constraints” on page 9
- “Triggers” on page 9
- “Indexes” on page 11
- “Views” on page 12
- “Aliases” on page 13
- “Packages and Access Plans” on page 13
- “Procedures” on page 14
- “Catalog” on page 15
- “Application Processes, Concurrency, and Recovery” on page 15
- “Threads” on page 20
- “Isolation Level” on page 21
- “Distributed Relational Database” on page 25
- “Character Conversion” on page 32
- “Sort Sequence” on page 35
- “Authorization, Privileges and Object Ownership” on page 36
- “Storage Structures” on page 37

Relational Database

| A *relational database* is a database that can be perceived as a set of tables and can be
| manipulated in accordance with the relational model of data. The relational
| database contains a set of objects used to store, access, and manage data. The set of
| objects includes tables, views, indexes, aliases, distinct types, functions, procedures
| and packages.

| There are three types of relational databases a user can access from an iSeries
| system.

| **system relational database**

| There is one default relational database on any iSeries system. The system
| relational database is always local to that iSeries system. It consists of all
| the database objects that exist on disk attached to the iSeries system that
| are not stored on independent auxiliary storage pools. For more
| information on independent auxiliary storage pools, see the System
| Management category of the iSeries Information Center.

The name of the system relational database is, by default, the same as the iSeries system name. However, a different name can be assigned through the use of the ADDRDBDIRE (Add RDB Directory Entry) command or iSeries Navigator.

user relational database

The user may create additional relational databases on an iSeries system by configuring independent auxiliary storage pools on the system. Each primary independent auxiliary storage pool is a relational database. It consists of all the database objects that exist on the independent auxiliary storage pool disks. Additionally, all database objects in the system relational database of the iSeries system to which the independent auxiliary storage pool is connected are logically included in a user relational database. Thus, the name of any schema created in a user relational database must not already exist in that user relational database or in the associated system relational database.

Although the objects in the system relational database are logically included in a user relational database, certain dependencies between the objects in the system relational database and the user relational database are not allowed:

- A view must be created into a schema that exists in the same relational database as its referenced tables, views, or functions.
- An index must be created into a schema that exists in the same relational database as its referenced table.
- A trigger or constraint must be created into a schema that exists in the same relational database as its base table.
- The parent table and dependent table in a referential constraint must both exist in the same relational database.
- A table must be created into a schema that exists in the same relational database as any referenced distinct types.
- The parent table and dependent table in a referential constraint must both exist in the same relational database.

Other dependencies between the objects in the system relational database and the user relational database are allowed. For example, a procedure in a schema in a user relational database may reference objects in the system relational database. However, operations on such an object may fail if the other relational database is not available. For example, if a user relational database is varied off and then varied on to another system.

A user relational database is local to an iSeries system while the independent auxiliary storage pool is varied on. Independent auxiliary storage pools can be varied off on one iSeries system and then varied on to another iSeries system. Hence, a user relational databases may be local to a given iSeries system at one point in time and remote at a different point in time. For more information on independent auxiliary storage pools, see the System Management category of the iSeries Information Center.

The name of the user relational database is, by default, the same as the independent auxiliary storage pool name. However, a different name can be assigned through the use of the ADDRDBDIRE (Add RDB Directory Entry) command or iSeries Navigator.

remote relational database

Relational databases on other iSeries and non-iSeries systems can be

| accessed remotely. These relational databases must be registered through
| the use of the ADDRDBDIRE (Add RDB Directory Entry) command or
| iSeries Navigator.

The database manager is the name used generically to identify the iSeries Licensed Internal Code and the DB2 UDB for iSeries portion of the code that manages the relational database.

Structured Query Language

Structured Query Language (SQL) is a standardized language for defining and manipulating data in a relational database. In accordance with the relational model of data, the database is perceived as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more 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. This transformation occurs when the SQL statement is *prepared*. This transformation is also known as *binding*.

All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or operational form of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish static SQL from dynamic SQL.

Static SQL

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

A source program containing static SQL statements must be processed by an SQL precompiler before it is compiled. The precompiler checks the syntax of the SQL statements, turns them into host language comments, and generates host language statements to call the database manager.

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

Dynamic SQL

A *dynamic* SQL statement is prepared during the execution of an SQL application. The operational form of the statement persists until the last SQL program leaves the call stack. The source form of the statement is a character string that is passed to the database manager by the program using the static SQL statement PREPARE or EXECUTE IMMEDIATE.

SQL statements embedded in a REXX application are dynamic SQL statements. SQL statements submitted to the interactive SQL facility are also dynamic SQL statements.

Extended Dynamic SQL

An extended dynamic SQL statement is neither fully static nor fully dynamic. The QSQRCE API provides users with extended dynamic SQL capability. Like dynamic SQL, statements can be prepared, described, and executed using this API. Unlike dynamic SQL, SQL statements prepared into a package by this API persist until the package or statement is explicitly dropped. For more information, see the OS/400 APIs information in the **Programming** category of the iSeries Information Center.

Interactive SQL

An interactive SQL facility is associated with every database manager. Essentially, every interactive SQL facility is an SQL application program that reads statements from a terminal, prepares and executes them dynamically, and displays the results to the user. Such SQL statements are said to be issued *interactively*.

The interactive facilities for DB2 UDB for iSeries are invoked by the STRSQL command, the STRQM command, or the SQL Script support of iSeries Navigator. For more information about the interactive facilities for SQL, see the SQL

Programming Concepts and Query Manager Use  books.

SQL Call Level Interface (CLI) and Open Database Connectivity (ODBC)

The DB2 Call Level Interface is an application programming interface in which functions are provided to application programs to process dynamic SQL statements. DB2 CLI allows users of any of the ILE languages to access SQL functions directly through procedure calls to a service program provided by DB2 UDB for iSeries. CLI programs can also be compiled using an Open Database Connectivity (ODBC) Software Developer's Kit, available from Microsoft or other vendors, enabling access to ODBC data sources. Unlike using embedded SQL, no precompilation is required. Applications developed using this interface may be executed on a variety of databases without being compiled against each of the databases. Through the interface, applications use procedure calls at execution time to connect to databases, to issue SQL statements, and to get returned data and status information.

The DB2 CLI interface provides many features not available in embedded SQL. For example:

- 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 application server specific catalog queries.
- Stored procedures called from application programs written using CLI can return result sets to those programs.

For a complete description of all the available functions, and their syntax, see SQL Call Level Interfaces (ODBC) book.

Java Database Connectivity (JDBC) and Embedded SQL for Java (SQLJ) Programs

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

JDBC calls are translated to calls to DB2 CLI through Java native methods. You can access iSeries databases through two JDBC drivers: IBM Developer Kit for Java driver or IBM Toolbox for Java JDBC driver. For specific information about the IBM Toolbox for Java JDBC driver, see IBM Toolbox for Java.

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

For more information about JDBC and SQLJ applications, refer to the Developer Kit for Java book.

Schemas

A schema is a collection of named objects. Schemas provide a logical classification of objects in a relational database. Some of the objects that a schema may contain include tables, views, aliases, functions, procedures, types, and packages. A schema is also called a collection or library.

A schema is also an object in the relational database. It is explicitly created using the CREATE SCHEMA statement.¹

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 the 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 creates a schema called C:

```
CREATE SCHEMA C
```

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

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

Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows. There is no inherent order of the rows within a table. At the intersection of every column and row is a specific data item called a *value*. A *column* is a set of values of the same type. A *row* is a sequence of values such that the *n*th value is a value of the *n*th column of the table.

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.

1. A schema can also be created using the CRTLIB CL command, however, the catalog views and journal created by using the CREATE SCHEMA statement will not be created with CRTLIB.

A base table has a name and may have a different system name. The system name is the name used by OS/400. Either name is acceptable wherever a *table-name* is specified in SQL statements. For more information see “CREATE TABLE” on page 525.

A column of a base table has a name and may have a different system column name. The system column name is the name used by OS/400. Either name is acceptable wherever *column-name* is specified in SQL statements. For more information see “CREATE TABLE” on page 525.

A *distributed table* is a table whose data is partitioned across a *nodegroup*. A *nodegroup* is an object that provides a logical grouping of a set of two or more systems. A *partitioning key* is a set of one or more columns in a distributed table that are used to determine on which system a row belongs. For more information about distributed tables, see the DB2 Multisystem book.

A *declared temporary table* is created with a DECLARE GLOBAL TEMPORARY TABLE statement and is used to hold temporary data on behalf of a single application. This table is dropped implicitly when the application disconnects from the database.

Keys

A *key* is one or more columns that are identified as such in the description of an index, unique constraint, or a 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.

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

Primary Keys and Unique Keys

A *unique constraint* is the rule that the values of a key are valid only if they are unique. A key that is constrained to have unique values is called a *unique key* and can be defined by using the CREATE UNIQUE INDEX statement. The resulting unique index is used by the database manager to enforce the uniqueness of the key during the execution of INSERT and UPDATE statements. Alternatively, unique keys can be defined:

- As a primary key using a CREATE TABLE or ALTER TABLE statement. A table cannot have more than one primary key. A CHECK constraint will be added implicitly to enforce the rule that the NULL value is not allowed in the columns that make up the primary key. A unique index on a primary key is called a primary index.
- Using the UNIQUE clause of the CREATE TABLE or ALTER TABLE statement. A table can have an arbitrary number of UNIQUE keys.

A unique key that is referenced by the foreign key of a referential constraint is called the parent key. A parent key is either a primary key or a UNIQUE key. When a table is defined as a parent in a referential constraint, the default parent key is its primary key.

Referential Integrity

Referential integrity is the state of a database in which all values of all foreign keys are valid. A *foreign key* is a key that is part of the definition of a referential constraint. 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, and DELETE statements. The enforcement is effectively performed at the completion of the statement except for delete and update rules of RESTRICT which are enforced as rows are processed.

Referential constraints with a delete or update rule of RESTRICT are always enforced before any other referential constraints. Other referential constraints are enforced in an order independent manner. That is, the order does not affect the result of the operation. Within an SQL statement:

- A row can be marked for deletion by any number of referential constraints with a delete rule of CASCADE.
- A row can only be updated by one referential constraint with a delete rule of SET NULL or SET DEFAULT.
- A row that was updated by a referential constraint cannot also be marked for deletion by another referential constraint with a delete rule of CASCADE.

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

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

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

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

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 dependent table is updated. The update rule of a referential constraint is that a nonnull update value of a 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.

The delete rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION, RESTRICT, CASCADE, SET NULL or SET DEFAULT. 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 is returned 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
- SET DEFAULT, each column of the foreign key of each dependent of p in D is set to its default value

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 if the deletion cascades to any of its descendants that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

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 SET DEFAULT, 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.

Check Constraints

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

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

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

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

Triggers

A *trigger* defines a set of actions that are executed automatically whenever a delete, insert, or update operation occurs on a specified table. When such an SQL operation is executed, the trigger is said to be activated.²

The set of actions can include almost any operation allowed on the system. A few operations are not allowed, such as:

- Commit or rollback (if the same commitment definition is used for the trigger actions and the triggering event)
- CONNECT, SET CONNECTION, DISCONNECT, and RELEASE statements

For a complete list of restrictions, see the Database Programming book.

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

Triggers are a useful mechanism to define and enforce transitional business rules that involve different states of the data (for example, salary cannot be increased by more than 10 percent). Such a limit requires comparing the value of a salary before

2. The ADDPFTRG CL command also defines a trigger that is activated on any read operation.

and after an increase. For rules that do not involve more than one state of the data, consider using referential and check constraints.

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

Triggers are optional and are defined using the CREATE TRIGGER statement or the ADDPFTRG (Add Physical File Trigger) CL command. Triggers are dropped using the DROP TRIGGER statement or the RMVPFTRG (Remove Physical File Trigger) CL command. For more information about creating triggers, see the CREATE TRIGGER statement. For more information about triggers in general, see the "CREATE TRIGGER" on page 556 statement or the SQL Programming Concepts and the Database Programming books.

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 *trigger 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.

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. Transition tables can only be used in after triggers. Separate transition tables can be defined for old and new values.

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, which may result in the activation of additional triggers. With trigger cascading, a significant chain of triggers may be activated causing significant change to the database as a result of a single delete, insert or update statement.

The actions performed in the trigger are considered to be part of the operation that caused the trigger to be executed. Thus, when the isolation level is anything other than NC (No Commit) and the trigger actions are performed using the same commitment definition as the trigger event:

- The database manager ensures that the operation and the triggers executed as a result of that operation either all complete or are backed out. Operations that occurred prior to the triggering operation are not affected.
- The database manager effectively checks all constraints (except for a constraint with a RESTRICT delete rule) after the operation and the associated triggers have been executed.

A trigger has an attribute that specifies whether it is allowed to delete or update a row that has already been inserted or updated within the SQL statement that caused the trigger to be executed.

- If ALWREPCHG(*YES) is specified when the trigger is defined, then within an SQL statement:
 - The trigger is allowed to update or delete any row that was inserted or already updated by that same SQL statement. This also includes any rows inserted or updated by a trigger or referential constraint caused by the same SQL statement.
- If ALWREPCHG(*NO) is specified when the trigger is defined, then within an SQL statement:
 - A row can be deleted by a trigger only if that row has not been inserted or updated by that same SQL statement. If the isolation level is anything other than NC (No Commit) and the trigger actions are performed using the same commitment definition as the trigger event, this also includes any inserts or updates by a trigger or referential constraint caused by the same SQL statement.
 - A row can be updated by a trigger only if that row has not already been inserted or updated by that same SQL statement. If the isolation level is anything other than NC (No Commit) and the trigger actions are performed using the same commitment definition as the trigger event, this also includes any inserts or updates by a trigger or referential constraint caused by the same SQL statement.

All triggers created by using the CREATE TRIGGER statement implicitly have the ALWREPCHG(*YES) attribute.

Indexes


An *index* is a 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 you request an index, the database manager builds this structure and maintains it automatically.

An index has a name and may have a different system name. The system name is the name used by OS/400. Either name is acceptable wherever an index-name is specified in SQL statements. For more information see “CREATE INDEX” on page 496.

The database manager uses two types of indexes:

- Binary radix tree index
Binary radix tree indexes provide a specific order to the rows of a table. The database manager uses them to:
 - Improve performance. In most cases, access to data is faster than without an index.
 - Ensure uniqueness. A table with a unique index cannot have rows with identical keys.
- Encoded vector index
Encoded vector indexes do not provide a specific order to the rows of a table. The database manager only uses these indexes to improve performance.

An encoded vector access path works with the help of encoded vector indexes and provides access to a database file by assigning codes to distinct key values and then representing these values in an array. The elements of the array can be 1, 2, or 4 bytes in length, depending on the number of distinct values that must be represented. Because of their compact size and relative simplicity, encoded vector access paths provide for faster scans that can be more easily processed in parallel.

You create encoded vector access paths by using the SQL CREATE INDEX statement. For more information about accelerating your queries with encoded vector indexes , go to the DB2 UDB for iSeries webpages.

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 effectively 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 569 for more information.)

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.

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 constraints as the base table. Likewise, if the base table of a view is a parent table, DELETE operations using that view are subject to the same rules as DELETE operations on the base table. A view also inherits any triggers that apply to its base table. For example, if the base table of a view has an update trigger, the trigger is fired when an update is performed on the view.

A view has a name and may have a different system name. The system name is the name used by OS/400. Either name is acceptable wherever a view-name is specified in SQL statements. For more information see “CREATE VIEW” on page 569.

A column of a view has a name and may have a different system column name. The system column name is the name used by OS/400. Either name is acceptable wherever column-name is specified in SQL statements. For more information, see “CREATE VIEW” on page 569.

Aliases

An *alias* is an alternate name for a table or view. You can use an alias to reference a table or view in those cases where an existing table or view can be referenced.³ Like tables and views, an alias may be created, dropped, and have a comment or label associated with it. No authority is necessary to use an alias. Access to the tables and views that are referred to by the alias, however, still require the appropriate authorization for the current statement.

An alias has a name and may have a different system name. The system name is the name used by OS/400. Either name is acceptable wherever an alias-name is specified in SQL statements. For more information see “CREATE ALIAS” on page 425.

Packages and Access Plans

For distributed programs, a *package* is an object that contains control structures used to execute SQL statements. For non-distributed SQL programs, the control structures used to execute SQL statements are stored in the associated space of the non-distributed SQL program. Packages are produced during program preparation. The control structures can be thought of as the bound or operational form of SQL statements. All control structures in a package are derived from the SQL statements embedded in a single source program.

In this book, the term *access plan* is used in general for packages, procedures, functions, triggers, and programs or service programs that contain control structures used to execute SQL statements. For example, the description of the DROP statement says that dropping an object also invalidates any access plans that reference the object (see “DROP” on page 629). This means that any packages, procedures, functions, triggers, and programs or service programs containing control structures referencing the dropped object are invalidated.

An invalidated *access plan* will be implicitly rebuilt the next time its associated SQL statement is executed. For example, if an index is dropped that is used in an *access plan* for a SELECT INTO statement, the next time that SELECT INTO statement is executed, the access plan will be rebuilt.

A package can also be created by the QSQPRCED API. Packages created by the QSQPRCED API can only be used by the QSQPRCED API. They cannot be used at a server through DRDA protocols. For more information, see the OS/400 APIs information in the **Programming** category of the iSeries Information Center.

The QSQPRCED API is used by iSeries Access for Windows to create packages for caching SQL statements executed via ODBC, JDBC and SQLJ interfaces.

3. You cannot use an alias in all contexts. For example, an alias that refers to an individual member of a database file cannot be used in data definition language (DDL) statements.

Procedures

A procedure (often called a stored procedure) is a programming construct that can be called to perform a set of operations. The operations can contain host language statements and SQL statements.

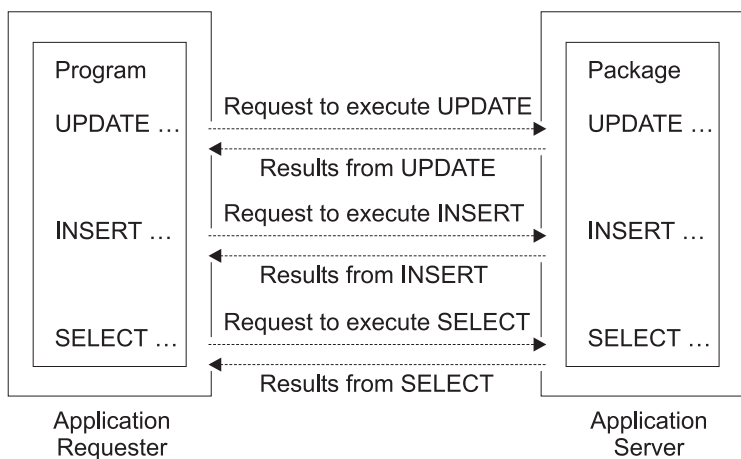
Procedures are classified as either SQL procedures or external procedures. SQL procedures contain only SQL statements. External procedures reference a host language program (or in the case of REXX, a source file member) which may or may not contain SQL statements. Both external procedures and SQL procedures are supported in DB2 UDB for iSeries.

Procedures in SQL provide the same benefits as procedures in a host language. That is, a common piece of code need only be written and maintained once and can be called from several programs. Both host languages and SQL can call procedures that exist on the local system. However, SQL can also call a procedure that exists on a remote system. In fact, the major benefit of procedures in SQL is that they can be used to enhance the performance characteristics of distributed applications.

Assume several SQL statements must be executed at a remote system. When the first SQL statement is executed, the application requester will send a request to a server to perform the operation. It will then wait for a reply that indicates whether the statement executed successfully or not and optionally returns results. When the second and each subsequent SQL statement is executed, the application requester will send another request and wait for another reply.

If the same SQL statements are stored in a procedure at a server, a CALL statement can be executed that references the remote procedure. When the CALL statement is executed, the application requester will send a single request to the current server to call the procedure. It will then wait for a single reply that indicates whether the procedure executed successfully or not and optionally returns results.

The following two figures illustrate the way stored procedures can be used in a distributed application to eliminate some of the remote requests.



RBAL3505-0

Figure 1. Application Without Remote Procedure

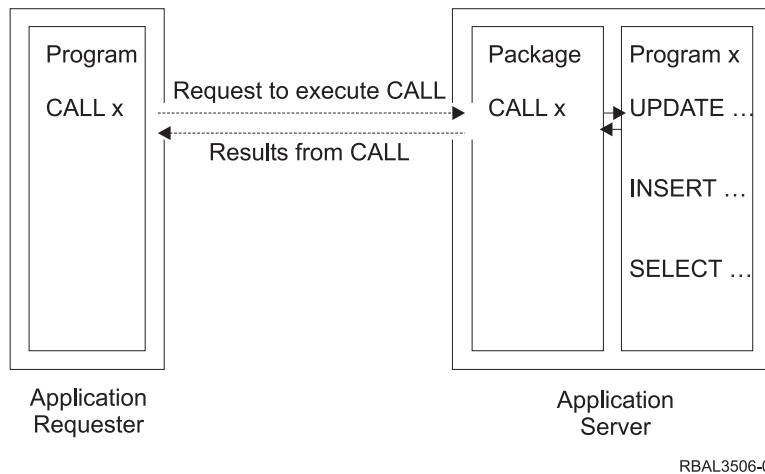


Figure 2. Application With Remote Procedure

Catalog

The database manager maintains a set of tables containing information about the data in the database. These tables are collectively known as the *catalog*. The *catalog tables* contain information about tables, parameters, procedures, packages, views, indexes, and constraints on the system.

The database manager provides views over the catalog tables. The views provide more consistency with the catalog views of other IBM SQL products and with the catalog views of the ANSI and ISO standard (called Information Schema in the standard). The catalog views in QSYS2 contain information about all tables, packages, views, indexes, and constraints on the system. Additionally, an SQL schema will contain a set of these views that only contains information about tables, packages, views, indexes, and constraints in the schema.

Tables and views in the catalog are like any other database tables and views. If you have authorization, you can use SQL statements to look at data in the catalog views in the same way that you retrieve data from any other table in the system. The database manager ensures that the catalog contains accurate descriptions of the objects in the database at all times.

For more information about catalog tables and views, see Appendix E, “DB2 UDB for iSeries Catalog Views” on page 881.

Application Processes, Concurrency, and Recovery

All SQL programs execute as part of an *application process*. In OS/400, an application process is called a job. In the case of ODBC and JDBC and DRDA, the application process ends when the connection ends even though the job they are using does not end and may be reused. An application process is made up of one or more activation groups. Each activation group involves the execution of one or more programs. Programs run under a non-default activation group or the default activation group. All programs except those created by ILE compilers run under the default activation group.

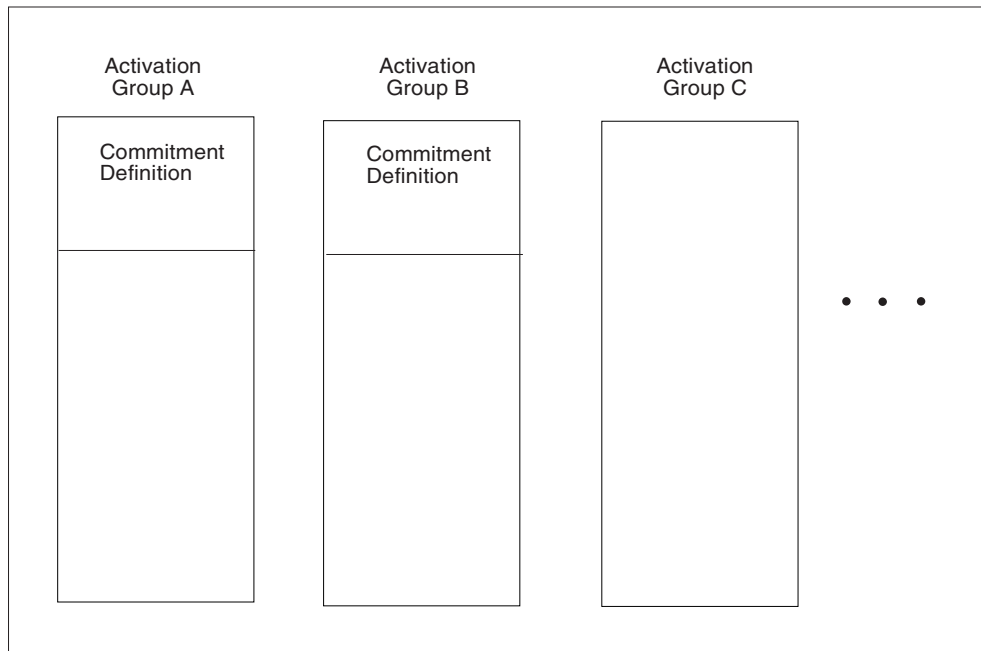
For more information about activation groups, see the book ILE Concepts  .

An application process that uses commitment control can run with one or more commitment definitions. A commitment definition provides a means to scope commitment control at an activation group level or at a job level. At any given time, an activation group that uses commitment control is associated with only one of the commitment definitions.

A commitment definition can be explicitly started through the Start Commitment Control (STRCMTCTL) command. If not already started, a commitment definition is implicitly started when the first SQL statement is executed under an isolation level different than COMMIT(*NONE). More than one activation group can share a job commitment definition.

Figure 3 shows the relationship of an application process, the activation groups in that application process, and the commitment definitions. Activation groups A and B run with commitment control scoped to the activation group. These activation groups have their own commitment definitions. Activation group C does not run with any commitment control and does not have a commitment definition.

Application Process Without Job-Level Commitment Definition



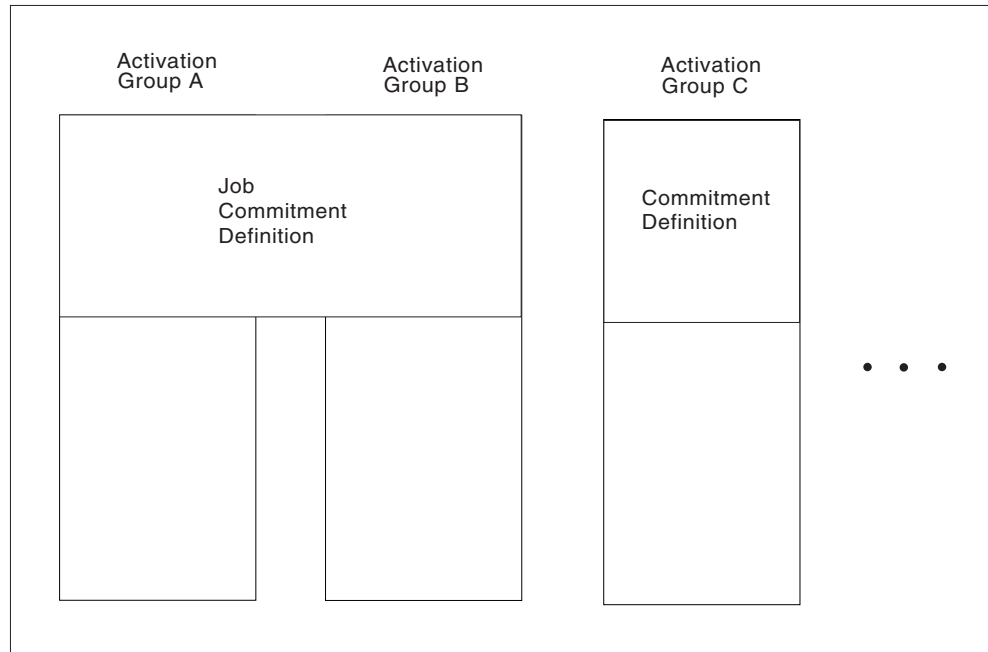
RV3F004-0

Figure 3. Activation Groups without Job Commitment Definition

Figure 4 on page 17 shows an application process, the activation groups in that application process, and the commitment definitions. Some of the activation groups are running with the job commitment definition. Activation groups A and B are running under the job commitment definition. Any commit or rollback operation in activation group A or B affects both because the commitment control is scoped to the same commitment definition. Activation group C in this example has a

separate commitment definition. Commit and rollback operations performed in this activation group only affect operations within C.

Application Process With Job-Level Commitment Control



RV2W931-1

Figure 4. Activation Groups with Job Commitment Definition

For more information about commitment definitions, see the Commitment control topic.

Locking, Commit, and Rollback

Application processes and activation groups that use different commitment definitions can request access to the same data at the same time. Locking is used to maintain data integrity under such conditions. Locking prevents such things as two application processes updating the same row of data simultaneously.

The database manager acquires locks to keep the uncommitted changes of one activation group undetected by activation groups that use a different commitment definition. Object locks and other resources are allocated to an activation group. Row locks are allocated to a commitment definition.

When an activation group other than the default activation group ends *normally*, the database manager releases all locks obtained by the activation group. A user can also explicitly request that most locks be released sooner. This operation is called *commit*. Object locks associated with cursors that remain open after commit are not released.

The recovery functions of the database manager provide a means of backing out of uncommitted changes made in a commitment definition. The database manager may implicitly back out uncommitted changes under the following situations:

- When the application process ends, all changes performed under the commitment definition associated with the default activation group are backed out. When an activation group other than the default activation group ends

| *abnormally*, all changes performed under the commitment definition associated
| with that activation group are backed out.

- When using Distributed Unit of Work and a failure occurs while attempting to commit changes on a remote system, all changes performed under the commitment definition associated with remote connection are backed out.
- When using Distributed Unit of Work and a request to back out is received from a remote system because of a failure at that site, all changes performed under the commitment definition associated with remote connection are backed out.

| A user can also explicitly request that their database changes be backed out. This
| operation is called *rollback*.

| Locks acquired by the database manager on behalf of an activation group are held
| until the unit of work is ended. A lock explicitly acquired by a LOCK TABLE
| statement can be held past the end of a unit of work if COMMIT HOLD or
| ROLLBACK HOLD is used to end the unit of work.

| A cursor can implicitly lock the row at which the cursor is positioned. This lock
| prevents:

- Other cursors associated with a different commitment definition from locking the same row.
- A DELETE or UPDATE statement associated with a different commitment definition from locking the same row.

| **Unit of Work**

| A *unit of work* (also known as a *logical unit of work* or *unit of recovery*) is a
| recoverable sequence of operations. Each commitment definition involves the
| execution of one or more units of work. At any given time, a commitment
| definition has a single unit of work.

| A unit of work is started either when the commitment definition is started, or
| when the previous unit of work is ended by a commit or rollback operation. A unit
| of work is ended by a commit operation, a rollback operation, or the ending of the
| activation group. A commit or rollback operation affects only the database changes
| made within the unit of work that the commit or rollback ends. While changes
| remain uncommitted, other activation groups using different commitment
| definitions running under isolation levels COMMIT(*CS), COMMIT(*RS), and
| COMMIT(*RR) cannot perceive the changes. The changes can be backed out until
| they are committed. Once changes are committed, other activation groups running
| in different commitment definitions can access them, and the changes can no
| longer be backed out.

| The start and end of a unit of work defines points of consistency within an
| activation group. 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 are added to the
| second account is consistency established again. When both steps are complete, the
| commit operation can be used to end the unit of work. After the commit operation,
| the changes are available to activation groups that use different commitment
| definitions.

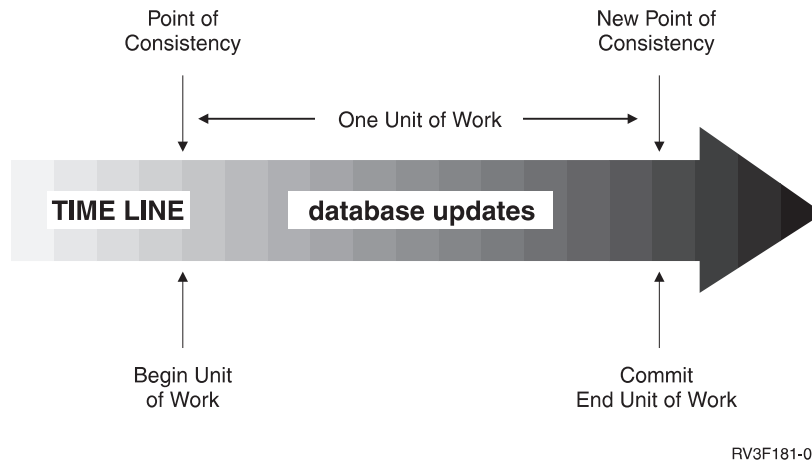


Figure 5. Unit of Work with a Commit Statement

Rolling Back Work

The database manager can back out all changes made in a unit of work or only selected changes. Only backing out all changes results in a point of consistency.

Rolling back all changes

The SQL ROLLBACK statement without the TO SAVEPOINT clause causes a full rollback operation. If such a rollback operation is successfully executed, the database manager backs out uncommitted changes to restore the data consistency that it assumes existed when the unit of work was initiated. That is, the database manager undoes the work, as shown in the diagram below:

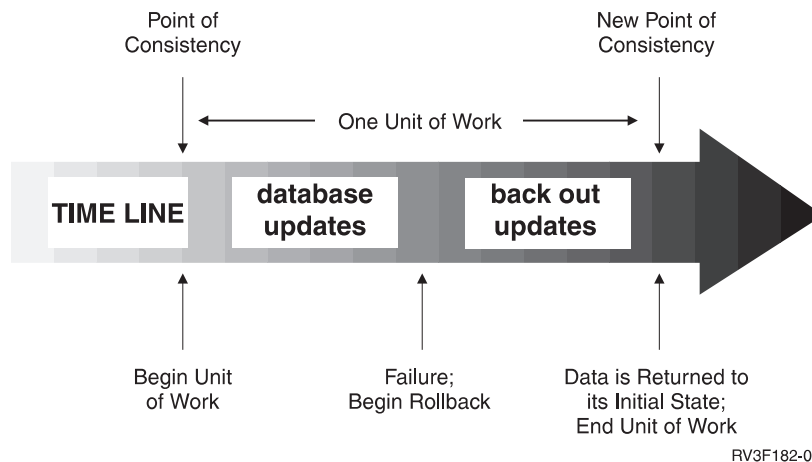


Figure 6. Unit of Work with a Rollback Statement

Rolling back selected changes using savepoints

A *savepoint* represents the state of data at some particular time during a unit of work. An application process can set savepoints within a unit of work, and then as logic dictates, roll back only the changes that were made after a savepoint was set. For example, part of a reservation transaction might involve booking an airline flight and then a hotel room. If a flight gets reserved but a hotel room cannot be reserved, the application process might want to undo the flight reservation without

undoing any database changes made in the transaction prior to making the flight reservation. SQL programs can use the SQL SAVEPOINT statement to set savepoints, the SQL ROLLBACK statement with the TO SAVEPOINT clause to undo changes to a specific savepoint or the last savepoint that was set, and the RELEASE SAVEPOINT statement to delete a savepoint.

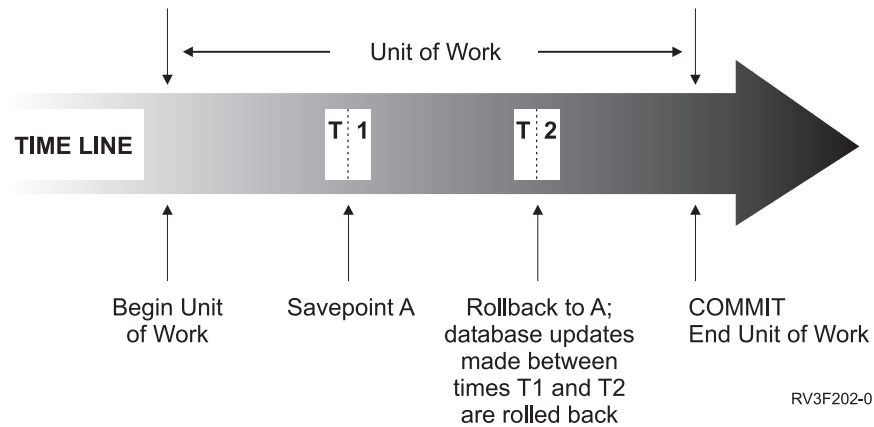


Figure 7. Unit of Work with a Rollback Statement and a Savepoint Statement

Threads

In OS/400, an application process can also consist of one or more threads. By default, a thread shares the same commitment definitions and locks as the other threads in the job. Thus, each thread can operate on the same unit of work so that when one thread commits or rolls back, it can commit or rollback all changes performed by all threads. This type of processing is useful if multiple threads are cooperating to perform a single task in parallel.

In other cases, it is useful for a thread to perform changes independent from other threads in the job. In this case, the thread would not want to share commitment definitions or lock with the other threads. Furthermore, a job can use SQL server mode in order to take more fine grain control of multiple database connections and transaction information. A typical multi-threaded job may require this control. There are several ways to accomplish this type of processing:

- Make sure the programs running in the thread use a separate activation group (be careful not to use ACTGRP(*NEW)).
- Make sure that the job is running in SQL server mode before issuing the first SQL statement. SQL server mode can be activated for a job by using one of the following mechanisms before data access occurs in the application:
 - Use the ODBC API, SQLSetEnvAttr() and set the SQL_ATTR_SERVER_MODE attribute to SQL_TRUE before doing any data access.
 - Use the Change Job API, QWTCHGJB(), and set the 'Server mode for Structured Query Language' key before doing any data access.
 - Use JAVA to access the database via JDBC. JDBC automatically uses server mode to preserve required semantics of JDBC.

When SQL server mode is established, all SQL statements are passed to an independent server job that will handle the requests. Server mode behavior for SQL behavior includes:

- For embedded SQL, each thread in a job implicitly gets one and only one connection to the database (and thus its own committable transaction).

- For ODBC/CLI and JDBC, each connection represents a stand-alone connection to the database and can be committed and used as a separate entity.

For more information, see the SQL Call Level Interface (ODBC) book.

The following SQL support is not threadsafe:

- Remote access through DRDA
- ALTER TABLE
- COMMENT
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DROP
- GRANT
- LABEL
- RENAME
- REVOKE

For more information, see Multithreaded applications in the Programming topic of the iSeries Information Center.

Isolation Level

The *isolation level* used during the execution of SQL statements determines the degree to which the activation group is isolated from concurrently executing activation groups. Thus, when activation group P executes an SQL statement, the isolation level determines:

- The degree to which rows retrieved by P and database changes made by P are available to other concurrently executing activation groups.
- The degree to which database changes made by concurrently executing activation groups can affect P.

The isolation level can be explicitly specified on a DELETE, INSERT, SELECT INTO, UPDATE, or select-statement. If the isolation level is not explicitly specified, the isolation level used when the SQL statement is executed is the *default isolation level*.

DB2 UDB for iSeries provides several ways to specify the *default isolation level*:

- Use the COMMIT parameter on the CRTSQLxxx, STRSQL, and RUNSQLSTM commands to specify the default isolation level.
- Use the SET OPTION statement to specify the default isolation level within the source of a module or program that contains embedded SQL.

- Use the SET TRANSACTION statement to override the default isolation level within a unit of work. When the unit of work ends, the isolation level returns to the value it had at the beginning of the unit of work.
- Use the isolation-clause on the SELECT, SELECT INTO, INSERT, UPDATE, DELETE, and DECLARE CURSOR statements to override the default isolation level for a specific statement or cursor. The isolation level is in effect only for the execution of the statement containing the isolation-clause and has no effect on any pending changes in the current unit of work.

These isolation levels are supported by automatically locking the appropriate data. Depending on the type of lock, this limits or prevents access to the data by concurrent activation groups that use different commitment definitions. Each database manager supports at least two types of locks:

Share Limits concurrent activation groups that use different commitment definitions to read-only operations on the data.

Exclusive

Prevents concurrent activation groups using different commitment definitions from updating or deleting the data. Prevents concurrent activation groups using different commitment definitions that are running COMMIT(*RS), COMMIT(*CS), or COMMIT(*RR) from reading the data. Concurrent activation groups using different commitment definitions that are running COMMIT(*UR) or COMMIT(*NC) are allowed to read the data.

The following descriptions of isolation levels refer to locking data in row units. Individual implementations can lock data in larger physical units than base table rows. However, logically, locking occurs at the base-table row level across all products. Similarly, a database manager can escalate a lock to a higher level. An activation group is guaranteed at least the minimum requested lock level.

DB2 UDB for iSeries supports five isolation levels. For all isolation levels except No Commit, the database manager places exclusive locks on every row that is inserted, updated, or deleted. This ensures that any row changed during a unit of work is not changed by any other activation group that uses a different commitment definition until the unit of work is complete. The isolation levels are:

- Repeatable Read (RR)

Level RR ensures:

- Any row read during a unit of work is not changed by other activation groups that use different commitment definitions until the unit of work is complete.⁴
- Any row changed (or a row that is currently locked with an UPDATE row lock) by another activation group using a different commitment definition cannot be read until it is committed.

In addition to any exclusive locks, an activation group running at level RR acquires at least share locks on all the rows it reads. Furthermore, the locking is performed so that the activation group is completely isolated from the effects of concurrent activation groups that use different commitment definitions.

4. For **WITH HOLD** cursors, these rules apply to when the rows were actually read. For read-only **WITH HOLD** cursors, the rows may have actually been read in a prior unit of work.

DB2 UDB for iSeries supports repeatable-read through COMMIT(*RR). Repeatable-read isolation level is supported by locking the tables containing any rows that are read or updated. In the ANS and ISO standards, Repeatable Read is called Serializable.

- Read Stability (RS)

Like level RR, level RS ensures that:

- Any row read during a unit of work is not changed by other activation groups that use different commitment definitions until the unit of work is complete. ⁴
- Any row changed (or a row that is currently locked with an UPDATE row lock) by another activation group using a different commitment definition cannot be read until it is committed.

Unlike RR, RS does not completely isolate the activation group from the effects of concurrent activation groups that use a different commitment definition. At level RS, activation groups 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. Activation group P1 reads the set of rows *n* that satisfy some search condition.
2. Activation group 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 activation group running at level RS acquires at least share locks on all the rows it reads.

DB2 UDB for iSeries supports read stability through COMMIT(*ALL) or COMMIT(*RS). In the ANS and ISO standards, Read Stability is called Repeatable Read.

- Cursor Stability (CS)

Like levels RR and RS, level CS ensures that any row that was changed (or a row that is currently locked with an UPDATE row lock) by another activation group using a different commitment definition cannot be read until it is committed. Unlike RR and RS, level CS only ensures that the current row of every updateable cursor is not changed by other activation groups using different commitment definitions. Thus, the rows that were read during a unit of work can be changed by other activation groups that use a different commitment definition. In addition to any exclusive locks, an activation group running at level CS may acquire a share lock for the current row of every cursor.

DB2 UDB for iSeries supports cursor stability through COMMIT(*CS). In the ANS and ISO standards, Cursor Stability is called Read Committed.

- Uncommitted Read (UR)

For a SELECT INTO, a FETCH with a read-only cursor, subquery, or subselect used in an INSERT statement, level UR allows:

- Any row read during the unit of work to be changed by other activation groups that run under a different commitment definition.
- Any row changed (or a row that is currently locked with an UPDATE row lock) by another activation group running under a different commitment definition to be read even if the change has not been committed.

For other operations, the rules of level CS apply.

DB2 UDB for iSeries supports uncommitted read through COMMIT(*CHG) or COMMIT(*UR). In the ANS and ISO standards, Uncommitted Read is called Read Uncommitted.

- No Commit (NC)

For all operations, the rules of level UR apply except:

- Commit and rollback operations have no effect on SQL statements. Cursors are not closed, and LOCK TABLE locks are not released. However, connections in the release-pending state are ended.
- Any changes are effectively committed at the end of each successful change operation and can be immediately accessed or changed by other application groups using different commitment definitions.

DB2 UDB for iSeries supports No Commit through COMMIT(*NONE) or COMMIT(*NC).

For a detailed description of record lock durations, see the discussion and table in the commitment control topic of the SQL Programming Concepts book.

Note: (For distributed applications.) When a requested isolation level is not supported by a server, the isolation level is escalated to the next highest supported isolation level. For example, if RS is not supported by a server, the RR isolation level is used.

Comparison of Isolation Levels

The following table summarizes information about isolation levels.

	NC	UR	CS	RS	RR
Can the application see uncommitted changes made by other application processes?	Yes	Yes	No	No	No
Can the application update uncommitted changes made by other application processes?	No	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	Yes	No
Can “updated” rows be updated by other application processes?	Yes	No	No	No	No
Can “updated” rows be read by other application processes that are running at an isolation level other than UR and NC?	Yes	No	No	No	No
Can “updated” rows be read by other application processes that are running at the UR or NC isolation level?	Yes	Yes	Yes	Yes	Yes
Can “accessed” rows be updated by other application processes?	Yes	Yes	Yes	No	No

For RS, “accessed rows” typically means rows selected. For RR, see the product-specific documentation. *See phenomenon P2 (nonrepeatable read) below.*

Can “accessed” rows be read by other application processes?	Yes	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	See Note below	No	No

Note: This depends on whether the cursor that is positioned on the “current” row is updatable:

- If the cursor is updatable, the current row cannot be updated or deleted by other application processes
- If the cursor is not updatable,
 - For UR or NC, the current row can be updated or deleted by other application processes.
 - For CS, the current row may be updatable in some circumstances.

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. 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. UW1 then re-reads the row and obtains the modified data 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. UW1 then repeats the initial read with the same search condition and obtains the original rows plus the inserted rows.
-

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 or logical partitions on the same computer system. Each computer system has a relational database manager that manages the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a 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 a *server* at the other end of the connection.⁵ Working together, the application requester and 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 8 on page 26.

5. This is also known as a *an application server*.

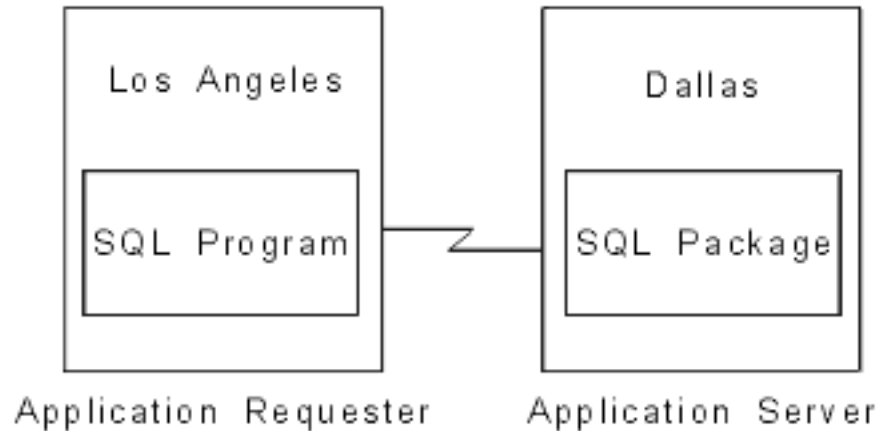


Figure 8. A Distributed Relational Database Environment

For more information about Distributed Relational Database Architecture (DRDA) communication protocols, see Distributed Relational Database Architecture

Reference 

Application Servers

An activation group must be connected to the server of a database manager before SQL statements that reference tables or views can be executed.

A *connection* is an association between an activation group and a local or remote server. Connections are managed by the application. The CONNECT statement can be used to establish a connection to a server and make that server the current server of the activation group.

A server can be local to, or remote from, the environment where the activation group is started. (A server is present, even when distributed relational databases are not used.) This environment includes a local directory that describes the servers that can be identified in a CONNECT statement. For more information about the directory, see the relational database folders in iSeries Navigator or the directory commands (ADDRDBDIRE, CHGRDBDIRE, DSPRDBDIRE, RMVRDBDIRE, and WRKRDBDIRE) in the following iSeries Information Center topics:

- SQL Programming Concepts
- Distributed Database Programming
- CL commands

To execute a static SQL statement that references tables or views, a 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. The appropriate package is determined by the combination of:

- The name of the package specified by the SQLPKG parameter on the CRTSQLxxx commands. See the SQL Programming with Host Languages book for a description of the CRTSQLxxx commands.
- The internal consistency token that makes certain the package and program were created from the same source at the same time.

A DB2 relational database product may support a feature that is not supported by the version of the DB2 UDB product that is connecting to the application server. Some of these features are product-specific, and some are shared by more than one product.

For the most part, an application can use the statements and clauses that are supported by the database manager of the server to which it is currently connected, even though that application is running via the application requester of a database manager that does not support some of those statements and clauses. Restrictions are listed in Appendix C, “Characteristics of SQL Statements” on page 835.

CONNECT (Type 1) and CONNECT (Type 2)

There are two types of CONNECT statements with the same syntax but different semantics:

- CONNECT (Type 1) is used for remote unit of work. See “CONNECT (Type 1)” on page 416.
- CONNECT (Type 2) is used for distributed unit of work. See “CONNECT (Type 2)” on page 421.

See “CONNECT (Type 1) and CONNECT (Type 2) Differences” on page 845 for a summary of the differences.

Remote Unit of Work

The *remote unit of work* facility provides for the remote preparation and execution of SQL statements. An activation group at computer system A can connect to a server at computer system B. Then, within one or more units of work, that activation group can execute any number of static or dynamic SQL statements that reference objects at B. After ending a unit of work at B, the activation group can connect to a 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 server.
- All of the SQL statements in a unit of work must be executed by the same server.

Remote Unit of Work Connection Management

An activation group is in one of three states at any time:

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

The following diagram shows the state transitions:

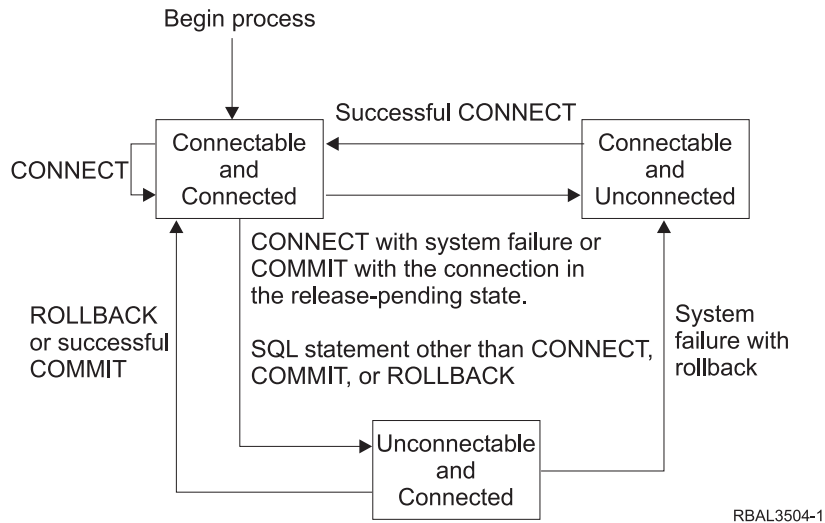


Figure 9. Remote Unit of Work Activation Group Connection State Transition

The initial state of an activation group is *connectable* and *connected*. The server to which the activation group is connected is determined by the RDB parameter on the CRTSQLxxx and STRSQL commands and may involve an implicit CONNECT operation. An implicit CONNECT operation cannot occur if an implicit or explicit CONNECT operation has already successfully or unsuccessfully occurred. Thus, an activation group cannot be implicitly connected to a server more than once.

The connectable and connected state: An activation group is connected to a server and CONNECT statements can be executed. The activation group enters this state when it completes a rollback or successful commit from the unconnectable and connected state, or a CONNECT statement is successfully executed from the connectable and unconnected state.

The unconnectable and connected state: An activation group is connected to a server, but a CONNECT statement cannot be successfully executed to change servers. The activation group enters this state from the connectable and connected state when it executes any SQL statement other than CONNECT, COMMIT, or ROLLBACK.

The connectable and unconnected state: An activation group is not connected to a server. The only SQL statement that can be executed is CONNECT.

The activation group enters this state when:

- The connection was previously released and a successful COMMIT is executed.
- The connection is disconnected using the SQL DISCONNECT statement.
- The connection was in a connectable state, but the CONNECT statement was unsuccessful.

Consecutive CONNECT statements can be executed successfully because CONNECT does not remove the activation group from the connectable state. A CONNECT to the server to which the activation group is currently connected is executed like any other CONNECT statement. CONNECT cannot execute successfully when it is preceded by any SQL statement other than CONNECT, COMMIT, DISCONNECT, SET CONNECTION, RELEASE, or ROLLBACK (unless running with COMMIT(*NC)). To avoid an error, execute a commit or rollback operation before a CONNECT statement is executed.

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. Like remote unit of work, an activation group at computer system A can connect to a server at computer system B and execute any number of static or 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 server. However, unlike remote unit of work, any number of servers can participate in the same unit of work. A commit or rollback operation ends the unit of work.

Distributed unit of work is fully supported for APPC and TCP/IP connections.

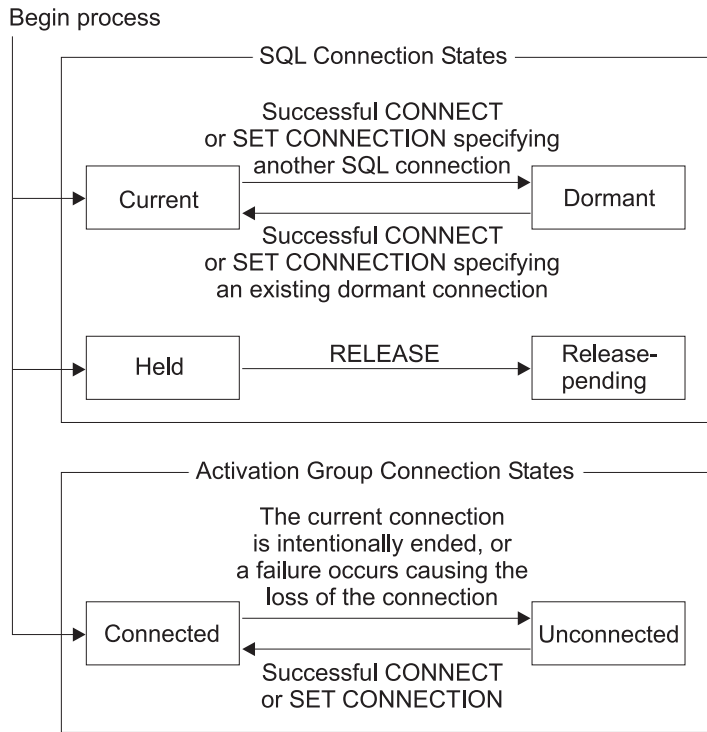
Application-Directed Distributed Unit of Work Connection Management

At any time:

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

Initial state of an activation group: An activation group is initially in the connected state and has exactly one connection. The initial state of a connection is *current and held*.

The following diagram shows the state transitions:



RBAL3503-0

Figure 10. Application-Directed Distributed Unit of Work Connection and Activation Group Connection State Transitions

Connection States

If an application process successfully executes a CONNECT statement:

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

If the server name is already in the set of existing connections of the activation group, an error is returned.

A connection in the dormant state is placed in the current state using the SET CONNECTION statement. When a connection is placed in the current state, the previous current connection, if any, is placed in the dormant state. No more than one connection in the set of existing connections of an activation group can be current at any time. Changing the state of a connection from current to dormant or from dormant to current has no effect on its held or release-pending state.

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

Activation Group Connection States

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

- An activation group cannot have more than one connection to the same server at the same time.
- When an activation group executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections of the activation group.
- When an activation group executes a CONNECT statement, the specified server name must not be an existing connection in the set of connections of the activation group.

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

An activation group in the unconnected state enters the connected state when it successfully executes a CONNECT or SET CONNECTION statement.

If an activation group does not have a current connection, the activation group is in the *unconnected* state. The CURRENT SERVER special register contents are equal to blanks. The only SQL statements that can be executed are CONNECT, DISCONNECT, SET CONNECTION, RELEASE, COMMIT, and ROLLBACK.

An activation group 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 current server and loss of the connection. Connections are intentionally ended when an activation group successfully executes a commit operation and the connection is in the release-pending state, or when an application process successfully executes the DISCONNECT statement.

When a Connection is Ended

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

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

Data Representation Considerations

Different systems represent data in different ways. When data is moved from one system to another, data conversion must sometimes 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 and the sending system's environment type. For example, when a floating-point variable from a DB2 UDB for iSeries application requester is assigned to a column of a table at an OS/390 server, the number is converted from IEEE format to System/370* format.

With character and graphic data, the data type and the environment type of the sending system are not sufficient. Additional information is needed to convert character and graphic strings. String conversion depends on both the coded character set of the data and the operation to be done with that data. String conversions are done in accordance with the IBM Character Data Representation Architecture (CDRA). For more information about character conversion, refer to the book *Character Data Representation Architecture Level 1 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*.⁶

Character conversion can occur when an SQL statement is executed remotely. Consider, for example, these two cases:

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

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

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

character set	A defined set of characters. For example, the following character set appears in several code pages: <ul style="list-style-type: none">• 26 non-accented letters A through Z• 26 non-accented letters a through z• digits 0 through 9• . , ; ? () ' " / - _ (underscore) & + % * = < >
code page	A set of assignments of characters to code points. In EBCDIC, for example, "A" is assigned code point X'C1' and "B" is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.
code point	A unique bit pattern that represents a character.
coded character set	A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.
encoding scheme	A set of rules used to represent character data. For example: <ul style="list-style-type: none">• Single-byte EBCDIC

6. 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. Thus, for many readers, character conversion may be irrelevant.

- Single-byte ASCII⁷
- Mixed single- and double-byte EBCDIC
- Mixed single- and double-byte ASCII
- UCS-2 (universal coded character set).

substitution character

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

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)

	0	1	2	3	4	5	...	E	F
0				0	1	P	...	À	
1				1	A	Q	...	Á	á
2			"	2	B	R	...	Â	â
3				3	C	S	...	Ã	ã
4				4	D	T	...	Ä	ä
5			‰	5	E	U	...	Å	å
...
E			,	>	N		...	‰	‰
F			/	*	O		...	®	

Code Point: 2F

Character-Set ss1
(in code-page pp1)

Code-Page: pp2 (EBCDIC)

	0	1	...	A	B	C	D	E	F
0			...	*					0
1			...	8	A	J			1
2			...	9	%	B	K	8	2
3			...	t	—	C	L	T	3
4			...	u	^	D	M	U	4
5			...	v	{	E	N	V	5
...
E			...		:	A	}		
F			...	A	9	:	A		

Character-Set ss1
(in code-page pp2)

Even with the same encoding scheme there are many different coded character sets, and the same code point can represent a different character in different coded character sets. Furthermore, a byte in a character string does not necessarily represent a character from a single-byte character set (SBCS). Character strings are also used for mixed data (a mixture of single-byte characters and double-byte characters) and for data that is not associated with any character set (called bit data). This is not the case with graphic strings; the database manager assumes that every pair of bytes in every graphic string represents a character from a double-byte character set (DBCS) or universal coded character set (UCS-2).

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

A CCSID in a native encoding scheme is one of the coded character sets in which data may be stored at that site. A CCSID in a foreign encoding scheme is one of the coded character sets in which data cannot be stored at that site. For example, DB2 UDB for iSeries can store data in a CCSID with an EBCDIC encoding scheme, but not in an ASCII encoding scheme.

A host variable containing data in a foreign encoding scheme is always converted to a CCSID in the native encoding scheme when the host variable is used in a function or in the select-list. A host variable containing data in a foreign encoding scheme is also effectively converted to a CCSID in the native encoding scheme when used in comparison or in an operation that combines strings. Which CCSID in the native encoding scheme the data is converted to is based on the foreign CCSID and the default CCSID.

Coded Character Sets and CCSIDs

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

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

In each database manager, character conversion involves the use of a *CCSID Conversion Selection Table*. The Conversion Selection Table contains a list of valid source and target combinations. For each pair of CCSIDs, the Conversion Selection Table contains information used to perform the conversion from one coded character set to the other. This information includes an indication of whether conversion is required. (In some cases, no conversion is necessary even though the strings involved have different CCSIDs.)

Default CCSID

Every server and application requester has a default CCSID (or default CCSIDs in installations that support DBCS data). The CCSID of the following types of strings is determined at the current server:

- String constants (including string constants that represent datetime values) when the CCSID of the source is in a foreign encoding scheme
- Special registers with string values (such as USER and CURRENT SERVER)
- CAST specifications where the result is a character or graphic string
- Results of CHAR, DIGITS, and HEX scalar functions⁸
- Results of VARCHAR, GRAPHIC, and VARGRAPHIC scalar functions when a CCSID is not specified as an argument
- Results of the CLOB and DBCLOB scalar functions when a CCSID is not specified as an argument
- String columns defined by the CREATE TABLE or ALTER TABLE statements when an explicit CCSID is not specified for the column⁹

8. If the default CCSID is 65535, and the function is a CAST to a CLOB or DBCLOB, the CCSID used will be the value of the DFTCCSID job attribute.

9. If the default CCSID is 65535, the character string columns will not use 65535. Instead, the CCSID used will be the value of the DFTCCSID job attribute.

In a distributed SQL program, the default CCSID of host variables is determined by the application requester. In a non-distributed SQL program, the default CCSID of host variables is determined by the server. On OS/400, the default CCSID is determined by the CCSID job attribute. For more information about CCSIDs, see the Work with CCSIDs topic in the Globalization section of the iSeries Information Center.

Sort Sequence

A sort sequence defines how characters in a character set relate to each other when they are compared and ordered. Different sort sequences are useful for those who want their data ordered for a specific language. For example, lists can be ordered as they are normally seen for a specific language. A sort sequence can also be used to treat certain characters as equivalent, for instance, **a** and **A**. A sort sequence works on all comparisons that involve:

- SBCS character data (including bit data)
- the SBCS portion of mixed data
- UCS-2 graphic data.

SBCS sort sequence support is implemented using a 256-byte table. Each byte in the table corresponds to a code point or character in a SBCS code page. Because the sort sequence is applicable to character data, a CCSID must be associated with the table. The bytes in the sort sequence table are set based on how each code point is to compare to other code points in that code page. For example, if the characters **a** and **A** are to be treated as equivalents for comparisons, the bytes in the sort sequence table for their code points contain the same value, or weight.

UCS-2 sort sequence support is implemented using a multi-byte table. A pair of bytes within the table corresponds to a character in the UCS-2 code page. Only a subset of the thousands of characters in UCS-2 are typically represented in the table. Only those characters that are to compare differently (and possibly other characters in the same ward) will be represented in the table. The bytes in the sort sequence table are set based on how each character is to compare with other characters in UCS-2.

When two or more bytes (or pair of bytes for UCS-2) in a sort sequence table have the same value, the sort sequence is a shared-weight sort sequence. If every byte (or pair of bytes for UCS-2) in a sort sequence table has a unique value, the sort sequence is a unique-weight sort sequence. For many languages, unique- and shared-weight sort sequences are shipped on the system as part of the operating system. If you need sort sequences for other languages or needs, you define them using the Create Table (CRTTBL) command.

It is important to remember that the data itself is not altered by the sort sequence. A weighted representation of the data is used for the comparison. In SQL, a sort sequence is specified on the CRTSQLxxx, STRSQL, and RUNSQLSTM commands. The SET OPTION statement can be used to specify the sort sequence within the source of a program containing embedded SQL. The sort sequence applies to all character comparisons performed in the SQL statements. The default sort sequence on the system is the internal sequence that occurs when the hexadecimal representation of characters are used. This is the sequence you get when the SRTSEQ(*HEX) is specified. For programs precompiled with a release of the product that is earlier than Version 2 Release 3, the sort sequence is *HEX.

Sort sequences do not apply to FOR BIT DATA or BLOB columns.

For more information about CCSIDs, see the Work with CCSIDs topic in the Globalization section of the iSeries Information Center. For more information about sort sequences and the sequences shipped with the system, see the Sort Sequence tables topic in the iSeries Information Center.

Authorization, Privileges and Object Ownership

Users can successfully execute SQL statements only if they have the authority to perform the specified function. 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 forth.

There are two forms of authorization:

administrative authority

The person or persons holding *administrative authority* are charged with the task of controlling the database manager and are responsible for the safety and integrity of the data. Those with *administrative authority* implicitly have all privileges on all objects and control who will have access to the database manager and the extent of this access.

The security officer and all users with *ALLOBJ authority have administrative authority.

privileges

Privileges are those activities that a user is allowed to perform. Authorized users can create objects, have access to objects they own, and can pass on *privileges* on their own objects to other users by using the GRANT statement.

Privileges may be granted to specific users or to PUBLIC. PUBLIC specifies that a privilege is granted to a set of users (authorization IDs). The set consists of those users (including future users) that do not have privately granted privileges on the table or view. This affects private grants. For example, if SELECT has been granted to PUBLIC, and UPDATE is then granted to HERNANDZ, this private grant prevents HERNANDZ from having the SELECT privilege.

The REVOKE statement can be used to REVOKE previously granted *privileges*. A revoke of a privilege from an authorization ID revokes the privilege granted by all authorization IDs. Revoking a privilege from an authorization ID will not revoke that same privilege from any other authorization IDs that were granted the privilege by that authorization ID.

When an object is created, the authorization ID of the statement must have the privilege to create objects in the implicitly or explicitly specified schema. The authorization ID of a statement has the privilege to create objects in the schema if:

- it is the owner of the schema, or
- it has *EXECUTE and *ADD to the schema.

When an object is created, one authorization ID is assigned *ownership* of the object. Ownership gives the user complete control over the object, including the privilege to drop the object. The privileges on the object can be granted by the owner, and can be revoked from the owner. In this case, the owner may temporarily be unable to perform an operation that requires that privilege. Because he is the owner, however, he is always allowed to grant the privilege back to himself.

When an object is created:

- If SQL names were specified, the *owner* of the object is the user profile with the same name as the schema into which the alias is created, if a user profile with that name exists. Otherwise, the *owner* of the object is the user profile or group user profile of the job executing the statement.
- If system names were specified, the *owner* of the object is the user profile or group user profile of the job executing the statement.

Authority granted to *PUBLIC on SQL objects depends on the naming convention that is used at the time of object creation. If *SYS naming convention is used, *PUBLIC acquires the create authority (CRTAUT) of the library into which the object was created. If *SQL naming convention is used, *PUBLIC acquires *EXCLUDE authority.

In the Authorization sections of this book, it is assumed that the owner of an object has not had any privileges revoked from that object since it was initially created. If the object is a view, it is also assumed that the owner of the view has not had the system authority *READ revoked from any of the tables or views that this view is directly or indirectly dependent on. The owner has system authority *READ for all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth. For more information

about authority and privileges, see the book *iSeries Security Reference* .

Storage Structures

The iSeries system is an object-based system. All database objects in DB2 UDB for iSeries (tables and indexes for example) are objects in OS/400. The single-level storage manager manages all storage of objects of the database, so database specific storage structures (for example, table spaces) are unnecessary.

A partitioned or distributed table allows data to be spread across different database partitions. The partitions included are determined by the nodegroup specified when the table is created or altered. A nodegroup is a group of one or more iSeries systems. A partitioning map is associated with each nodegroup. The partitioning map is used by the database manager to determine which system from the nodegroup will store a given row of data. For more information about nodegroups and data partitioning see the DB2 Multisystem book.

A table can also include columns that register links to data that are 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 that is stored in an external file server.

The DB2 File Manager on a file server works in conjunction with DB2 to provide the following optional functionality:

- Referential integrity to ensure that files currently linked to DB2 are not deleted or renamed.
- Security to ensure that only those with suitable SQL privileges on the DataLink column can read the files linked to that column.

The DataLinker 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 ensure that registered files are not deleted or renamed. Optionally, filters commands to ensure that proper access authority exists.

Chapter 2. Language Elements

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

For details, see the following sections:

- “Characters”
- “Tokens” on page 41
- “Identifiers” on page 43
- “Naming Conventions” on page 45
- “SQL Path” on page 53
- “Aliases” on page 56
- “Authorization IDs and Authorization-Names” on page 57
- “Data Types” on page 59
- “Promotion of Data Types” on page 75
- “Casting Between Data Types” on page 77
- “Assignments and Comparisons” on page 80
- “Rules for Result Data Types” on page 93
- “Conversion Rules for Operations That Combine Strings” on page 97
- “Constants” on page 99
- “Special Registers” on page 105
- “References to Variables” on page 114
- “Host Structures” on page 120
- “Host Structure Arrays” on page 122
- “Functions” on page 123
- “Expressions” on page 129
- “Predicates” on page 147

Characters

The basic symbols of keywords and operators in the SQL language are single-byte characters¹⁰ that are part of all character sets supported by the IBM relational database products. 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 of the English alphabet.¹¹

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

A *special character* is any of the characters listed below:¹²

10. Note that if the SQL statement is encoded as UCS-2 data, all characters of the statement except for string constants will be converted to single-byte characters prior to processing. Tokens representing string constants may be processed as UCS-2 graphic strings without conversion to single-byte.

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

Characters

	space	-	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
¹³	vertical bar		

12. The not symbol (¬) and the exclamation point symbol (!) are also special characters used by DB2 UDB for iSeries. Avoid using them because they are variant characters.

13. Using the vertical bar (|) character might inhibit code portability between IBM relational database products. It is preferable to use the CONCAT operator instead of the concatenation operator (||). Use of the vertical bar should be avoided because it is a variant character.

Tokens

The basic syntactical units of the language are called *tokens*. A token consists of one or more characters, excluding blanks, control characters, and characters within a string constant or delimited identifier. (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.
- 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 691.

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

Control Characters: A *control character* is a special character that is used for string alignment. The following table contains the control characters that are handled by the database manager:

Table 1. Control Characters

Control Character	EBCDIC Hex Value	UCS-2 Hex Value
Tab	05	0009
Form Feed	0C	000C
Carriage Return	0D	000D
New Line	15	0085
Line Feed (New line)	25	000A

Tokens, other than string constants and certain delimited identifiers, must not include a control character or space. A control character or space can follow a token. A delimiter token, a control character, or a space *must* follow every ordinary token. If the syntax does not allow a delimiter token to follow an ordinary token, then a control character or a space must follow that ordinary token. The following examples illustrate the rule that is stated in this paragraph.

Here are some examples of ordinary tokens:

1 .1 +2 SELECT E 3

Here are some examples of combinations of the above ordinary tokens that, in effect, change the tokens:

1.1 .1+2 SELECTE .1E E3 SELECT1

This demonstrates why ordinary tokens must be followed by a delimiter token or a space.

Here are some examples of delimiter tokens:

, 'string' "fld1" = .

Here are some examples of combinations of the above ordinary tokens and the above delimiter tokens that, in effect, change the tokens:

1. .3

Tokens

The period (.) is a delimiter token when it is used as a separator in the qualification of names. Here the dot is used in combination with an ordinary token of a numeric constant. Thus, the syntax does not allow an ordinary token to be followed by a delimiter token. Instead, the ordinary token must be followed by a space.

If the decimal point has been defined to be the comma, as described in “Decimal Point” on page 102, the comma is interpreted as a decimal point in numeric constants. Here are some examples of these numeric constants:

1,2 ,1 1, 1,e1

If '1,2' and '1,e1' are meant to be two items, both the ordinary token (1) and the delimiter token (,) must be followed by a space, to prevent the comma from being interpreted as a decimal point. Although the comma is usually a delimiter token, the comma is part of the number when it is interpreted as a decimal point. Therefore, the syntax does not allow an ordinary token (1) to be followed by a delimiter token (,). Instead, an ordinary token must be followed by a space.

Comments: Static SQL statements can include host language comments or SQL comments. Dynamic SQL statements can include SQL comments. Either type of comment can be specified wherever a space may be specified, except within a delimiter token or between the keywords EXEC and SQL. There are two types of SQL comments:

simple comments

Simple comments are introduced by two consecutive hyphens (--). Simple comments cannot continue past the end of the line. For more information, see “SQL Comments” on page 368.

bracketed comments

Bracketed comments are introduced by /* and end with */. A bracketed comment can continue past the end of the line. For more information, see “SQL Comments” on page 368.

Uppercase and Lowercase: Lowercase letters used in an ordinary token other than a C host variable will be folded to uppercase. Delimiter tokens are never folded to uppercase. Thus, the statement:

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

is equivalent, after folding, to:

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

Identifiers

An *identifier* is a token used to form a name. An identifier in an SQL statement is one of the following types:

- “SQL Identifiers”
- “System identifiers”
- “Host Identifiers” on page 44

Note: \$, @, #, and all other variant characters should not be used in identifiers because the code points used to represent them vary depending on the CCSID of the string in which they are contained. If they are used, unpredictable results may occur. For more information about variant characters, see the Variant characters topic in the iSeries Information Center.

SQL Identifiers

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

- An *ordinary identifier* is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. Note that ordinary identifiers are converted to uppercase. An ordinary identifier should not be a reserved word. See Appendix G, “Reserved Words” on page 1009 for a list of reserved words. If a reserved word is used as an identifier in SQL, it should be specified in uppercase and enclosed within the SQL escape characters.
- A *delimited identifier* is a sequence of one or more characters enclosed within SQL escape characters. The sequence must consist of one or more characters. Leading blanks in the sequence are significant. Trailing blanks in the sequence are not significant. The length of a delimited identifier does not include the two SQL escape characters. Note that delimited identifiers are not converted to uppercase. The escape character is the quotation mark (") except in the following cases where the escape character is the apostrophe ('):
 - Interactive SQL when the SQL string delimiter is set to the quotation mark in COBOL syntax checking statement mode
 - Dynamic SQL in a COBOL program when the CRTSQLCBL or CRTSQLCBLI parameter OPTION(*QUOTESQL) specifies that the string delimiter is the quotation mark (")
 - COBOL application program when the CRTSQLCBL or CRTSQLCBLI parameter OPTION(*QUOTESQL) specifies that the string delimiter is the quotation mark (")

The following characters are not allowed within delimited identifiers:

- X'00' through X'3F' and X'FF'

System identifiers

A system identifier is used to form the name of system objects in OS/400. There are two types of system identifiers: ordinary identifiers and delimited identifiers.

- The rules for forming a system ordinary identifier are identical to the rules for forming an SQL ordinary identifier.
- The rules for forming a system delimited identifier are identical to those for forming SQL delimited identifiers, except:
 - The following special characters are not allowed in a delimited system identifier:
 - A blank (X'40')

Identifiers

- An asterisk (X'5C')
 - An apostrophe (X'7D')
 - A question mark (X'6F')
 - A quotation mark (X'7F')
- The bytes required for the escape characters are included in the length of the identifier unless the characters within the delimiters would form an ordinary identifier.

For example, "PRIVILEGES" is in uppercase and the characters within the delimiters form an ordinary identifier; therefore, it has a length of 10 bytes and is a valid system name for a column. On the other hand, "privileges" is in lowercase, has a length of 12 bytes, and is not a valid system name for a column because the bytes required for the delimiters must be included in the length of the identifier.

Examples

WKLYSAL WKLY_SAL "WKLY_SAL" "UNION" "wkly_sal"

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; except that DBCS characters cannot be used. For example, the rules for forming a host-identifier in a COBOL program are the same as the rules for forming a user-defined word in COBOL. Names beginning with the characters 'SQ'¹⁴, 'SQL', 'sql', 'RDI', or 'DSN' should not be used because precompilers generate host variables that begin with these characters.

14. 'SQ' is allowed in C, COBOL, and PL/I; it should not be used in RPG.

Naming Conventions

The rules for forming a name depend on the type of the object designated by the name and the naming option (*SQL or *SYS). The naming option is specified on the CRTSQL_{xxx}, RUNSQLSTM, and STRSQL commands. The SET OPTION statement can be used to specify the naming option within the source of a program containing embedded SQL. The syntax diagrams use different terms for different types of names. The following list defines these terms.

alias-name

A qualified or unqualified name that designates an alias. The qualified form of an alias-name depends on the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.

The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.

An alias-name can specify either the name of the alias or the system object name of the alias.

authorization-name

A system identifier that designates a user or group of users. An authorization name is a user profile name on the server. It must not be a delimited identifier that includes lowercase letters or special characters. See “Authorization IDs and Authorization-Names” on page 57 for the distinction between an authorization name and an authorization ID.

column-name

A qualified or unqualified name that designates a column of a table or a view. The unqualified form of a column name is an SQL identifier. The qualified form is a qualifier followed by a period and an SQL identifier. The qualifier is a table name, a view name, or a correlation name.

Column names cannot be qualified with system names in the form *schema-name/table-name.column-name*, except in the COMMENT and LABEL statements. If column names need to be qualified, and correlation names are allowed in the statement, a correlation name must be used to qualify the column.

A column-name can specify either the column name or the system column name of a column of a table or view. If a column name is delimited, the delimiters are considered to be part of the name when determining the length of the name.

constraint-name

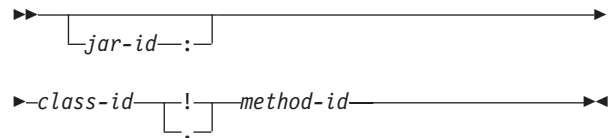
A qualified or unqualified name that designates a constraint on a table. The qualified form of a constraint name depends on the naming option. For SQL naming, the qualified form is a

Naming Conventions

	<p>schema-name followed by a period (.) and a system identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.</p> <p>The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of Unqualified Object Names" on page 53.</p>
correlation-name	<p>An SQL identifier that designates a table, a view, or individual rows of a table or view.</p>
cursor-name	<p>An SQL identifier that designates an SQL cursor.</p>
descriptor-name	<p>A colon followed by a host-identifier that designates an SQL descriptor area (SQLDA). See "References to Host Variables" on page 114 for a description of a host identifier. A host variable that designates an SQL descriptor area must not have an indicator variable. The form <i>:host-variable:indicator-variable</i> is not allowed.</p>
distinct-type-name	<p>A qualified or unqualified name that designates a distinct type. The qualified form of a distinct-type-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.</p> <p>The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of Unqualified Object Names" on page 53.</p> <p>For system naming, distinct type names cannot be qualified when used in a parameter data type of an SQL routine or in an SQL variable declaration in an SQL function, SQL procedure, or trigger.</p>
external-program-name	<p>A qualified name, unqualified name, or a character string that designates an external program. The qualified form of an external-program-name depends on the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and a system identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by a system identifier.</p> <p>The unqualified form is a system identifier. The unqualified form is implicitly qualified based on the rules specified in "Qualification of Unqualified Object Names" on page 53.</p> <p>The format of the character string form is either:</p> <ul style="list-style-type: none">• An OS/400 qualified program name ('library-name/program-name').

Naming Conventions

- An OS/400 qualified source file name, followed by a left parenthesis, followed by an OS/400 member name, and a right parenthesis ('library-name/source-file-name(member-name)'). This form is only valid when calling a REXX procedure.
- An OS/400 qualified service program name, followed by a left parenthesis, followed by an OS/400 entry-point-name, followed by a right parenthesis ('library-name/service-program-name(entry-point-name)'). This form is only valid for functions.
- An optional jar-id, followed by a class identifier, followed by an exclamation point or period, followed by a method identifier ('class-id!method-id' or 'class-id.method-id').



The jar-id identifies the jar schema when it was installed in the database. It can be either a simple identifier, or a schema qualified identifier. Examples are 'myJar' and 'myCollection.myJar'. The class-id identifies the class identifier of the Java object. If the class is part of a package, the class identifier must include the complete package prefix. For example, if the class identifier is 'myPackage.StoredProcs', the Java Virtual machine will look in the following directory for the StoredProcs class:

```
'/QIBM/UserData/OS400/SQLLib/  
Function/myPackage/StoredProcs/'
```

The method-id identifies the method name of the Java object to be invoked.

This form is only valid for Java procedures and Java functions.

function-name

A qualified or unqualified name that designates a user-defined function, a cast function that was generated when a distinct type was created, or a built-in function. The qualified form of a function-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.

The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on

Naming Conventions

the rules specified in “Qualification of Unqualified Object Names” on page 53.

For system naming, functions names can only be qualified in the form schema-name/function-name when the name is used in a CREATE, COMMENT, DROP, GRANT, or REVOKE statement.

host-label	A token that designates a label in a host program.
host-variable	A sequence of tokens that designates a host variable. A host-variable includes at least one host-identifier, as explained in “References to Host Variables” on page 114.
index-name	<p>A qualified or unqualified name that designates an index. The qualified form of an index-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.</p> <p>The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.</p>
nodegroup-name	<p>A qualified or unqualified name that designates a nodegroup. A nodegroup is a group of iSeries servers across which a table will be distributed. For more information about distributed tables and nodegroups, see the DB2 Multisystem book.</p> <p>The qualified form of a nodegroup-name depends on the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and a system identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by a system identifier.</p> <p>The unqualified form is a system identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.</p>
package-name	<p>A qualified or unqualified name that designates a package. The qualified form of a package-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and a system identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by a system identifier.</p> <p>The unqualified form is a system identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.</p>

Naming Conventions

parameter-name	An SQL identifier that designates a parameter for a function or procedure. If the parameter is for a procedure, the identifier may be preceded by a colon.
procedure-name	<p>A qualified or unqualified name that designates a procedure. The qualified form of a procedure-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.</p> <p>The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.</p>
savepoint-name	An unqualified identifier that designates a savepoint.
schema-name	<p>A qualified or unqualified name that provides a logical grouping for SQL objects. A schema name is used as a qualifier of the name of a table, view, index, procedure, function, trigger, constraint, alias, type, or package. The unqualified form of a schema-name is a system identifier. The qualified form of a schema-name depends on the naming option.</p> <p>For SQL names, the unqualified schema name in an SQL statement is implicitly qualified by the server-name. The qualified form is a server-name followed by a (.) and a system identifier. The server-name must identify the current server.</p> <p>For system names, the unqualified schema name in an SQL statement is implicitly qualified by the server-name. The qualified form is a server-name followed by a slash (/) and a system identifier. The server-name must identify the current server.</p> <p>Note: Schema-name refers to either a schema created by the CREATE SCHEMA statement or to an OS/400 library.</p>
server-name	An SQL identifier that designates a server. The identifier must not include lowercase letters or special characters.
specific-name	A qualified or unqualified name that uniquely identifies a procedure or function. The qualified form of a specific-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.

Naming Conventions

	<p>The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.</p>
SQL-label	<p>An unqualified name that designates a label in an SQL procedure, SQL function, or trigger body. An SQL label name is an SQL identifier.</p>
SQL-parameter-name	<p>A qualified or unqualified name that designates a parameter in an SQL routine body. The unqualified form of an SQL parameter name is an SQL identifier. The qualified form is a procedure-name followed by a period (.) and an SQL identifier.</p>
SQL-variable-name	<p>A qualified or unqualified name that designates a variable in an SQL routine body. The unqualified form of an SQL variable name is an SQL identifier. The qualified form is an SQL label followed by a period (.) and an SQL identifier.</p>
statement-name	<p>An SQL identifier that designates a prepared SQL statement.</p>
system-column-name	<p>An unqualified name that designates the OS/400 column name of a table or a view. A system-column-name is a system identifier. System-column-names can be delimited identifiers, but the characters within the delimiters must not include lowercase letters or special characters.</p>
system-object-name	<p>An unqualified name that designates the OS/400 name of a table, view, index, or alias. A system-object-name is a system identifier.</p> <p>If the unqualified name of the table, view, index, or alias is a valid system identifier, the system-object-name of the table, view, index, or alias is the unqualified name of the table, view, index, or alias.</p>
table-name	<p>A qualified or unqualified name that designates a table. The qualified form of a table-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.</p> <p>The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.</p> <p>A table-name can specify either the name of the table or the system object name of the table.</p>
trigger-name	<p>A qualified or unqualified name that designates a trigger on a table. The qualified form of a trigger name depends on the naming option. For SQL</p>

Naming Conventions

naming, the qualified form is a schema-name followed by a period (.) and a system identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.

The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.

view-name

A qualified or unqualified name that designates a view. The qualified form of a view-name depends upon the naming option. For SQL naming, the qualified form is a schema-name followed by a period (.) and an SQL identifier. For system naming, the qualified form is a schema-name followed by a slash (/) followed by an SQL identifier.

The unqualified form is an SQL identifier. The unqualified form is implicitly qualified based on the rules specified in “Qualification of Unqualified Object Names” on page 53.

A view-name can specify either the name of the view or the system object name of the view.

Table 2. Identifier Length Limits (in bytes)

Identifier Type	Maximum Length
Longest authorization name	10
Longest condition name	128
Longest correlation name	128
Longest cursor name	18
Longest external program name (unqualified form) ¹⁵	10
Longest external program name (string form)	279
Longest host identifier	64
Longest savepoint name	128
Longest schema name	10
Longest server name	18
Longest SQL label	128
Longest statement name	18
Longest unqualified alias name	128
Longest unqualified column name	30
Longest unqualified constraint name	128
Longest unqualified distinct type name	128
Longest unqualified function name	128
Longest unqualified index name	128
Longest unqualified nodegroup name	10
Longest unqualified package name	10
Longest unqualified parameter name	128

Naming Conventions

Table 2. Identifier Length Limits (in bytes) (continued)

Identifier Type	Maximum Length
Longest unqualified procedure name	128
Longest unqualified specific name	128
Longest unqualified SQL parameter name	128
Longest unqualified SQL variable name	128
Longest unqualified system column name	10
Longest unqualified system object name	10
Longest unqualified table and view name	128
Longest unqualified trigger name	128

15. For REXX procedures, the limit is 33.

SQL Path

The *SQL path* is an ordered list of schema names. The database manager uses the path to resolve the schema name for unqualified distinct type names (both built-in types and distinct types), function names, and procedure names that appear in any context other than as the main object of a CREATE, DROP, COMMENT, GRANT or REVOKE statement. Searching through the path from left to right, the database manager implicitly qualifies the object name with the first schema name in the path that contains the same object with the same unqualified name. For procedures, the database manager selects a matching procedure name only if the number of parameters is also the same. For functions, the database manager uses a process called function resolution in conjunction with the SQL path to determine which function to choose because several functions with the same name can reside in a schema. (For details, see “Function resolution” on page 124.)

For example, if the SQL path is SMITH, XGRAPHIC, QSYS, QSYS2 and an unqualified distinct type name MYTYPE was specified, the database manager looks for MYTYPE first in schema SMITH, then XGRAPHIC, and then QSYS and QSYS2.

The path used is determined as follows:

- For all static SQL statements (except for a CALL *:host-variable* statement), the path used is the path specified in the SQLPATH parameter on the CRTSQLxxx command. The SQLPATH can also be set using the SET OPTION statement.
- For dynamic SQL statements (and for a CALL *:host-variable* statement), the path used is the path specified in the CURRENT PATH special register. For more information about the CURRENT PATH special register, see “CURRENT PATH, CURRENT_PATH, or CURRENT FUNCTION PATH” on page 105.

If the SQL path is not explicitly specified, the SQL path is SYSTEM followed by the authorization ID of the statement.

For more information on the SQL path, see “CURRENT PATH, CURRENT_PATH, or CURRENT FUNCTION PATH” on page 105.

Qualification of Unqualified Object Names

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

Unqualified Alias, Constraint, External Program, Index, Nodegroup, Package, Table, Trigger, and View Names

Unqualified alias, constraint, external program, index, nodegroup, package, table, trigger, and view names are implicitly qualified as follows:

- For static SQL statements:
 - If the DFTRDBCOL parameter is specified on the CRTSQLxxx command (or with the SET OPTION statement), the implicit qualifier is the schema-name that is specified for that parameter.
 - In all other cases, the implicit qualifier is based on the naming convention.
 - For SQL naming, the implicit qualifier is the authorization identifier of the statement.
 - For system naming, the implicit qualifier is the job library list (*LIBL).
- For dynamic SQL statements the implicit qualifier depends on whether or not a default schema name has been explicitly specified. The mechanism for explicitly specifying this depends on the interface used to dynamically prepare and execute SQL statements.

SQL Path

- If a default schema name is not explicitly specified:
 - For SQL naming, the implicit qualifier is the run-time authorization identifier.
 - For system naming, the implicit qualifier is the job library list (*LIBL).
- If a default schema name is explicitly specified, the implicit qualifier is that default schema. The default schema name can be specified through the following interfaces:

Table 3. Default Schema Interfaces

SQL Interface	Specification
Embedded SQL	DFTRDBCOL parameter and DYNDFTCOL(*YES) on the Create SQL Program (CRTSQLxxx) and Create SQL Package (CRTSQLPKG) commands. The SET OPTION statement can also be used to set the DFTRDBCOL and DYNDFTCOL values. (For more information about CRTSQLxxx commands, see the SQL Programming with Host Languages book.)
Run SQL Statements	DFTRDBCOL parameter on the Run SQL Statements (RUNSQLSTM) command. (For more information about the RUNSQLSTM command, see the SQL Programming Concepts book.)
Call Level Interface (CLI) on the server	SQL_ATTR_DEFAULT_LIB or SQL_ATTR_DBC_DEFAULT_LIB environment or connection variables (For more information about CLI, see the SQL Call Level Interfaces (ODBC) book.)
JDBC or SQLJ on the server using Developer Kit for Java	libraries property object (For more information about JDBC and SQLJ, see the IBM Developer Kit for Java topic in the iSeries Information Center.)
ODBC on a client using the iSeries Access ODBC Driver	SQL Default Library in ODBC Setup (For more information about ODBC, see the iSeries Access category in the iSeries Information Center.)
JDBC on a client using the IBM Toolbox for Java	SQL Default Library in JDBC Setup (For more information about JDBC, see the iSeries Access category in the iSeries Information Center.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java topic in the iSeries Information Center .)
All interfaces	SET SCHEMA or QSQCHGDC (Change Dynamic Default Collection) API (For more information about QSQCHGDC, see the File APIs category in the iSeries Information Center.)

Unqualified Function, Procedure, Specific, and Distinct Type Names

The qualification of data type (both built-in types and distinct types), function, procedure, and specific names depends on the SQL statement in which the unqualified name appears:

- If an unqualified name is the main object of a CREATE, COMMENT, DROP, GRANT, or REVOKE statement, the name is implicitly qualified using the same

rules as for qualifying unqualified table names (See “Unqualified Alias, Constraint, External Program, Index, Nodegroup, Package, Table, Trigger, and View Names” on page 53).

- Otherwise, the implicit schema name is determined as follows:
 - For distinct type names, the database manager searches the SQL path and selects the first schema in the path such that the data type exists in the schema.
 - For procedure names, the database manager searches the SQL path and selects the first schema in the path such that the schema contains a procedure with the same name and number of parameters.
 - For function names and for specific names specified for sourced functions, the database manager uses the SQL path in conjunction with function resolution, as described under “Function resolution” on page 124.

SQL Names and System Names: Special Considerations

The CL command Override Database File (OVRDBF) can be specified to override an SQL or system name with another object name for local data manipulation SQL statements. Overrides are ignored for data definition SQL statements and data manipulation SQL statements executing at a remote relational database. See the File Management book for more information about the override function.

Aliases

An *alias* can be thought of as an alternative name for a table, view, or member of a database file. A table or view in an SQL statement can be referenced by its name or by an alias. An alias can only refer to a table, view, or database file member within the same relational database.

An alias can be used wherever a table or view name can be used, except:

- Do not use an alias name 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.
- An alias that refers to an individual member of a database file member can only be used in a select statement, DELETE, INSERT, SELECT INTO, SET variable, UPDATE, or VALUES INTO statement.

Aliases can also help avoid using file overrides. Not only does an alias perform better than an override, but an alias is also a permanent object that only need be created once.

An alias can be created even though the object the alias refers to does not exist. However, the object must exist when a statement that references the alias is executed. A warning is returned if the object does not exist when the alias is created. An alias cannot refer to another alias. An alias can only refer to a table, view, or database file member within the same relational database.

The option of referring to a table, view, or database file member by an alias name is not explicitly shown in the syntax diagrams or mentioned in the description of the SQL statements.

A new alias cannot have the same fully-qualified name as an existing table, view, index, file, 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 before the SQL statement is executed, is replaced by the qualified base table, view, or database file member name. For example, if PBIRD.SALES is an alias for DSPN014.DIST4_SALES_148, then at statement run time:

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

effectively becomes

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

If an alias is dropped and recreated to refer to another table, any SQL statements that refer to that alias will be implicitly rebound when they are next run. If a CREATE VIEW or CREATE INDEX statement refers to an alias, dropping and re-creating the alias has no effect on the view or index.

For syntax toleration of existing DB2 UDB for OS/390 and z/OS 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 ID's are used by the database manager to provide authorization checking of SQL statements.

An authorization ID applies to every SQL statement. The authorization ID that is used for authorization checking for a static SQL statement depends on the USRPRF value specified on the precompiler command:

- If USRPRF(*OWNER) is specified, or if USRPRF(*NAMING) is specified and SQL naming mode is used, the authorization ID of the statement is the owner of the non-distributed SQL program. For distributed SQL programs, it is the owner of the SQL package.
- If USRPRF(*USER) is specified, or if USRPRF(*NAMING) is specified and system naming mode is used, the authorization ID of the statement is the authorization ID of the user running the non-distributed SQL program. For distributed SQL programs, it is the authorization ID of the user at the current server.

The authorization ID that is used for authorization checking for a dynamic SQL statement also depends on where and how the statement is executed:

- If the statement is prepared and executed from a non-distributed program:
 - If the USRPRF value is *USER and the DYNUSRPRF value is *USER for the program, the authorization ID that applies is the ID of the user running the non-distributed program. This is called the *run-time authorization ID*.
 - If the USRPRF value is *OWNER and the DYNUSRPRF value is *USER for the program, the authorization ID that applies is the ID of the user running the non-distributed program.
 - If the USRPRF value is *OWNER and the DYNUSRPRF value is *OWNER for the program, the authorization ID that applies is the ID of the owner of the non-distributed program.
- If the statement is prepared and executed from a distributed program:
 - If the USRPRF value is *USER and the DYNUSRPRF value is *USER for the SQL package, the authorization ID that applies is the ID of the user running the SQL package at the current server. This is also called the run-time authorization ID.
 - If the USRPRF value is *OWNER and the DYNUSRPRF value is *USER for the SQL package, the authorization ID that applies is the ID of the user running the SQL package at the current server.
 - If the USRPRF value is *OWNER and the DYNUSRPRF value is *OWNER for the SQL package, the authorization ID that applies is the ID of the owner of the SQL package at the current server.
- If the statement is issued interactively, the authorization ID that applies is the ID of the user that issued the Start SQL (STRSQL) command.
- If the statement is executed from the RUNSQLSTM command, the authorization ID that applies is the ID of the user that issued the RUNSQLSTM command.
- If the statement is executed from REXX, the authorization ID that applies is the ID of the user that issued the STRREXPRC command.

Authorization IDs and Names

On OS/400, the run-time authorization ID is the user profile of the job.

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 in GRANT and REVOKE statements to designate a target of the grant or revoke. The premise of a grant of privileges to *X* is that *X* will subsequently be the authorization ID of statements which require those privileges. A group user profile can also be used when checking authority for an SQL statement. For information

on group user profiles, see the book *iSeries Security Reference*  .

Example

Assume SMITH is your user ID; then SMITH is the authorization ID when you execute the following statement interactively:

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

SMITH is the authorization ID of the statement. Thus, the authority to execute the statement is checked against SMITH.

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

Data Types

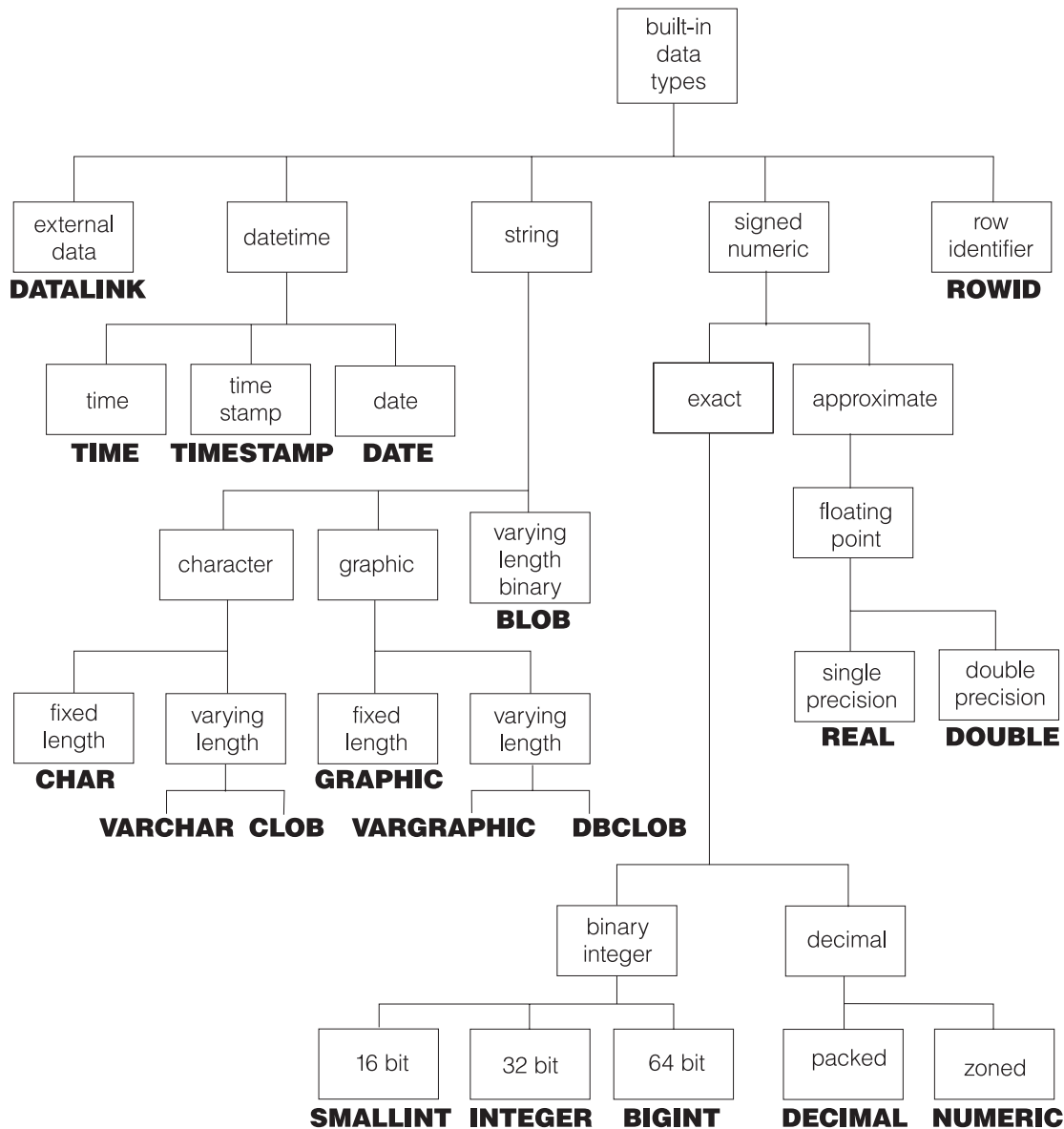
The smallest unit of data that can be manipulated in SQL is called a *value*. How values are interpreted depends on the data type of their source. The sources of values are:

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

The DB2 UDB relational database products support both both built-in data types and user-defined data types. This section describes the built-in data types. For a description of distinct types, see “User-Defined Types” on page 73.

The following figure illustrates the various built-in data types supported by the DB2 UDB for iSeries program.

Data Types



RBAFZ501-1

For more details on data types, see the following topics:

- “Binary Strings” on page 65
- “Character Strings” on page 62
- “Character Encoding Schemes” on page 63
- “Graphic Strings” on page 64
- “Graphic Encoding Schemes” on page 65
- “Large Objects (LOBs)” on page 65
- “Numbers” on page 61
- “Datetime Values” on page 67
- “DataLink Values” on page 72
- “Row ID Values” on page 73
- “User-Defined Types” on page 73

For information about specifying the data types of columns, see “CREATE TABLE” on page 525.

Nulls

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

Numbers

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

Small Integer

A *small integer* is a binary number composed of 2 bytes (16 bits) with a precision of 5 digits. The range of small integers is -32768 to $+32767$.

For small integers, decimal precision and scale are supported by COBOL, RPG, and iSeries system files. For information concerning the precision and scale of binary integers, see the DDS Reference book.

Large Integer

A *large integer* is a binary number composed of 4 bytes (32 bits) with a precision of 10 digits. The range of large integers is -2147483648 to $+2147483647$.

For large integers, decimal precision and scale are supported by COBOL, RPG, and iSeries system files. For information concerning the precision and scale of binary integers, see the DDS Reference book.

Big Integer (BIGINT)

A *big integer* is a binary number composed of 8 bytes (64 bits) with a precision of 19 digits. The range of big integers is -9223372036854775808 to $+9223372036854775807$.

Floating-Point

A *single-precision floating-point* number is a 32-bit approximate representation of a real number. The range of magnitude is approximately $1.17549436 \times 10^{-38}$ to $3.40282356 \times 10^{38}$.

A *double-precision floating-point* number is a IEEE 64-bit approximate representation of a real number. The range of magnitude is approximately $2.2250738585072014 \times 10^{-308}$ to $1.7976931348623158 \times 10^{308}$.

Decimal

A *decimal* value is a packed-decimal or zoned-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.

Data Types

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 negative $10^{31}+1$ to 10^{31} minus 1.

Numeric Host Variables

Small and large binary integer variables can be used in all host languages. Big integer variables can only be used in C, C++, ILE COBOL, and ILE RPG. Floating-point variables can be used in all host languages except RPG/400 and COBOL/400. Decimal variables can be used in all supported host languages.

String Representations of Numeric Values

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

Character Strings

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

Fixed-Length Character Strings

All values of a fixed-length character-string column have the same length. This is determined by the length attribute of the column. The length attribute must be between 1 through 32766 inclusive.

Varying-Length Character Strings

The types of varying-length character strings are:

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

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

For a VARCHAR column, the length attribute must be between 1 through 32740 inclusive. For a CLOB column, the length attribute must be between 1 through 2 147 483 647 inclusive. For more information about CLOBs, see “Large Objects (LOBs)” on page 65.

Character-String Variables

- Fixed-length character-string variables can be used in all host languages except REXX. (In C, fixed-length character-string variables are limited to a length of 1.)
- VARCHAR varying-length character-string variables can be used in C, COBOL, PL/I, REXX, and RPG:
 - In PL/I, REXX, and ILE RPG, there is a varying-length character-string data type.
 - In COBOL and C, varying-length character strings are represented as structures.

- In C, varying-length character-string variables can also be represented by NUL-terminated strings.
- In RPG/400, varying-length character-string variables can only be represented by VARCHAR columns included as a result of an externally described data structure.
- CLOB varying-length character-string variables can be defined in all host languages except REXX, RPG/400, and COBOL/400.
 - In ILE RPG, a CLOB varying-length character string is declared using the SQLTYPE keyword.
 - In all other languages, an SQL TYPE IS CLOB clause is used.

Character Encoding Schemes

Each character string is further defined as one of:

Bit data	Data that is not associated with a coded character set and is never converted. The CCSID for bit data is 65535.
SBCS data	Data in which every character is represented by a single byte. Each SBCS data character string has an associated CCSID. If necessary, an SBCS data character string is converted before it is used in an operation with a character string that has a different CCSID.
Mixed data	Data that may contain a mixture of characters from a single-byte character set (SBCS) and a double-byte character set (DBCS). Each mixed data character string has an associated CCSID. If necessary, a mixed data character string is converted before an operation with a character string that has a different CCSID. If mixed data contains a DBCS character, it cannot be converted to SBCS data.

The database manager does not recognize subclasses of double-byte characters, and it does not assign any specific meaning to particular double-byte codes. However, if you choose to use mixed data, then two single-byte EBCDIC codes are given special meanings:

- X'0E', the “shift-out” character, is used to mark the beginning of a sequence of double-byte codes.
- X'0F', the “shift-in” character, is used to mark the end of a sequence of double-byte codes.

In order for the database manager to recognize double-byte characters in a mixed data character string, the following condition must be met:

Within the string, the double-byte characters must be enclosed between paired shift-out and shift-in characters.

The pairing is detected as the string is read from left to right. The code X'0E' is recognized as a shift out character if X'0F' occurs later; otherwise, it is invalid. The first X'0F' following the X'0E' that is on a double-byte boundary is the paired shift-in character. Any X'0F' that is not on a double-byte boundary is not recognized.

There must be an even number of bytes between the paired characters, and each pair of bytes is considered to be a double-byte character. There can be more than one set of paired shift-out and shift-in characters in the string.

The length of a mixed data character string is its total number of bytes, counting two bytes for each double-byte character and one byte for each shift-out or shift-in character.

Data Types

When the job CCSID indicates that DBCS is allowed, CREATE TABLE will create character columns as DBCS-Open fields, unless FOR BIT DATA, FOR SBCS DATA, or an SBCS CCSID is specified. The SQL user will see these as character fields, but the system database support will see them as DBCS-Open fields. For a definition of a DBCS-Open field, see the Database Programming book.

Graphic Strings

A graphic string is a sequence of two-byte characters. The length of the string is the number of its characters. Like character strings, graphic strings can be empty.

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 through 16383 inclusive.

Varying-Length Graphic Strings

The types of varying-length graphic strings are:

- VARGRAPHIC (or synonym GRAPHIC VARYING)
- DBCLOB

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

For a VARGRAPHIC column, the length attribute must be between 1 through 16370 inclusive. For a DBCLOB column, the length attribute must be between 1 through 1 073 741 823 inclusive. For more information about DBCLOBs, see “Large Objects (LOBs)” on page 65.

Graphic-String Variables

- Fixed-length graphic-string host variables can be defined in C, ILE COBOL, and ILE RPG/400. (In C, fixed-length graphic-string host variables are limited to a length of 1.)
Although fixed-length graphic-string host variables cannot be defined in PL/I, COBOL/400, and RPG/400, a character-string host variable will be treated like a fixed-length graphic-string host variable if it was generated in the source from a GRAPHIC column in the external definition of a file.
- Varying-length graphic-string host variables can be defined in C, ILE COBOL, REXX, and ILE RPG.
 - In REXX and ILE RPG, there is a varying-length graphic-string data type.
 - In C and ILE COBOL, varying-length graphic strings are represented as structures.
 - In C, varying-length graphic-string variables can also be represented by NUL-terminated graphic strings.
 - Although varying-length graphic-string host variables cannot be defined in PL/I, COBOL/400, and RPG/400, a character-string host variable will be treated like a varying-length graphic-string host variable if it was generated in the source from a VARGRAPHIC column in the external definition of a file.
- DBCLOB varying-length character-string variables can be defined in all host languages except REXX, RPG/400, and COBOL/400.
 - In ILE RPG, a DBCLOB varying-length character string is declared using the SQLTYPE keyword.
 - In all other languages, an SQL TYPE IS DBCLOB clause is used.

Graphic Encoding Schemes

Each graphic string is further defined as DBCS data or UCS-2 data.

DBCS data Data in which every character is represented by a character from the double-byte character set (DBCS) that does not include the shift-out or shift-in characters.

Every DBCS graphic string has a CCSID that identifies a double-byte coded character set. If necessary, a DBCS graphic string is converted before it is used in an operation with a DBCS graphic string that has a different DBCS CCSID.

UCS-2 data Data in which every character is represented by a character from the Universal Coded Character Set (UCS-2).

When graphic-string host variables are not explicitly tagged with a CCSID, the associated DBCS CCSID for the job CCSID is used. If no associated DBCS CCSID exists, the host variable is tagged with 65535. A graphic-string host variable is never implicitly tagged with a UCS-2 CCSID. See the DECLARE VARIABLE statement for information on how to tag a graphic host variable with a CCSID.

Binary Strings

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

For a BLOB column, the length attribute must be between 1 and 2 147 483 647 bytes inclusive. For more information about BLOBs, see “Large Objects (LOBs)”.

A host variable with a BLOB string type can be defined in all host languages except REXX, RPG/400, and COBOL/400.

Large Objects (LOBs)

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

Character Large Object (CLOB) Strings

A *Character Large Object (CLOB)* is a varying-length character string with a maximum length of 2 147 483 647. A CLOB is used to store large SBCS data or mixed data, such as lengthy documents. For example, you can store information such as an employee resume, the script of a play, or the text of novel in a CLOB.

The CCSID of a CLOB cannot be 65535.

Double-byte Character Large Object (DBCLOB) Strings

A *Double-Byte Character Large Object (DBCLOB)* is a varying-length graphic string with a maximum length of 1 073 741 823 double-byte characters. A DBCLOB is designed to store large DBCS data, such as lengthy documents in UCS-2.

The CCSID of a DBCLOB cannot be 65535.

Binary Large Object (BLOB) Strings

A *Binary Large Object (BLOB)* is a varying-length string with a maximum length of 2 147 483 647. A BLOB is designed to store non-traditional data such as pictures, voice, and mixed media. BLOBs can also store structured data for use by distinct types and user-defined functions. A BLOB is considered to be a binary string.

Data Types

Although BLOB strings and FOR BIT DATA character strings might be used for similar purposes, the two data types are not compatible. The BLOB function can be used to change a FOR BIT DATA character string into a binary string.

The CCSID of a BLOB is 65535.

Manipulating Large Objects (LOBs) With Locators

When an application does not want an entire LOB value to be moved into a host variable, the application can use a large object locator (LOB locator) to reference the LOB value.

A LOB locator is a host variable with a value that represents a single LOB value in the database server. LOB locators provide users with a mechanism by which very large objects can be manipulated in application programs without requiring the entire LOB value to be stored in a host variable or transferred to the application requester (client) 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 application 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_value_1 CONCAT lob_value_2 CONCAT lob_value_3, 42, 6000000)
```

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.

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 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, CALL, and EXECUTE statements.

Limitations on Use of Strings

The following varying-length string data types cannot be referenced in certain contexts:

- for character strings, any CLOB string
- for graphic strings, DBCLOB string
- for binary strings, any BLOB string.

Table 4. Contexts for limitations on use of varying-length strings

Context of usage	LOB (CLOB, DBCLOB, or BLOB)
A GROUP BY clause	Not allowed
An ORDER BY clause	Not allowed
A CREATE INDEX statement	Not allowed
A SELECT DISTINCT statement	Not allowed
A subselect of a UNION without the ALL keyword	Not allowed
The definition of primary, unique, and foreign keys	Not allowed
Parameters of built-in functions	Some functions that allow varying-length character strings, varying-length graphic strings, or both types of strings as input arguments do not support CLOB or DBCLOB strings, or both as input. See the description of the individual functions in Chapter 3, “Built-In Functions” on page 163 for the data types that are allowed as input to each function.

Datetime Values

Although datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, they are neither strings nor numbers. However, strings can represent datetime values; see “String Representations of Datetime Values” on page 68.

Date

A *date* is a three-part value (year, month, and day) designating a point in time under the Gregorian calendar¹⁶, which is assumed to have been in effect from the year 1 A.D. The range of the year part is 0001 to 9999. The date formats *JUL, *MDY, *DMY, and *YMD can only represent dates in the range 1940 through 2039. The range of the month part is 1 to 12. The range of the day part is 1 to x , where x is 28, 29, 30, or 31, depending on the month and year.

The internal representation of a date is a string of 4 bytes that contains an integer. The integer (called the Scaliger number) represents the date.

The length of a DATE column as described in the SQLDA is 6, 8, or 10 bytes, depending on which format is used. These are the appropriate lengths for character-string representations for the value.

16. Note that historical dates do not always follow the Gregorian calendar. Dates between 1582-10-04 and 1582-10-15 are accepted as valid dates although they never existed in the Gregorian calendar.

Data Types

Time

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

The internal representation of a time is a string of 3 bytes. Each byte consists of two packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column as described in the 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 previously, except that the time includes a fractional specification of microseconds.

The internal representation of a timestamp is a string of 10 bytes. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds (the last 3 bytes contain 6 packed digits).

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.

Datetime Host Variables

Character string host variables are normally used to contain date, time, and timestamp values. However, date, time, and timestamp host variables can also be specified in ILE COBOL and ILE RPG. Date, time, and timestamp host variables can also be specified in Java as `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp` respectively.

String Representations of Datetime Values

Values whose data types are DATE, TIME, or TIMESTAMP are represented in an internal form that is transparent to the user of SQL. Dates, times, and timestamps, however, can also be represented by character or UCS-2 graphic strings. These representations directly concern the user of SQL since there are no constants whose data types are DATE, TIME, or TIMESTAMP. Only ILE RPG and ILE COBOL support datetime variables. To be retrieved, a datetime value can be assigned to a character-string variable. The format of the resulting string will depend on the default date format and the default time format in effect when the statement was prepared. The default date and time formats is set based on the date format (DATFMT), the date separator (DATSEP), the time format (TIMFMT), and the time separator (TIMSEP) parameters.

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. If the CCSID of the string represents a foreign encoding scheme (for example, ASCII), it is first converted to the coded character set identified by the default CCSID before the string is converted to the internal form of the datetime value.

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 6 characters. Trailing blanks can be included. Leading zeros can be omitted from the month and day portions when

using the IBM SQL standard formats. Each IBM SQL standard format is identified by name and includes an associated abbreviation (for use by the CHAR function). Other formats do not have an abbreviation to be used by the CHAR function. The separators for two-digit year formats are controlled by the date separator (DATSEP) parameter. Valid string formats for dates are listed in Table 5.

The database manager recognizes the string as a date when it is either:

- In the format specified by the default date format, or
- In one of the IBM SQL standard date formats, or
- In the unformatted Julian format

Table 5. Formats for String Representations of Dates

Format Name	Abbreviation	Date Format	Example
International Standards Organization (*ISO)	ISO	yyyy-mm-dd	1987-10-12
IBM USA standard (*USA)	USA	mm/dd/yyyy	10/12/1987
IBM European standard (*EUR)	EUR	dd.mm.yyyy	12.10.1987
Japanese industrial standard Christian era (*JIS)	JIS	yyyy-mm-dd	1987-10-12
Unformatted Julian	–	yyyyddd	1987285
Julian (*JUL)	–	yy/ddd	87/285
Month, day, year (*MDY)	–	mm/dd/yy	10/12/87
Day, month, year (*DMY)	–	dd/mm/yy	12/10/87
Year, month, day (*YMD)	–	yy/mm/dd	87/12/10

The default date format can be specified through the following interfaces:

Table 6. Default Date Format Interfaces

SQL Interface	Specification
Embedded SQL	The DATFMT and DATSEP parameters are specified on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to specify the DATFMT and DATSEP parameters within the source of a program containing embedded SQL. (For more information about CRTSQLxxx commands, see the SQL Programming with Host Languages book.)
Interactive SQL and Run SQL Statements	The DATFMT and DATSEP parameters on the Start SQL (STRSQL) command or by changing the session attributes. The DATFMT and DATSEP parameters on the Run SQL Statements (RUNSQLSTM) command. (For more information about STRSQL and RUNSQLSTM commands, see the SQL Programming Concepts book.)
Call Level Interface (CLI) on the server	SQL_ATTR_DATE_FMT and SQL_ATTR_DATE_SEP environment or connection variables (For more information about CLI, see the SQL Call Level Interfaces (ODBC) book.)

Data Types

Table 6. Default Date Format Interfaces (continued)

SQL Interface	Specification
JDBC or SQLJ on the server using Developer Kit for Java	Date Format and Date Separator connection property (For more information about JDBC and SQLJ, see the IBM Developer Kit for Java topic in the iSeries Information Center.)
ODBC on a client using the iSeries Access ODBC Driver	Date Format and Date Separator in the Advanced Server Options in ODBC Setup (For more information about ODBC, see the iSeries Access category in the iSeries Information Center.)
JDBC on a client using the IBM Toolbox for Java	Format in JDBC Setup (For more information about ODBC, see the iSeries Access category in the iSeries Information Center.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java topic in the iSeries Information Center .)

Time Strings: A string representation of a time is a character string that starts with a digit and has a length of at least 4 characters. Trailing blanks can be included; a leading zero can be omitted from the hour part of the time and seconds can be omitted entirely. If you choose to omit seconds, an implicit specification of 0 seconds is assumed. Thus, 13.30 is equivalent to 13.30.00.

Valid string formats for times are listed in Table 7. Each IBM SQL standard format is identified by name and includes an associated abbreviation (for use by the CHAR function). The other format (*HMS) does not have an abbreviation to be used by the CHAR function. The separator for the *HMS format is controlled by the time separator (TIMSEP) parameter.

The database manager recognizes the string as a time when it is either:

- In the format specified by the default time format, or
- In one of the IBM SQL standard time formats

The TIMFMT and TIMSEP parameters are specified on the CRTSQLxxx, RUNSQLSTM, and STRSQL commands. The SET OPTION statement can be used to specify TIMFMT and TIMSEP within the source of a program containing embedded SQL.

Table 7. Formats for String Representations of Times

Format Name	Abbreviation	Time Format	Example
International Standards Organization (*ISO)	ISO	hh.mm.ss ¹⁷	13.30.05
IBM USA standard (*USA)	USA	hh:mm AM or PM	1:30 PM
IBM European standard (*EUR)	EUR	hh.mm.ss	13.30.05
Japanese industrial standard Christian era (*JIS)	JIS	hh:mm:ss	13:30:05
Hours, minutes, seconds (*HMS)	–	hh:mm:ss	13:30:05

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

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

Table 8. USA Time Format

USA Format	24-Hour Clock
12:01 AM through 11:59 AM	00.01.00 through 00.59.00
01:00 AM through 11:59 AM	01:00.00 through 11:59.00
12:00 PM (noon) through 11:59 PM	12:00.00 through 23.59.00
12:00 AM (midnight)	24.00.00
00:00 AM (midnight)	00.00.00

In the USA format, a single space character exists between the minutes portion of the time of day and the AM or PM.

The default time format can be specified through the following interfaces:

Table 9. Default Time Format Interfaces

SQL Interface	Specification
Embedded SQL	The TIMFMT and TIMSEP parameters are specified on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to specify the TIMFMT and TIMSEP parameters within the source of a program containing embedded SQL. (For more information about CRTSQLxxx commands, see the SQL Programming with Host Languages book.)
Interactive SQL and Run SQL Statements	The TIMFMT and TIMSEP parameters on the Start SQL (STRSQL) command or by changing the session attributes. The TIMFMT and TIMSEP parameters on the Run SQL Statements (RUNSQLSTM) command. (For more information about STRSQL and RUNSQLSTM commands, see the SQL Programming Concepts book.)
Call Level Interface (CLI) on the server	SQL_ATTR_TIME_FMT and SQL_ATTR_TIME_SEP environment or connection variables (For more information about CLI, see the SQL Call Level Interfaces (ODBC) book.)
JDBC or SQLJ on the server using Developer Kit for Java	Time Format and Time Separator connection property object (For more information about JDBC and SQLJ, see the IBM Developer Kit for Java topic in the iSeries Information Center.)
ODBC on a client using the iSeries Access ODBC Driver	Time Format and Time Separator in the Advanced Server Options in ODBC Setup (For more information about ODBC, see the iSeries Access category in the iSeries Information Center.)
JDBC on a client using the IBM Toolbox for Java	Format in JDBC Setup (For more information about ODBC, see the iSeries Access category in the iSeries Information Center.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java topic in the iSeries Information Center .)

Data Types

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* or *yyyymmddhhmmss*. Trailing blanks can be included. Leading zeros can be omitted from the month, day, and hour part of the timestamp when using the timestamp form with separators. Trailing zeros can be truncated or omitted entirely from microseconds. If you choose to omit any digit of the microseconds portion, an implicit specification of 0 is assumed. Thus, *1990-3-2-8.30.00.10* is equivalent to *1990-03-02-08.30.00.100000*.

A timestamp whose time part is *24.00.00.000000* is also accepted.

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.

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.

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 is important to distinguish between these DataLink references to files and the LOB file reference variables described in "References to LOB File Reference Variables" on page 118. 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 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).

Row ID Values

A *row ID* is a value that uniquely identifies a row in a table. A column or a host variable can have a row ID data type. A ROWID column enables queries to be written that navigate directly to a row in the table. Each value in a ROWID column must be unique. The database manager maintains the values permanently, even across table reorganizations. When a row is inserted into the table, the database manager generates a value for the ROWID column unless one is supplied. If a value is supplied, it must be a valid row ID value that was previously generated by either DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries.

The internal representation of a row ID value is transparent to the user. The value is never subject to CCSID conversion because it is considered to contain BIT data. The length attribute of a ROWID column is 40.

User-Defined Types

Distinct Types

A *distinct type* is a user-defined data type that shares its internal representation with a built-in data type (its "source type"), but is considered to be a separate and incompatible type for most operations. For example, the semantics for a picture type, a text type, and an audio type that all use the built-in data type BLOB for their internal representation are quite different. A distinct type is created with the SQL statement CREATE DISTINCT TYPE.

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

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type that is not comparable to a BLOB or to any other type. This inability to compare AUDIO to other data types allows functions to be created specifically for AUDIO and assures that these functions cannot be applied to other data types.

The name of a distinct type is qualified with a schema name. The implicit schema name for an unqualified name depends upon the context in which the distinct type appears. If an unqualified distinct type name is used:

- In a CREATE DISTINCT TYPE or the object of DROP, COMMENT, GRANT, or REVOKE statement, the database manager uses the normal process of qualification by authorization ID to determine the schema name. For more information about qualification rules, see "Unqualified Function, Procedure, Specific, and Distinct Type Names" on page 54.
- In any other context, the database manager uses the SQL path to determine the schema name. The database manager searches the schemas in the path, in sequence, and selects the first schema that has a distinct type that matches. For a

Data Types

description of the SQL path, see “CURRENT PATH, CURRENT_PATH, or CURRENT FUNCTION PATH” on page 105.

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.) Instead, distinct types support *strong typing*. Strong typing ensures that only the functions and operators that are explicitly defined for a distinct type can be applied to that distinct type. However, a function or operator of the source type can be applied to the distinct type by creating an appropriate user-defined function. The user-defined function must be sourced on the existing function that has the source type as a parameter. For example, the following series of SQL statements shows how to create a distinct type named MONEY based on data type DECIMAL(9,2), how to define the + operator for the distinct type, and how the operator might be applied to the distinct type:

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M)
CREATE DISTINCT TYPE MONEY AS DECIMAL(9,2) WITH COMPARISONS
CREATE FUNCTION "+"(MONEY,MONEY)
  RETURNS MONEY
  SOURCE "+"(DECIMAL(9,2),DECIMAL(9,2))
CREATE TABLE SALARY_TABLE
  (SALARY MONEY,
  COMMISSION MONEY)
SELECT "+"(SALARY, COMMISSION) FROM SALARY_TABLE
```

A distinct type is subject to the same restrictions as its source type. For example, a table can only have one ROWID column. Therefore, a table with a ROWID column cannot also have a column with distinct type that is sourced on a row ID.

The comparison operators are automatically generated for distinct types, except for distinct types that are sourced on a DataLink. In addition, the database manager automatically generates functions for a distinct type that support casting from the source type to the distinct type and from the distinct type to the source type. For example, for the AUDIO type created above, these are the generated cast functions:

Name of generated cast function	Parameter list	Returns data type
schema-name.BLOB	schema-name.AUDIO	BLOB
schema-name.AUDIO	BLOB	schema-name.AUDIO

Promotion of Data Types

Data types can be classified into groups of related data types. Within such groups, an order of precedence exists in which one data type is considered to precede another data type. This precedence enables the database manager to support the *promotion* of one data type to another data type that appears later in the precedence order. For example, the database manager can promote the data type CHAR to VARCHAR and the data type INTEGER to DOUBLE PRECISION; however, the database manager cannot promote a CLOB to a VARCHAR.

The database manager considers the promotion of data types when:

- Performing function resolution (see “Function resolution” on page 124)
- Casting distinct types (see “Casting Between Data Types” on page 77)
- Assigning distinct types to built-in data types (see “Distinct Type Assignments” on page 88)

For each data type, Table 10 shows the precedence list (in order) that the database manager uses to determine the data types to which each data type can be promoted. The table indicates that the best choice is the same data type and not promotion to another data type. Note that the table also shows data types that are considered equivalent during the promotion process. For example, CHARACTER and GRAPHIC are considered to be equivalent data types.

Table 10. Data Type Precedence Table

Data Type	Data Type Precedence List (in best-to-worst order)
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
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB or DBCLOB
CLOB or DBCLOB	CLOB or DBCLOB
BLOB	BLOB
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
DATALINK	DATALINK
ROWID	ROWID
udt	udt (same name)

Promotion of Data Types

Table 10. Data Type Precedence Table (continued)

Data Type	Data Type Precedence List (in best-to-worst order)
-----------	----------------------------------------------------

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 a specification for single precision floating point

double = DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is a specification for double precision floating point

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.

Character and graphic strings are only compatible for Unicode data.

Casting Between Data Types

There are many occasions when a value with a given data type needs to be cast (changed) to a different data type or to the same data type with a different length, precision, or scale. Data type promotion, as described in “Promotion of Data Types” on page 75, is one example of when a value with one data type needs to be cast to a new data type. A data type that can be changed to another data type is *castable* from the source data type to the target data type.

The casting of one data type to another can occur implicitly or explicitly. You can use cast functions or CAST specifications to explicitly cast a data type. The database manager might implicitly cast data types during assignments that involve a distinct type (see “Distinct Type Assignments” on page 88). In addition, when you create a sourced user-defined function, the data types of the parameters of the source function must be castable to the data types of the function that you are creating (see “CREATE FUNCTION (Sourced)” on page 469).

If truncation occurs when a character or graphic string is cast to another data type, a warning occurs if any non-blank characters are truncated. This truncation behavior is similar to retrieval assignment of character or graphic strings (see “Retrieval Assignment” on page 83).

For casts that involve a distinct type as either the data type to be cast to or from, Table 11 shows the supported casts. For casts between built-in data types, Table 12 on page 78 shows the supported casts.

Table 11. Supported Casts When a Distinct Type is Involved

Data Type ...	Is Castable to Data Type ...
Distinct type <i>DT</i>	Source data type of distinct type <i>DT</i>
Source data type of distinct type <i>DT</i>	Distinct type <i>DT</i>
Distinct type <i>DT</i>	Distinct type <i>DT</i>
Data type <i>A</i>	Distinct type <i>DT</i> where <i>A</i> is promotable to the source data type of distinct type <i>DT</i> (see “Promotion of Data Types” on page 75)
INTEGER	Distinct type <i>DT</i> if <i>DT</i> 's source type is SMALLINT
DOUBLE	Distinct type <i>DT</i> if <i>DT</i> 's source data type is REAL
VARCHAR	Distinct type <i>DT</i> if <i>DT</i> 's source data type is CHAR or GRAPHIC
VARGRAPHIC	Distinct type <i>DT</i> if <i>DT</i> 's source data type is GRAPHIC or CHAR

When a distinct type is involved in a cast, a cast function that was generated when the distinct type was created is used. How the database manager chooses the function depends on whether function notation or the CAST specification syntax is used. For details, see “Function resolution” on page 124, and “CAST Specification” on page 143. Function resolution is used for both. However, in a CAST specification, when an unqualified distinct type is specified as the target data type, the database manager resolves the schema name of the distinct type and then uses that schema name to locate the cast function.

The following table describes the supported casts between built-in data types. See the description preceding the table for information on supported casts involving user-defined types.

Casting Between Data Types

Table 12. Supported Casts Between Built-In Data Types

Source Data Type	SMALLINT		CHAR		GRAPHIC	BLOB			TIME	ROW ID
	INTEGER	DECIMAL	REAL	VARCHAR	VARGRAPHIC	DATE	TIME	STAMP		
	BIGINT	NUMERIC	DOUBLE	CLOB	DBCLOB					
SMALLINT	Y	Y	Y	Y	—	—	—	—	—	—
INTEGER	Y	Y	Y	Y	—	—	—	—	—	—
BIGINT	Y	Y	Y	Y	—	—	—	—	—	—
DECIMAL	Y	Y	Y	Y	—	—	—	—	—	—
NUMERIC	Y	Y	Y	Y	—	—	—	—	—	—
REAL	Y	Y	Y	Y	—	—	—	—	—	—
DOUBLE	Y	Y	Y	Y	—	—	—	—	—	—
CHAR	Y	Y	Y	Y	*	Y	Y	Y	Y	Y
VARCHAR	Y	Y	Y	Y	*	Y	Y	Y	Y	Y
CLOB	Y	Y	Y	Y	*	Y	—	—	—	—
GRAPHIC	—	—	—	Y*	Y	Y	—	—	—	—
VARGRAPHIC	—	—	—	Y*	Y	Y	—	—	—	—
DBCLOB	—	—	—	Y*	Y	Y	—	—	—	—
BLOB	—	—	—	—	—	Y	—	—	—	—
DATE	—	—	—	Y**	—	—	Y	—	Y	—
TIME	—	—	—	Y**	—	—	—	Y	Y	—
TIMESTAMP	—	—	—	Y**	—	—	Y	Y	Y	—
ROWID	—	—	—	Y	—	Y	—	—	—	Y

Notes: * Conversion is only supported for UCS-2 graphic.

** Casting from DATE, TIME, TIMESTAMP, or ROWID to CLOB is not supported.

Only a DATALINK can be cast to a DATALINK type.

The following table describes the rules for casting to a data type:

Table 13. Rules for Casting to a Data Type

Target Data Type	Source Data Type	Rules
SMALLINT	Any	See the SMALLINT scalar function.
INTEGER	Any	See the INTEGER scalar function.
BIGINT	Any	See the BIGINT scalar function.
DECIMAL	Any	See the DECIMAL scalar function.
NUMERIC	Any	See the ZONED scalar function.
REAL	Any	See the REAL scalar function.
DOUBLE	Any	See the DOUBLE scalar function.
CHAR	Any	See the CHAR scalar function.
VARCHAR	Any	See the VARCHAR scalar function.
CLOB	Any	See the CLOB scalar function.
GRAPHIC	Any	See the rules for string assignment to a host variable.
VARGRAPHIC	Any	See the rules for string assignment to a host variable.
DBCLOB	Any	See the DBCLOB scalar function.
BLOB	Any	See the BLOB scalar function.
DATE	Any	See the DATE scalar function.
TIME	Any	See the TIME scalar function.
TIMESTAMP	CHAR	See the TIMESTAMP scalar function, where one operand is specified.
TIMESTAMP	DATE	The timestamp is composed of the specified date and a time of 00:00:00.
TIMESTAMP	TIME	The timestamp is composed of the CURRENT_DATE and the specified time.
DATALINK	DATALINK	See the rules for DataLink assignments.
ROWID	Any	See the ROWID scalar function.

I

Assignments and Comparisons

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of CALL, INSERT, UPDATE, FETCH, SELECT, SET variable, and VALUES INTO statements. 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 UNION, concatenation, CASE expressions, and the CONCAT, VALUE, COALESCE, IFNULL, MIN, and MAX scalar functions. The compatibility matrix is as follows:

Table 14. Data Type Compatibility

Operands	Binary Integer	Decimal Number ⁴	Floating Point	Character String	Graphic String	Binary String	Date	Time	Timestamp	Distinct Type
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	Yes ⁵	No ³	1	1	1	2
Graphic String	No	No	No	Yes ⁵	Yes	No	No	No	No	2
Binary String	No	No	No	No ³	No	Yes	No	No	No	2
Date	No	No	No	1	No	No	Yes	No	No	2
Time	No	No	No	1	No	No	No	Yes	No	2
Timestamp	No	No	No	1	No	No	No	No	Yes	2
Distinct Type	2	2	2	2	2	2	2	2	2	2

Notes:

1. The compatibility of datetime values and character strings is limited to assignment, comparison, and the VALUE, COALESCE, IFNULL, MIN, and MAX scalar functions.
 - Datetime values can be assigned to character-string columns and to character-string variables as explained in “Datetime Assignments” on page 85.
 - A valid string representation of a date can be assigned to a date column, compared with a date, or used in a VALUE, COALESCE, IFNULL, MIN, or MAX scalar function with a date.
 - A valid string representation of a time can be assigned to a time column, compared with a time, or used in a VALUE, COALESCE, IFNULL, MIN, or MAX scalar function with a time.
 - A valid string representation of a timestamp can be assigned to a timestamp column, compared with a timestamp, or used in a VALUE, COALESCE, IFNULL, MIN, or MAX scalar function with a timestamp.
2. A value with a distinct type is comparable only to a value that is defined with the same distinct type. In general, the database manager supports assignments between a distinct type value and its source data type. For additional information, see “Distinct Type Assignments” on page 88.
3. All character strings, even those with FOR BIT DATA, are not compatible with binary strings.
4. Decimal refers to both packed and zoned decimal.
5. Bit data and graphic strings are not compatible.
6. 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. A DATALINK operand can not be directly compared to any data type. The DLCOMMENT, DLLINKTYPE, DLURLCOMPLETE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, and DLURLSERVER scalar functions can be used to extract character string values from a datalink which can then be compared to other strings.
7. A ROWID operand can only be assigned to another ROWID operand.

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 114 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 cannot be truncated. If necessary, the fractional part of a decimal number is truncated. In the case of the assignment to a host variable, a positive value may be returned in the SQLCODE.

An error occurs if:

- Truncation of the whole part of the number occurs on assignment to a column
- Truncation of the whole part of the number occurs on assignment to a host variable that does not have an indicator variable

A warning occurs if:

Truncation of the whole part of the number occurs on assignment to a host variable with an indicator variable. In this case, the number is not assigned to the host variable and the indicator variable is set to negative 2.

Note: Decimal refers to both packed and zoned decimal.

Note: When fetching decimal data from a file that was *not* created by an SQL CREATE TABLE statement, a decimal field may contain data that is not valid. In this case, the data will be returned as stored, without any warning or error message being issued. A table that is created by the SQL CREATE TABLE statement does not allow decimal data that is not valid.

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.

The approximation is more accurate if the receiving column or variable is defined as double precision (64 bits) rather than single precision (32 bits).

Floating-Point or Decimal to Integer

When a decimal or floating-point number is assigned to a binary integer column or variable, the number is converted, if necessary, to the precision and the scale of the target. If the scale of the target is zero, the fractional part of the number is lost. The necessary number of leading zeros is added or eliminated, and, in the fractional part of the number, the necessary number of trailing zeros is added, or the necessary number of trailing digits is eliminated.

Decimal to Decimal

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and the scale of the target. The necessary number of leading zeros is added or eliminated, and, in the fractional part of the number, the necessary number of trailing zeros is added, or the necessary number of trailing digits is eliminated.

Integer to Decimal

When an integer is assigned to a decimal column or variable, the number is converted first to a temporary decimal number and then, if necessary, to the precision and scale of the target. If the scale of the integer is zero, the precision of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer.

Floating-Point to Decimal

When a floating-point number is assigned to a decimal column or variable, 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 0.5×10^{-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.

To COBOL and RPG Integers

Assignment to COBOL and RPG small or large integer host variables takes into account any scale specified for the host variable. However, assignment to integer host variables uses the full size of the integer. Thus, the value placed in the COBOL data item or RPG field may be larger than the maximum precision specified for the host variable.

In COBOL, for example, if COL1 contains a value of 12345, the statements:

```
01 A PIC S9999 BINARY.
EXEC SQL SELECT COL1
        INTO :A
        FROM TABLEX
END-EXEC.
```

result in the value 12345 being placed in A, even though A has been defined with only 4 digits.

Notice that the following COBOL statement:

```
MOVE 12345 TO A.
```

results in 2345 being placed in A.

String Assignments

There are two types of string assignments:

- *Storage assignment* is when a value is assigned to a column or a parameter of a function or stored procedure.
- *Retrieval assignment* is when a value is assigned to a host variable.

Binary String Assignments

Storage Assignment: The basic rule is that the length of a string assigned to a column or parameter of a function or procedure must not be greater than the length attribute of the column or parameter. If the string is longer than the length attribute of that column, a negative SQLCODE is returned. For a description of the SQLCA, see Appendix B, "SQL Communication Area" on page 825.

Retrieval Assignment: The length of a string assigned to a host variable can be greater than the length attribute of the host variable. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters. When this occurs, the value 'W' is assigned to the SQLWARN1 field of the SQLCA.

When a string of length n is assigned to a varying-length string variable with a maximum length greater than n , the bytes after the n th byte of the variable are undefined.

Assignments and Comparisons

Character and Graphic String Assignments

The following rules apply when both the source and the target are strings. When a datetime data type is involved, see “Datetime Assignments” on page 85. For the special considerations that apply when a distinct type is involved in an assignment, especially to a host variable, see “Distinct Type Assignments” on page 88.

Storage Assignment: The basic rule is that the length of a string assigned to a column or parameter of a function or procedure must not be greater than the length attribute of the column or parameter. If the string is longer than the length attribute of that column, a negative SQLCODE is returned. (Trailing blanks are normally included in the length of the string. For storage assignments, however, trailing blanks are not included in the length of the string.)

For a description of the SQLCA, see Appendix B, “SQL Communication Area” on page 825.

When a string is assigned to a fixed-length string column or parameter and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte, double-byte, or UCS-2 blanks.¹⁸ The pad character is always a blank, even for bit data.

Retrieval Assignment: The length of a string assigned to a host variable can be greater than the length attribute of the host variable. When a string is assigned to a variable and the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters. When this occurs, the value 'W' is assigned to the SQLWARN1 field of the SQLCA. Furthermore, if an indicator variable is provided, it is set to the original length of the string. If only the NUL-terminator is truncated for a C NUL-terminated host variable and the *NOCNULRQD option was specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement), the value of 'N' is assigned to the SQLWARN1 field of the SQLCA and a NUL is not placed in the variable.

When a string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte, double-byte, or UCS-2 blanks.¹⁸ The pad character is always a blank, even for bit data.

When a string of length n is assigned to a varying-length string variable with a maximum length greater than n , the characters after the n th character of the variable are undefined.

Assignments Involving Mixed Strings: If a string contains mixed data, the assignment rules may require truncation within a sequence of double-byte codes. To prevent the loss of the shift-in character that ends the double-byte sequence, additional characters may be truncated from the end of the string, and a shift-in character added. In the truncated result, there is always an even number of bytes between each shift-out character and its matching shift-in character.

Assignments Involving C NUL-terminated Strings: When a string of length n is assigned to a C NUL-terminated string variable with a length greater than $n+1$:

- If the *CNULRQD option was specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the string is

18. UCS-2 defines a blank character at code point X'0020' and X'3000'. The database manager pads with the blank at code point X'0020'.

Assignments and Comparisons

padding on the right with $x-n-1$ blanks where x is the length of the variable. The padded string is then assigned to the variable and the NUL-terminator is placed in the next character position.

- If the *NOCNULRQD precompiler option was specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement), the string is not padded on the right. The string is assigned to the variable and the NUL-terminator is placed in the next character position.

Conversion Rules for Assignments: A 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 CCSIDs are different.
- Neither CCSID is 65535.
- The string is neither null nor empty.
- The CCSID Conversion Selection Table indicates that conversion is necessary.

An error occurs if:

- The CCSID Conversion Selection Table is used but does not contain any information about the pair of CCSIDs.
- A character of the string cannot be converted, and the operation is assignment to a column or assignment to a host variable without an indicator variable. For example, a double-byte character (DBCS) cannot be converted to a column or host variable with a single-byte character (SBCS) CCSID.

A warning occurs if:

- A character of the string is converted to the substitution character.
- A character of the string cannot be converted, and the operation is assignment to a host variable with an indicator variable. For example, a DBCS character cannot be converted to a host variable with an SBCS CCSID. In this case, the string is not assigned to the host variable and the indicator variable is set to -2.

Datetime Assignments

A value assigned to a DATE column must be a date or a valid string representation of a date. A date can only be assigned to a DATE column, a character-string column, a character-string variable or an ILE RPG/400 timestamp variable. A value assigned to a TIME column must be a time or a valid string representation of a time. A time can only be assigned to a TIME column, a character-string column, a character-string variable or an ILE RPG/400 timestamp variable. A value assigned to a TIMESTAMP column must be a timestamp or a valid string representation of a timestamp. A timestamp can only be assigned to a TIMESTAMP column, a character-string column, a character-string variable or an ILE RPG/400 timestamp variable.

When a datetime value is assigned to a character-string variable or column, it is converted to its string representation. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target varies depending on the format of the string representation. If the length of the target is greater than required, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of datetime value involved and on the type of target.

- If the target is a character-string column, truncation is not allowed. The following rules apply:

DATE

Assignments and Comparisons

The length attribute of the column must be at least 10 if the date format is *ISO, USA, *EUR, or *JIS. If the date format is *YMD, *MDY, or *DMY, the length attribute of the column must be at least 8. If the date format is *JUL, the length of the host variable must be at least 6.

TIME

The length attribute of the column must be at least 8.

TIMESTAMP

The length attribute of the column must be at least 26.

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

DATE

The length of the host variable must be at least 10 if the date format is *ISO, *USA, *EUR, or *JIS. If the date format is *YMD, *MDY, or *DMY, the length of the host variable must be at least 8. If the date format is *JUL, the length of the host variable must be at least 6.

TIME

- If the *USA format is used, the length of the host variable must not be less than 8. This format does not include seconds.
- If the *ISO, *EUR, *JIS, or *HMS time format is used, the length of the host variable must not be less than 5. If the length is 5, 6, or 7, the seconds part of the time is omitted from the result, and SQLWARN1 is set to 'W'. In this case, the seconds part of the time is assigned to the indicator variable if one is provided, and, if the length is 6 or 7, blank padding occurs so that the value is a valid string representation of a time.

TIMESTAMP

The length of the host variable must not be less than 19. If the length is between 19 and 25, the timestamp is truncated like a string, causing the omission of one or more digits of the microsecond part. If the length is 20, the trailing decimal point is replaced by a blank so that the value is a valid string representation of a timestamp.

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 link pending 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 by using the DLVALUE scalar function. The DLVALUE scalar function creates a new DataLink value which can then be assigned 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 can occur:

- Data Location (URL) format is invalid
- File server is not registered with this database
- Invalid link type specified
- Invalid length of comment or URL

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 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.
- File does not exist.
- Referenced file cannot be accessed for linking.
- File already linked to another column.

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

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

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

A DataLink value may be retrieved from the database through the use of scalar functions (such as DLLINKTYPE and DLURLPATH). The results of these scalar functions can then be assigned to host variables.

Note that usually no attempt is made to access the file server at retrieval time.¹⁹It is therefore possible that subsequent attempts to access the file server through file system commands might fail.

A warning may be returned when retrieving a DataLink value because the table is in link pending state.

Row ID Assignments

A row ID value can only be assigned to a column, parameter, or host variable with a row ID data type. For the value of the ROWID column, the column must be defined as GENERATED BY DEFAULT or OVERRIDING SYSTEM VALUE must be specified. A unique constraint is implicitly added to every table that has a ROWID column that guarantees that every ROWID value is unique. The value that is specified for the column must be a valid row ID value that was previously generated by DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries.

¹⁹It 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.

Assignments and Comparisons

Distinct Type Assignments

The rules that apply to the assignments of distinct types to host variables are different than the rules for all other assignments that involve distinct types.

Assignments to Host Variables

The assignment of a distinct type to a host variable is based on the source data type of the distinct type. Therefore, the value of a distinct type is assignable to a host variable only if the source data type of the distinct type is assignable to the host variable.

Example: Assume that distinct type AGE was created with the following SQL statement and column STU_AGE in table STUDENTS was defined with that distinct type. 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.

```
CREATE DISTINCT TYPE AGE AS SMALLINT WITH COMPARISONS
```

Next, consider this valid assignment of a student's age to host variable HV_AGE, which has an INTEGER data type.

```
SELECT STU_AGE INTO :HV_AGE FROM STUDENTS WHERE STU_NUMBER = 200
```

The distinct type value is assignable to the host variable HV_AGE because the source data type of the distinct type (SMALLINT) is assignable to the host variable (INTEGER). If distinct type AGE had been sourced on a character data type such as CHAR(5), the above assignment would be invalid because a character type cannot be assigned to an integer type.

Assignments Other Than to Host Variables

A distinct type can be either the source or target of an assignment. Assignment is based on whether the data type of the value to be assigned is castable to the data type of the target. "Casting Between Data Types" on page 77 shows which casts are supported when a distinct type is involved. Therefore, a distinct type value can be assigned to any target other than a host variable when:

- The target of the assignment has the same distinct type, or
- The distinct type is castable to the data type of the target

Any value can be assigned to a distinct type when:

- The value to be assigned has the same distinct type as the target, or
- The data type of the assigned value is castable to the target distinct type

Example: Assume that the source data type for distinct type AGE is SMALLINT:

```
CREATE DISTINCT TYPE AGE AS SMALLINT WITH COMPARISONS
```

Next, assume that two tables TABLE1 and TABLE2 were created with four identical column descriptions:

```
AGECOL    AGE
SMINTCOL  SMALLINT
INTCOL    INTEGER
DECCOL    DEC(6,2)
```

Using the following SQL statement and substituting various values for X and Y to insert values into various columns of TABLE1 from TABLE2, Table 15 on page 89 shows whether the assignments are valid.

```
INSERT INTO TABLE1 (Y) SELECT X FROM TABLE2
```

Table 15. Assessment of various assignments (for example on INSERT)

TABLE2.X	TABLE1.Y	Valid	Reason
AGECOL	AGECOL	Yes	Source and target are same distinct type
SMINTCOL	AGECOL	Yes	SMALLINT can be cast to AGE (because AGE's source type is SMALLINT)
INTCOL	AGECOL	Yes	INTEGER can be cast to AGE (because AGE's source type is SMALLINT)
DECCOL	AGECOL	No	DECIMAL cannot be cast to AGE
AGECOL	SMINTCOL	Yes	AGE can be cast to its source type SMALLINT
AGECOL	INTCOL	No	AGE cannot be cast to INTEGER
AGECOL	DECCOL	No	AGE cannot be cast to DECIMAL

Numeric Comparisons

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

If one number is an integer and the other number is decimal, the comparison is made with a temporary copy of the integer, which has been converted to decimal.

When decimal or nonzero scale binary 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 number is integer, decimal, or single-precision floating point, the comparison is made with a temporary copy of the second number converted to a double-precision floating-point number. However, if a single-precision floating-point column is compared to a constant and the constant can be represented by a single-precision floating-point number, the comparison is made with a single-precision form of the constant.

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

String Comparisons

Binary String Comparisons

Binary string comparisons always use a sort sequence of *HEX and the corresponding bytes of each string are compared. Additionally, two binary strings are equal only if the length of the two strings is identical.

Character and Graphic String Comparisons

Character and UCS-2 graphic string comparisons use the sort sequence in effect when the statement is executed for all SBCS data and the single-byte portion of mixed data. If the sort sequence is *HEX, the corresponding bytes of each string are compared. For all other sort sequences, the corresponding bytes of the weighted value of each string are compared.

Assignments and Comparisons

If the strings have different lengths, a temporary copy of the shorter string is padded on the right with blanks before comparison. The padding makes each string the same length. The pad character is always a blank, regardless of the sort sequence. For bit data, the pad character is also a blank. For DBCS graphic data, the pad character is a DBCS blank (x'4040'). For UCS-2 graphic data, the pad character is a UCS-2 blank.²⁰

Two strings are equal if any of the following are true:

- Both strings are empty.
- A *HEX sort sequence is used and all corresponding bytes are equal.
- A sort sequence other than *HEX is used and all corresponding bytes of the weighted value are equal.

An empty string is equal to a blank string. The relationship between two unequal strings is determined by a comparison of the first pair of unequal bytes (or bytes of the weighted value) from the left end of the string. This comparison is made according to the sort sequence in effect when the statement is executed.

Two varying-length strings with different lengths are equal if they differ only in the number of trailing blanks. In operations that select one value from a set of such values, the value selected is arbitrary. The operations that can involve such an arbitrary selection are DISTINCT, MAX, MIN, UNION and references to a grouping column. See the description of GROUP BY for further information about the arbitrary selection involved in references to a grouping column.

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

- The CCSIDs of the two strings are different.
- Neither CCSID is 65535.
- The string selected for conversion is neither null nor empty.
- The CCSID Conversion Selection Table indicates that conversion is necessary.

If two strings with different encoding schemes are compared and the operands are the same type, any necessary conversion applies to the string as follows:

Table 16. Selecting the Encoding Scheme for Character Conversion

First Operand	Second Operand			
	SBCS Data	DBCS Data	Mixed Data	UCS-2 Data
SBCS Data	See below	Second	Second	Second
DBCS Data	First	See below	Second	Second
Mixed Data	First	First	See below	Second
UCS-2 Data	First	First	First	See below

Otherwise, the string selected for conversion depends on the type of each operand. The following table shows which operand is selected for conversion, given the operand types:

20. UCS-2 defines a blank character at code point X'0020' and X'3000'. The database manager pads with the blank at code point X'0020'.

Table 17. Selecting the Operand for Character Conversion

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

A host variable containing data in a foreign encoding scheme is always effectively converted to the native encoding scheme before it is used in any operation. The above rules are based on the assumption that this conversion has already occurred.

An error occurs if a character of the string cannot be converted or the CCSID Conversion Selection Table is used but does not contain any information about the pair of CCSIDs. A warning occurs if a character of the string is converted to the substitution character.

Datetime Comparisons

A DATE, TIME, or TIMESTAMP value can be compared either with another value of the same data type or with a string representation of that data type. All comparisons are chronological, which means the 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 are implied. The time 24:00:00 compares greater than the time 00:00:00.

Comparisons involving TIMESTAMP values are chronological without regard to representations that might be considered equivalent. Thus, the following predicate is true:

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

Distinct Type Comparisons

A value with a distinct type can be compared only to another value with exactly the same distinct type.

For example, assume that distinct type YOUTH and table CAMP_DB2_ROSTER table were created with the following SQL statements:

```
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 because AGE and HIGH_SCHOOL_LEVEL have the same distinct type:

Assignments and Comparisons

```
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
```

However, AGE can be compared to ATTENDEE_NUMBER by using a cast function or CAST specification to cast between the distinct type and the source type. All of the following comparisons are valid:

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

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

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

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


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

The data type of the result is determined by the data type of the operands. The data types of the first two operands determine an intermediate result data type, this data type and the data type of the next operand determine a new intermediate result data type, and so on. The last intermediate result data type and the data type of the last operand determine the data type of the result. For each pair of data types, the result data type is determined by the sequential application of the rules summarized in the tables that follow.

If neither operand column allows nulls, the result does not allow nulls. Otherwise, the result allows nulls.

If the data type and attributes of any operand column are not the same as those of the result, the operand column values are converted to conform to the data type and attributes of the result. The conversion operation is exactly the same as if the values were assigned to the result. For example,

- If one operand column is CHAR(10), and the other operand column is CHAR(5), the result is CHAR(10), and the values derived from the CHAR(5) column are padded on the right with five blanks.
- If the whole part of a number cannot be preserved then an error is returned.

Numeric Operands

Numeric types are compatible only with other numeric types.

If one operand column is...	And the other operand is...	The data type of the result column is...
SMALLINT	SMALLINT	SMALLINT
INTEGER	SMALLINT	INTEGER
INTEGER	INTEGER	INTEGER
BIGINT	SMALLINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	BIGINT	BIGINT
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where $p = \min(31, x + \max(w-x, 5))$
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where $p = \min(31, x + \max(w-x, 11))$
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where $p = \min(31, x + \max(w-x, 19))$

Rules for Result Data Types

If one operand column is...	And the other operand is...	The data type of the result column is...
DECIMAL(w,x)	DECIMAL(y,z) or NUMERIC(y,z)	DECIMAL(p,s) where $p = \min(31, \max(x,z) + \max(w-x, y-z))$ $s = \max(x,z)$
NUMERIC(w,x)	SMALLINT	NUMERIC(p,x) where $p = \min(31, x + \max(w-x, 5))$
NUMERIC(w,x)	INTEGER	NUMERIC(p,x) where $p = \min(31, x + \max(w-x, 11))$
NUMERIC(w,x)	BIGINT	NUMERIC(p,x) where $p = \min(31, x + \max(w-x, 19))$
NUMERIC(w,x)	NUMERIC(y,z)	NUMERIC(p,s) where $p = \min(31, \max(x,z) + \max(w-x, y-z))$ $s = \max(x,z)$
NONZERO SCALE BINARY	NONZERO SCALE BINARY	NONZERO SCALE BINARY (If either operand is nonzero scale binary, both operands must be binary with the same scale.)
REAL	REAL	REAL
REAL	DECIMAL, NUMERIC, BIGINT, INTEGER, or SMALLINT	DOUBLE
DOUBLE	any numeric type	DOUBLE

Character and Graphic String Operands

Character and graphic strings are compatible with other character and graphic strings when there is a defined conversion between their corresponding CCSIDs.

Rules for Result Data Types

If one operand column is...	And the other operand is...	The data type of the result column is...
CHAR(x)	CHAR(y)	CHAR(z) where $z = \max(x,y)$
GRAPHIC(x)	GRAPHIC(y) or CHAR(y)	GRAPHIC(z) where $z = \max(x,y)$
VARCHAR(x)	VARCHAR(y) or CHAR(y)	VARCHAR(z) where $z = \max(x,y)$
VARCHAR(x)	GRAPHIC(y)	VARGRAPHIC(z) where $z = \max(x,y)$
VARGRAPHIC(x)	VARGRAPHIC(y) or GRAPHIC(y) or VARCHAR(y) or CHAR(y)	VARGRAPHIC(z) where $z = \max(x,y)$
CLOB(x)	CLOB(y) or VARCHAR(y) or CHAR(y)	CLOB(z) where $z = \max(x,y)$
CLOB(x)	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where $z = \max(x,y)$
DBCLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y) or GRAPHIC(y) or VARGRAPHIC(y) or DBCLOB(y)	DBCLOB(z) where $z = \max(x,y)$

The CCSID of the result graphic string will be derived based on the “Conversion Rules for Operations That Combine Strings” on page 97.

Binary String Operands

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

If one operand column is...	And the other operand is...	The data type of the result column is...
BLOB(x)	BLOB(y)	BLOB(z) where $z = \max(x,y)$

Datetime Operands

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

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

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

Rules for Result Data Types

If one operand column is...	And the other operand is...	The data type of the result column is...
DATE	DATE	DATE
TIME	TIME	TIME
TIMESTAMP	TIMESTAMP	TIMESTAMP

Distinct Type Operands

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

If one operand column is...	And the other operand is...	The data type of the result column is...
Distinct Type	Distinct Type	Distinct Type

DataLink Operands

A DataLink is compatible with another DataLink. However, DataLinks with NO LINK CONTROL are only compatible with other DataLinks with NO LINK CONTROL; DataLinks with FILE LINK CONTROL READ PERMISSION FS are only compatible with other DataLinks with FILE LINK CONTROL READ PERMISSION FS; and DataLinks with FILE LINK CONTROL READ PERMISSION DB are only compatible with other DataLinks with FILE LINK CONTROL READ PERMISSION DB. The data type of the result is DATALINK. The length of the result DATALINK is the largest length of all the data types.

If one operand column is...	And the other operand is...	The data type of the result column is...
DATALINK(x)	DATALINK(y)	DATALINK(z) where $z = \max(x,y)$

Conversion Rules for Operations That Combine Strings

The operations that combine strings are concatenation, UNION, and UNION ALL. (These rules also apply to the MAX, MIN, VALUE, COALESCE, IFNULL, and CONCAT scalar functions and CASE expressions.) In each case, the CCSID 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 CCSID.

The CCSID of the result is determined by the CCSIDs of the operands. The CCSIDs of the first two operands determine an intermediate result CCSID, this CCSID and the CCSID of the next operand determine a new intermediate result CCSID, and so on. The last intermediate result CCSID and the CCSID of the last operand determine the CCSID of the result string or column. For each pair of CCSIDs, the result CCSID is determined by the sequential application of the following rules:

- If the CCSIDs are equal, the result is that CCSID.
- If either CCSID is 65535, the result is 65535.²¹
- If one CCSID denotes data in an encoding scheme different from the other CCSID, the result is determined by the following table:

Table 18. Selecting the Encoding Scheme of the Intermediate Result

First Operand	Second Operand			
	SBCS Data	DBCS Data	Mixed Data	UCS-2 Data
SBCS Data	See below	Second	Second	Second
DBCS Data	First	See below	Second	Second
Mixed Data	First	First	See below	Second
UCS-2 Data	First	First	First	See below

- Otherwise, the resulting CCSID is determined by the following table:

Table 19. Selecting the CCSID 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

However, a host variable containing data in a foreign encoding scheme is effectively converted to the native encoding scheme before it is used in any operation. The above rules are based on the assumption that this conversion has already occurred.

Note that an intermediate result is considered to be a derived value operand. For example, assume COLA, COLB, and COLC are columns with CCSIDs 37, 278, and 500, respectively. The result CCSID of COLA CONCAT COLB CONCAT COLC is determined as follows:

21. If either operand is a CLOB or DBCLOB, the resulting CCSID is the job default CCSID.

Conversion Rules for Operations That Combine Strings

1. The result CCSID of COLA CONCAT COLB is first determined to be 37 because both operands are columns, so the CCSID of the first operand is chosen.
2. The result CCSID of the concatenation of the result from step 1 and COLC is determined to be 500. The result CCSID of 500 is determined because the first operand is a derived value and the second operand is a column, so the CCSID of the second operand is chosen.

An operand of concatenation or the selected argument of the MAX, MIN, VALUE, COALESCE, IFNULL, and CONCAT scalar function is converted, if necessary, to the coded character set of the result string. Each string of an operand of UNION or UNION ALL is converted, if necessary, to the coded character set of the result column. Character conversion is necessary only if all of the following are true:

- The CCSIDs are different.
- Neither CCSID is 65535.
- The string is neither null nor empty.
- The CCSID Conversion Selection Table indicates that conversion is necessary.

An error occurs if a character of a string cannot be converted or if the CCSID Conversion Selection Table is used but does not contain any information about the CCSID pair. A warning occurs if a character of a string is converted to the substitution character.

Constants

A *constant* (sometimes called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. String constants are further classified as character or graphic. Numeric constants are further classified as integer, floating point, or decimal.

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

Integer Constants

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

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

Examples

64 -15 +100 32767 720176 12345678901

Floating-Point Constants

A *floating-point constant* specifies a double-precision floating-point number as two numbers separated by an E. The first number can include a sign and a decimal point; the second number can include a sign but not a decimal point. The value of the constant is the product of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of characters in the constant must not exceed 24. Excluding leading zeros, the number of digits in the first number must not exceed 17 and the number of digits in the second must not exceed 3.

Examples

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

Decimal Constants

A *decimal constant* specifies a decimal number as a signed or unsigned number that includes at most 31 digits. The constant must either:

- Include a decimal point, or
- Be larger than 2147483647 or smaller than -2147483647

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 12345678901

Character-String Constants

A *character-string constant* specifies a varying-length character string. The two forms of character-string constant follow:

Constants

- A sequence of characters that starts and ends with a string delimiter. The number of bytes between the string delimiters cannot be greater than 32740. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive string delimiters that are not contained within a string represent the empty string.
- An X followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 65480. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. This form of string constant allows you to specify characters that do not have a keyboard representation.

Character-string constants can contain mixed data. If the job CCSID supports mixed data, a character-string constant is classified as mixed data if it includes a DBCS substring. In all other cases, a character-string constant is classified as SBCS data.

The CCSID assigned to the constant is the CCSID of the source containing the constant unless the source is encoded in a foreign encoding scheme (such as ASCII). The data in the host variable is converted from the foreign encoding scheme to the default CCSID of the current server. In this case, the CCSID assigned to the constant is the default CCSID of the current server.

The CCSID of the source is determined by the application requester. The CCSID of the source is:

- For STRSQL, the default CCSID of the application requester
- For the RUNSQLSTM or STRREXPRC commands, the CCSID of the specified source file
- For CRTSQLxxx:
 - For static SQL, the CCSID of the source is the CCSID of the source file used on the CRTSQLxxx command.
 - For dynamic SQL, the CCSID of the source is the CCSID of the host variable specified on the PREPARE statement, or if a string constant is specified on the PREPARE statement, the default CCSID of the current server.

Examples

```
'Peggy'      '14.12.1990'  '32'      'DON'T CHANGE'  ''      X'FFFF'
```

Graphic-String Constants

DBCS Graphic-String Constants

A *graphic-string constant* is a varying-length graphic string. The length of the specified string cannot be greater than 16370. The three forms of DBCS graphic-string constants are:

Context	Graphic String Constant	Empty String	Example
All contexts	G ' ₀ dbcs-string ₁ '	G ' ₀ ₁ ' G '' g ' ₀ ₁ ' g ''	G ' ₀ 元 気 ₁ '
	N ' ₀ dbcs-string ₁ '	N ' ₀ ₁ ' N '' n ' ₀ ₁ ' n ''	
PL/I	₀ 'dbcs-string 'G ₁	₀ ''G ₁	₀ '元 気 'G ₁

RV3F000-0

In the normal form, the SQL delimiters and the G or the N are SBCS characters. The SBCS ' is the EBCDIC apostrophe, X'7D'.

In the PL/I form, the apostrophes and the G are DBCS characters. Two consecutive DBCS string delimiters are used to represent one string delimiter within the string. Note that this PL/I form is only valid for static statements embedded in PL/I programs.

A hexadecimal DBCS graphic constant is also supported. The form of the hexadecimal DBCS graphic constant is:

GX'ssss'

In the constant, **sss** represents a string from 0 to 32766 hexadecimal digits. The number of characters between the string delimiters must be an even multiple of 4. Each group of 4 digits represents a single DBCS graphic character. The hexadecimal for shift-in and shift-out ('0E'X and '0F'X) are not included in the string.

The CCSID assigned to constants is the DBCS CCSID associated with the CCSID of the source unless the source is encoded in a foreign encoding scheme (such as ASCII). In this case, the CCSID assigned to the constant is the DBCS CCSID associated with the default CCSID of the current server when the SQL statement containing the constant is prepared. If there is no DBCS CCSID associated with the CCSID of the source, the CCSID is 65535.

For information on associated DBCS CCSIDs, see the Globalization DBCS CCSIDs topic in the iSeries Information Center. For information on the CCSID of the source, see Character String Constants.

UCS-2 Graphic-String Constants

A hexadecimal UCS-2 graphic constant is supported. The form of the hexadecimal UCS-2 graphic constant is:

UX'ssss'

Constants

In the constant, *ssss* represents a string from 0 to 32766 hexadecimal digits. The number of characters between the string delimiters must be an even multiple of 4. Each group of 4 digits represents a single UCS-2 graphic character.

The CCSID of a UCS-2 constant is 13488.

Binary-String Constants

A *binary-string constant* specifies a varying-length binary string. The form of a binary-string constant follows:

- An X followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 32740. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase).

The CCSID assigned to the constant is 65535.

Note that the syntax of a binary string constant is identical to the second form of a character constant. A constant of this form is only treated as a binary string constant if the SET OPTION statement was specified with the binary string option (SQLCURRULE = *STD) or the SQLCURRULE(*STD) parameter on the CRTSQLxxx command.

Example

```
X'FFFF'
```

Decimal Point

The default decimal point can be specified:

- To interpret numeric constants
- To determine the decimal point character to use when casting a character string to a number (for example, in the DECIMAL, DOUBLE_PRECISION, FLOAT, and REAL scalar functions and the CAST expression)
- to determine the decimal point character to use in the result when casting a number to a string (for example, in the CHAR, CLOB, and VARGRAPHIC scalar functions and the CAST expression)

The default decimal point can be specified through the following interfaces:

Table 20. Default Decimal Point Interfaces

SQL Interface	Specification
Embedded SQL	The *JOB, *PERIOD, *COMMA, or *SYSVAL value in the OPTION parameter is specified on the Create SQL Program (CRTSQLxxx) commands. The SET OPTION statement can also be used to specify the DECMPT parameter within the source of a program containing embedded SQL. (For more information about CRTSQLxxx commands, see the SQL Programming with Host Languages book.)

Table 20. Default Decimal Point Interfaces (continued)

SQL Interface	Specification
Interactive SQL and Run SQL Statements	The DECPNT parameter on the Start SQL (STRSQL) command or by changing the session attributes. The DECMPT parameter on the Run SQL Statements (RUNSQLSTM) command. (For more information about STRSQL and RUNSQLSTM commands, see the SQL Programming Concepts book.)
Call Level Interface (CLI) on the server	SQL_ATTR_DATE_FMT and SQL_ATTR_DATE_SEP environment or connection variables (For more information about CLI, see the SQL Call Level Interfaces (ODBC) book.)
JDBC or SQLJ on the server using Developer Kit for Java	Decimal Separator conneciton property (For more information about JDBC and SQLJ, see the IBM Developer Kit for Java topic in the iSeries Information Center.)
ODBC on a client using the iSeries Access ODBC Driver	Decimal Separator in the Advanced Server Options in ODBC Setup (For more information about ODBC, see the iSeries Access category in the iSeries Information Center.)
JDBC on a client using the IBM Toolbox for Java	Format in JDBC Setup (For more information about ODBC, see the iSeries Access category in the iSeries Information Center.) (For more information about the IBM Toolbox for Java, see IBM Toolbox for Java topic in the iSeries Information Center .)

If the comma is the decimal point, the following rules apply:

- A period will also be allowed as a decimal point.
- A comma intended as a separator of numeric constants in a list must be followed by a space.
- A comma intended as a decimal point must not be followed by a space.

Thus, to specify a decimal constant without a fractional part, the trailing comma must be followed by a non-blank character. The non-blank character can be a separator comma, as in:

```
VALUES(999999999, , 111)
```

Delimiters

*APOST and *QUOTE are mutually exclusive COBOL precompiler options that name the string delimiter within COBOL statements. *APOST names the apostrophe (') as the string delimiter; *QUOTE names the quotation mark ("). *APOSTSQL and *QUOTESQL are mutually exclusive COBOL precompiler options that play a similar role for SQL statements embedded in COBOL programs. *APOSTSQL names the apostrophe (') as the SQL string delimiter; with this option, the quotation mark (") is the SQL escape character. *QUOTESQL names the quotation mark as the SQL string delimiter; with this option, the apostrophe is the SQL escape character. The values of *APOSTSQL and *QUOTESQL are respectively the same as the values of *APOST and *QUOTE.

In host languages other than COBOL, the usages are fixed. The string delimiter for the host language and for static SQL statements is the apostrophe ('); the SQL

Constants

escape character is the quotation mark (").

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. A reference to a special register is a reference to a value provided by the current server. If the value is a string, its CCSID is a default CCSID of the current server. DB2 UDB for iSeries includes the following special registers.

CURRENT DATE or CURRENT_DATE

The CURRENT DATE special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the current server. If this special register is used more than once within a single SQL statement, or used with CURRENT TIME, CURRENT TIMESTAMP, or the CURDATE, CURTIME, or NOW scalar functions 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 PATH, CURRENT_PATH, or CURRENT FUNCTION PATH

The CURRENT PATH special register specifies the SQL path used to resolve unqualified distinct type names (both built-in types and distinct types), procedure names, and function names in dynamically prepared SQL statements. It is also used to resolve unqualified procedure names that are specified as host variables in SQL CALL statements (CALL host-variable). The data type is VARCHAR(3483).

The CURRENT PATH special register contains a list of one or more schema names, where each schema name is enclosed in delimiters and separated from the following schema by a comma. The delimiters and commas are included in the 3483 character length. The maximum number of schema names in the path is 268.

For information on when the SQL path is used to resolve unqualified names in both dynamic and static SQL statements and the effect of its value, see “SQL Path” on page 53.

The initial value of the CURRENT PATH special register in an activation group is established by the first SQL statement that is executed.

- If the first SQL statement in an activation group is executed from an SQL program or SQL package and the SQLPATH parameter was specified on the CRTSQLxxx command, the path is the value specified in the SQLPATH parameter. The SQLPATH value can also be specified using the SET OPTION statement.
- Otherwise,
 - For SQL naming, “QSYS”, “QSYS2”, “the value of the authorization ID of the statement” .
 - For system naming, “*LIBL”.

You can change the value of the register by executing the statement SET PATH. For details about this statement, see “SET PATH” on page 747.

Special Registers

Example

Set the special register so that schema SMITH is searched before schemas QSYS and QSYS2 (SYSTEM PATH).

```
SET CURRENT PATH SMITH, SYSTEM PATH
```

CURRENT SCHEMA

The CURRENT SCHEMA special register specifies a VARCHAR(128) value that identifies the schema name used to qualify unqualified database object references where applicable in dynamically prepared SQL statements.²² CURRENT SCHEMA is not used to qualify names in programs where the DYNDFTCOL has been specified. If DYNDFTCOL is specified in a program, its schema name is used instead of the CURRENT SCHEMA schema name.

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

The DFTRDBCOL keyword controls the schema name used to qualify unqualified database object references where applicable for static SQL statements.

Example

Set the schema for object qualification to 'D123'.

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

CURRENT SERVER or CURRENT_SERVER

The CURRENT SERVER special register specifies a VARCHAR(18) value that identifies the current server.

CURRENT SERVER can be changed by the CONNECT (Type 1), CONNECT (Type 2), or SET CONNECTION statements, but only under certain conditions. See the description in "CONNECT (Type 1)" on page 416, "CONNECT (Type 2)" on page 421, and "SET CONNECTION" on page 730.

CURRENT SERVER cannot be specified unless the local relational database is named by adding the entry to the relational database directory using the ADDRDBDIRE or WRKRDBDIRE command.

Example

Set the host variable APPL_SERVE (VARCHAR(18)) to the name of the current server.

```
SELECT CURRENT SERVER  
  INTO :APPL_SERVE  
  FROM SYSDUMMY1
```

CURRENT TIME or CURRENT_TIME

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

22. For compatibility with DB2 UDB for OS/390 and z/OS, the special register CURRENT SQLID is treated as a synonym for CURRENT SCHEMA.

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 or CURRENT_TIMESTAMP

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

Example

Insert a row into the IN_TRAY sample 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 or CURRENT_TIMEZONE

The CURRENT_TIMEZONE special register specifies the difference between UTC²³ and local time at the current 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.

Example

Using the IN_TRAY table, select all the rows from the table and adjust the value to UTC.

```
SELECT RECEIVED - CURRENT_TIMEZONE, SOURCE,
SUBJECT, NOTE_TEXT FROM IN_TRAY
```

USER

The USER special register specifies the run-time authorization ID at the current server. The data type of the special register is VARCHAR(18).

Example

Select all notes from the IN_TRAY table that the user placed there himself.

```
SELECT * FROM IN_TRAY
WHERE SOURCE = USER
```

23. Coordinated Universal Time, formerly known as GMT.

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 4, “Queries” on page 329. 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.

Qualified Column Names

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

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

- In the COMMENT and LABEL statements, the column name must be qualified.
- Where the column name specifies values of the column, a column name can be qualified at the user’s option.
- In all other contexts, a column name must not be qualified.

Where a qualifier is optional it can serve two purposes. See “Column Name Qualifiers to Avoid Ambiguity” on page 110 and “Column Name Qualifiers in Correlated References” on page 111 for details.

Correlation Names

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

```
FROM X.MYTABLE Z
```

A correlation name is associated with a table, view, or alias 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. A correlation name can also be used as a shorter name for a table, view, or alias. In the example that is shown above, 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, any qualified reference to a column of that instance of the table, view or alias must use the correlation name, rather than the table name, view name, 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:

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

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 correlation name is always an exposed name. A table name, view name, or alias name is said to be *exposed* in that FROM clause if a correlation name is not specified. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE but not for DEPARTMENT, so DEPARTMENT is an exposed name, and EMPLOYEE is not:

```
FROM EMPLOYEE E, DEPARTMENT
```

A table name, view name, or alias name that is exposed in a FROM clause must not be the same as any other table name or view name exposed in that FROM clause or any correlation name in the FROM clause. The names are compared after qualifying any unqualified table or view names.

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                               ***INCORRECT***
```

the two exposed table names included in this clause (EMPLOYEE and EMPLOYEE) are the same, and this is not allowed.

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).

Column Names

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). This FROM clause is only valid if the authorization ID of the statement is not X.

A correlation name specified in a FROM clause must not be the same as:

- Any other correlation name in that FROM clause
- Any unqualified table name or view name exposed in the FROM clause

For example, the following FROM clauses are incorrect:

```
FROM EMPLOYEE E, EMPLOYEE E
FROM EMPLOYEE DEPARTMENT, DEPARTMENT      ***INCORRECT***
FROM X.T1, EMPLOYEE T1
```

The following FROM clause is technically correct, though potentially confusing:

```
FROM EMPLOYEE DEPARTMENT, DEPARTMENT EMPLOYEE
```

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.

If a list of columns is specified, it must consist of as many names as there are columns in the table-reference. Each column name must be unique and unqualified.

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 or view. The tables and views that might contain the column are called the *object tables* of the context. Two or more object tables might contain columns with the same name. One reason for qualifying a column name is to designate the object from which the column comes.

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 or view name 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.
- In SQL naming, 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.
- In system naming, the table designator for an exposed table or view name is the unqualified table or view name. In the following example MYTABLE is the table designator for OWNY/MYTABLE.

```
SELECT CORZ.COLA, MYTABLE.COLA
FROM OWNX/MYTABLE CORZ, OWNY/MYTABLE
```

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.

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 object table names can be used as designators.

Two or more object tables can be instances of the same table. In this case, distinct correlation names must be used to unambiguously designate the particular instances of the table. In the following FROM clause, X and Y are defined to refer, respectively, to the first and second instances of the table CORPDATA.EMPLOYEE:

```
FROM CORPDATA.EMPLOYEE X, CORPDATA.EMPLOYEE Y
```

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, then:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
FROM EMPLOYEE
```

is a valid statement.

2. If the authorization ID of the statement is REGION, then:

```
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 *subselect* is a form of a query that can be used as a component of various SQL statements. Refer to Chapter 4, “Queries” on page 329 for more information about

Column Names

subselects. A *subquery* is a form of a fullselect that is enclosed within parenthesis. For example, a *subquery* can be used in a search condition.

A subquery can include search conditions of its own, and these search conditions can, in turn, include subqueries. Therefore, an SQL statement can contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are said to be at a higher level than the subqueries they contain.

Every element of the hierarchy has a clause that establishes one or more table designators. This is the FROM clause, except in the highest level of an UPDATE or DELETE statement. A search condition, the select list, the join clause, or an argument of a table function in a subquery can reference not only columns of the tables identified by the FROM clause of its own element of the hierarchy, but also columns of tables identified at any level along the path from its own element to the highest level of the hierarchy. A reference to a column of a table identified at a higher level is called a *correlated reference*.

A correlated reference to column C of table T can be of the form C, T.C, or Q.C, if Q is a correlation name defined for T. However, a correlated reference in the form of an unqualified column name is not good practice. The following explanation is based on the assumption that a correlated reference is always in the form of a qualified column name and that the qualifier is a correlation name.

Q.C is a correlated reference only if these three conditions are met:

- Q.C is used in a search condition, select list, join clause, or an argument of a table function in a subquery.
- Q does not designate a table used in the FROM clause of that subquery, select list, join clause, or an argument of a table function in a subquery.
- Q does designate a table used at some higher level.

Q.C refers to column C of the table or view at the level where Q is used as the table designator of that table or view. Because the same table or view can be identified at many levels, unique correlation names are recommended as table designators. If Q is used to designate a table at more than one level, Q.C refers to the lowest level that contains the subquery that includes Q.C.

In the following statement, Q is used as a correlation name for T1 and T2, but Q.C refers to the correlation name associated with T2, because it is the lowest level that contains the subquery that includes Q.C.

```
SELECT *
  FROM T1 Q
 WHERE A < ALL (SELECT B
                FROM T2 Q
                WHERE B < ANY (SELECT D
                              FROM T3
                              WHERE D = Q.C))
```

Unqualified Column Names

An unqualified column name can also be a correlated reference if the column:

- Is used in a search condition of a subquery
- Is not contained in a table used in the FROM clause of that subquery
- Is contained in a table used at some higher level

Unqualified correlated references are not recommended because it makes the SQL statement difficult to understand. The column will be implicitly qualified when the statement is prepared depending on which table the column was found in. Once

this implicit qualification is determined it will not change until the statement is re-prepared. An SQL precompiler issues a warning message in the precompile listing and the database manager issues a positive SQLCODE (+12) and SQLSTATE (01545) when an SQL statement that has an unqualified correlated reference is prepared or executed.

References to Variables

A *variable* in an SQL statement specifies a value that can be changed when the SQL statement is executed. There are several types of *variables* used in SQL statements:

host variable

Host variables are defined by statements of a host language. For more information about how to refer to host variables see “References to Host Variables” on page 114.

transition variable

Transition variables are defined in a trigger and refer to either the old or new values of columns. For more information about how to refer to transition variables see “CREATE TRIGGER” on page 556.

SQL variable

SQL variables are defined by an SQL compound statement in an SQL function, SQL procedure, or trigger. For more information about SQL variables, see “References to SQL Parameters and Variables” on page 781.

SQL parameter

SQL parameters are defined in an CREATE FUNCTION (SQL Scalar), CREATE FUNCTION (SQL Table), or CREATE PROCEDURE (SQL) statement. For more information about SQL variables, see “References to SQL Parameters and Variables” on page 781.

parameter marker

Variables cannot be referenced in dynamic SQL statements. Parameter markers are defined in an SQLDA and used instead. For more information about parameter markers, see “Parameter Markers” on page 694.

In this book, unless otherwise noted, the term *host variable* in syntax diagrams is used to describe where a host variable, transition variable, SQL variable, SQL parameter, or parameter marker can be used.

References to Host Variables

A *host variable* is a COBOL data item, an RPG field, or a PLI, REXX, C++, or C variable that is referenced in an SQL statement. Host variables are defined by statements of the host language. For more information about how to refer to host structures in C, C++, COBOL, PL/I, and RPG, see “Host Structures” on page 120. For more information about host variables in REXX, see the SQL Programming with Host Languages book.

A *host-variable* in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

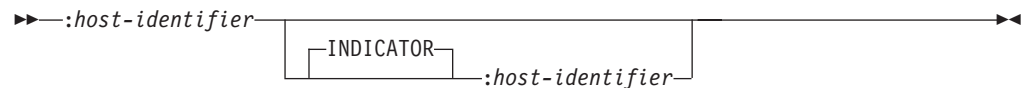
All host variables used in an SQL statement should be declared in an SQL declare section in all host languages other than Java, REXX, and RPG. (Variables do not have to be declared in REXX. In Java and RPG, there is no declare section, and host variables may be declared throughout the program.) No variables 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.

For further information about using host variables, see the SQL Programming Concepts book.

References to Host Variables

The term *host-variable*, as used in the syntax diagrams, shows a reference to a host variable. A *host-variable* in the INTO clause of a FETCH, SELECT INTO, SET variable, or VALUES INTO statement identifies a host variable to which a value from a column of a row is assigned. A host variable in a CALL statement or in an EXECUTE statement identifies either or both a host variable to which an output parameter value is assigned, and a host variable that specifies an input argument value to be passed to the database manager from the application program. In all other contexts a *host-variable* specifies a value to be passed to DB2 UDB for iSeries from the application program.

The general form of a *host-variable* reference in all languages other than Java is:



Each *host-identifier* must be declared in the source program. The variable designated by the second *host-identifier* is called an *indicator variable* and must have a data type of *small integer*.

The purposes of the indicator variable are to:

- Specify the null value. A negative value of the indicator variable specifies the null value.
- Indicate one of the following data mapping errors:
 - Characters could not be converted
 - Numeric conversion error (underflow or overflow)
 - Arithmetic expression error (division by 0)
 - Date or timestamp conversion error (a date or timestamp that is not within the valid range of the dates for the specified format)
 - String representation of the datetime value is not valid
 - Mixed data not properly formed
 - A numeric value that is not valid
 - Argument of SUBSTR scalar function is out of range
- Record the original length of a truncated string.
- Record the seconds portion of a time if the time is truncated on assignment to a host variable.

For example, if :V1:V2 is used to specify an insert or update value, and if V2 is negative, the value specified is the null value. If V2 is not negative the value specified is the value of V1.

Similarly, if :V1:V2 is specified in a CALL, FETCH, SELECT INTO, or VALUES INTO statement and the value returned is null, V1 is undefined, and V2 is set to a negative value. The negative value is:

- -1 if the value selected was the null value, or
- -2 if the null value was returned due to data mapping errors in the select list of an outer subselect.²⁴

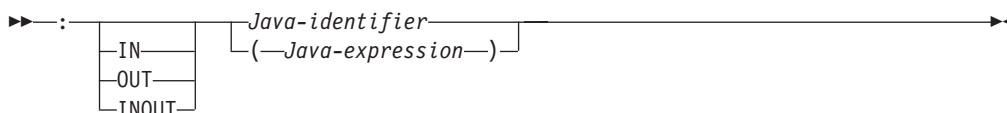
24. It should be noted that although the null value returned for data mapping errors can be returned on certain scalar functions and for arithmetic expressions, the result column is not considered null capable unless an argument of the arithmetic expression or scalar function is null capable.

References to Host Variables

If the value returned is not null, that value is assigned to V1 and V2 is set to zero (unless the assignment to V1 requires string truncation, in which case, V2 is set to the original length of the string). If an assignment requires truncation of the seconds part of time, V2 is set to the number of seconds.

If the second *host-identifier* is omitted, the *host variable* does not have an indicator variable. The value specified by the *host-variable* :V1 is always the value of V1, and null values cannot be assigned to the variable. Thus, this form should not be used unless the corresponding result column cannot contain null values. If this form is used and the column contains nulls, the database manager returns a negative value (-407) in the SQLCODE field of the SQLCA. If your data is truncated and there is no indicator variable, no error condition results.

The general form of a *host-variable* reference in Java is:



In Java, indicator variables are not used. Instead, instances of a Java class can be set to a null value. Variables defined as Java primitive types can not be set to a null value.

If IN, OUT, or INOUT is not specified, the default depends on the context in which the variable is used. If the Java variable is used in an INTO clause, OUT is the default. Otherwise, IN is the default. For more information on Java variables, see Developer Kit for Java.

In C, C++, ILE RPG, and PL/I, 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.

The CCSID of a string host variable is either:

- The CCSID specified in the DECLARE VARIABLE statement, or
- If a DECLARE VARIABLE with a CCSID clause is not specified for the host variable, the default CCSID of the application requester at the time the SQL statement that contains the host variable is executed unless the CCSID is for a foreign encoding scheme (such as ASCII). In this case, the host variable is converted to the default CCSID of the current server.

Example

Using the PROJECT table, set the host variable PNAME (VARCHAR(26)) to the project name (PROJNAME), the host variable STAFF (DECIMAL(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'
```


Host Variables in Dynamic SQL

In dynamic SQL statements, parameter markers are used instead of host variables. A parameter marker is a question mark (?) that represents a position in a dynamic SQL statement where the application will provide a value; that is, where a host variable would be found if the statement string were a static SQL statement. The following examples shows a static SQL that uses host variables and a dynamic statement that uses parameter markers:

```
INSERT INTO DEPT
VALUES( :HV_DEPTNO, :HV_DEPTNAME, :HV_MGRNO:IND_MGRNO, :HV_ADMRDEPT)

INSERT INTO DEPT
VALUES( ?, ?, ?, ? )
```

For more information about parameter markers, see “Parameter Markers” on page 694.

References to LOB Host Variables

Regular LOB variables, LOB locator variables (see “References to LOB Locator Variables”) and LOB file reference variables (see “References to LOB File Reference Variables” on page 118), can be defined in the following host languages:

- C
- C++
- ILE RPG
- ILE COBOL
- PL/I

Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these variables are not native data types in host programming languages, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

When it is possible to define a host variable that is large enough to hold an entire LOB value and the performance benefit of delaying the transfer of data from the server is not required, a LOB locator is not needed. However, it is often not acceptable to store an entire LOB value in temporary storage due to host language restrictions, storage restrictions, or performance requirements. When storing a entire LOB value at one time is not acceptable, a LOB value can be referred to by a LOB locator and portions of the LOB value can be accessed.

Like all other host variables, a LOB locator variable or LOB file reference variable can have an associated indicator variable. Indicator variables for LOB locator variables and LOB file reference 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 host variable is unchanged. This means that a locator can never point to a null value.

References to LOB Locator Variables

A LOB *locator variable* is a host variable that contains the locator representing a LOB value on the server, which can be defined in the following host languages:

- C
- C++
- ILE RPG

References to Host Variables

- ILE COBOL
- PL/I

See “Manipulating Large Objects (LOBs) With Locators” on page 66 for information on how locators can be used to manipulate LOB values.

A locator variable in an SQL statement must identify a LOB locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement. For example, in C:

```
static volatile SQL TYPE IS CLOB_LOCATOR *loc1;
```

The term *locator-variable*, as used in the syntax diagrams, shows a reference to a LOB locator variable. The meta-variable *locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

When the indicator variable associated with a LOB locator is null, the value of the referenced LOB is null.

If a locator variable does not currently represent any value, an error occurs when the locator variable is referenced.

At transaction commit or any transaction termination, all LOB locators that were acquired by the transaction are released.

It is the application programmer’s responsibility to guarantee that any LOB locator is only used in SQL statements that are executed at the same server that originally generated the LOB locator. For example, assume that a LOB locator is returned from one server and assigned to a LOB locator variable. If that LOB locator variable is subsequently used in an SQL statement that is executed at a different server, unpredictable results will occur.

References to LOB File Reference Variables

A LOB *file reference variable* is used for direct file input and output for a LOB, which can be defined in the following host languages:

- C
- C++
- ILE RPG
- ILE COBOL
- PL/I

Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

A file reference variable represents (rather than contains) the file, just as a LOB locator represents, rather than contains, the LOB data. Database queries, updates, and inserts may use file reference variables to store or to retrieve single column values. The file referenced must exist at the application requester.

As with all other host variables, a file reference variable may have an associated indicator variable.

The length attribute of a file reference variable is assumed to be the maximum length of a LOB.

References to Host Variables

File reference variables are currently supported in the root (/), QOpenSys, and UDFS file systems. When a file is created, it is given the CCSID of the data that is being written to the file. Currently, mixed CCSIDs are not supported. To use a file created with a file reference variable, the file should be opened in binary mode.

For more information about file reference variables, see the SQL Programming Concepts book.

Host Structures

A host structure is a COBOL group, PL/I, C, or C++ structure, or RPG data structure that is referenced in an SQL statement. Host structures are defined by statements of the host language, as explained in the SQL Programming with Host Languages book. As used here, the term host structure does not include an SQLCA or SQLDA.

The form of a host structure reference is identical to the form of a host variable reference. The reference :S1:S2 is a host structure reference if S1 names a host structure. If S1 designates a host structure, S2 must be either a small integer variable, or an array of small integer variables. S1 is the host structure and S2 is its indicator array.

A host structure can be referenced in any context where a list of host variables can be referenced. A host structure reference is equivalent to a reference to each of the host variables contained within the structure in the order which they are defined in the host language structure declaration. The *n*th variable of the indicator array is the indicator variable for the *n*th variable of the host structure.

In C, for example, if V1, V2, and V3 are declared as variables within the structure S1, the statement:

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

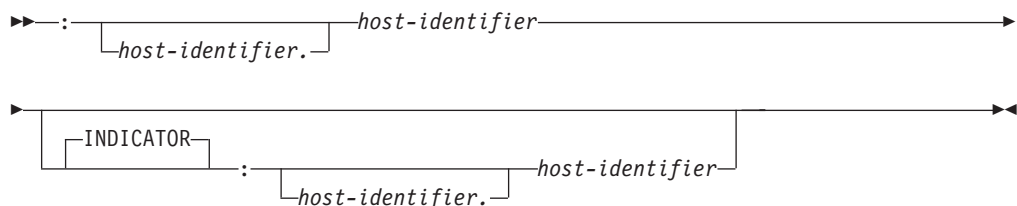
is equivalent to:

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

If the host structure has *m* more variables than the indicator array, the last *m* variables of the host structure do not have indicator variables. If the host structure has *m* fewer variables than the indicator array, the last *m* variables of the indicator array are ignored. These rules also apply if a reference to a host structure includes an indicator variable or if a reference to a host variable includes an indicator array. If an indicator array or indicator variable is not specified, no variable of the host structure has an indicator variable.

In addition to structure references, individual host variables in the host structure or indicator variables in the indicator array can be referenced by qualified names. The qualified form is a host identifier followed by a period and another host identifier. The first host identifier must name a host structure, and the second host identifier must name a host variable within that host structure.

The general form of a host variable or host structure reference is:



A *host-variable* in an expression must identify a host variable (not a structure) described in the program according to the rules for declaring host variables.

The following C example shows a references to host structure, host indicator array, and a host variable:

```

struct { char empno[7];
        struct
            { short int firstname_len;
              char firstname_text[12];
            } firstname;
        char midint,
        struct
            { short int lastname_len;
              char lastname_text[15];
            } lastname;
        char workdept[4];
    } pemp1;
short ind[14];
short eind
struct { short ind1;
        short ind2;
    } indstr;

.....
strcpy("000220",pemp1.empno);
.....
EXEC SQL
  SELECT *
  INTO :pemp1:ind
  FROM corpdata.employee
  WHERE empno=:pemp1.empno;

```

In the example above, the following references to host variables and host structures are valid:

```
:pemp1      :pemp1.empno      :pemp1.empno:eind      :pemp1.empno:indstr.ind1
```

Host Structure Arrays

Host Structure Arrays

In PL/I, C++, and C, a host structure array is a structure name having a dimension attribute. In COBOL, it is a one-dimensional table. In RPG, it is an occurrence data structure. A host structure array can only be referenced in the FETCH statement when using a multiple-row fetch, or in an INSERT statement when using a blocked insert. Host structure arrays are defined by statements of the host language, as explained in the SQL Programming with Host Languages book.

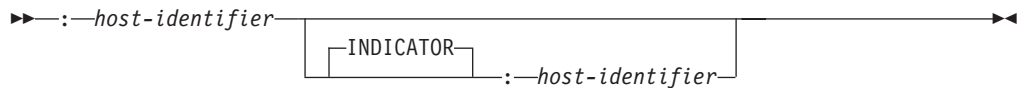
The form of a host structure array is identical to the form of a host variable reference. The reference :S1:S2 is a reference to host structure array if S1 names a host structure array. If S1 designates a host structure, S2 must be either a small integer host variable, an array of small integer host variables, or a two dimensional array of small integer host variables. In the following example, S1 is the host structure array and S2 is its indicator array.

```
EXEC SQL FETCH CURSOR1 FOR 5 ROWS  
      INTO :S1:S2;
```

The dimension of the host structure and the indicator array must be equal.

If the host structure has m more variables than the indicator array, the last m variables of the host structure do not have indicator variables. If the host structure has m fewer variables than the indicator array, the last m variables of the indicator array are ignored. If an indicator array or variable is not specified, no variable of the host structure array has an indicator variable.

The following diagram specifies the syntax of references to an array of host structures:



Arrays of host structures are not supported in REXX.

Functions

A *function* is an operation denoted by a function name followed by one or more operands that are enclosed in parentheses. It represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*. For example, a function can be passed two input arguments that have date and time data types and return a value with a timestamp data type as the result.

Types of Functions

There are several ways to classify functions. One way to classify functions is as built-in, user-defined, or generated user-defined functions for distinct types.

- *Built-in functions* are IBM-supplied functions that come with the database manager. These functions provide a single-value result. Built-in functions include operator functions such as "+", column functions such as AVG, and scalar functions such as SUBSTR. For a list of the built-in column and scalar functions and information on these functions, see Chapter 3, "Built-In Functions" on page 163.²⁵
- *User-defined functions* are functions that are created using the CREATE FUNCTION statement and registered to the database manager in catalog table QSYS2.SYSROUTINES and catalog view QSYS2.SYSFUNCS. These functions allow users to extend the function of the database manager by adding their own or third party vendor function definitions.

A user-defined function is either *SQL*, *external*, or *sourced*. An SQL function is defined to the database using only SQL statements. An external function is defined to the database with a reference to an external program or service program that is executed when the function is invoked. A sourced function is defined to the database with a reference to a built-in function or another user-defined function. Sourced functions can be used to extend built-in column and scalar functions for use on distinct types.

A user-defined function resides in the schema in which it was created. The schema cannot be QSYS, QSYS2, or QTEMP.

- The database manager automatically generates some user-defined functions when a distinct type is created using the CREATE DISTINCT TYPE statement. These functions support casting from the distinct type to the source type and from the source type to the distinct type. The ability to cast between the data types is important because a distinct type is compatible only with itself.

The generated cast functions reside in the same schema as the distinct type for which they were created. The schema cannot be QSYS, QSYS2, or QTEMP. For more information about the functions that are generated for a distinct type, see "CREATE DISTINCT TYPE" on page 428.

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

A *column function* receives a set of values for each argument (such as the values of a column) and returns a single-value result for the set of input values. Column functions are sometimes called aggregating functions. Built-in functions and user-defined sourced functions can be column functions.

²⁵ *Built-in functions* are implemented internally by the database manager, so an associated program or service program object does not exist for a *built-in function*. Furthermore, the catalog does not contain information about *built-in functions*. However, *built-in functions* can be treated as if they exist in QSYS2 and a *built-in function* name can be qualified with QSYS2.

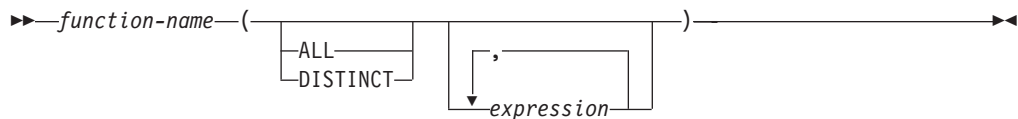
Functions

A *scalar function* receives a single value for each argument and returns a single-value result. Built-in functions and user-defined functions can be scalar functions. The functions that are created for distinct types are also scalar functions.

A *table function* returns a table for the set of arguments it receives. Each argument is a single value. A table function can only be referenced in the FROM clause of a subselect. A table function can be defined as an external function or as an SQL function (a table function cannot be a sourced function.).

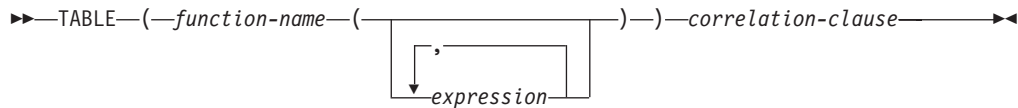
Table functions can be used to apply SQL language processing power to data that is not DB2 data or to convert such data into a DB2 table. For example, a table function can take a file and convert it to a table, get data from the World Wide Web and tabularize it, or access a Lotus Notes database and return information about mail messages.

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



The `ALL` or `DISTINCT` keyword can only be specified for a column function or a user-defined function that is sourced on a column function.

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



In the above syntax, *expression* is the same as it is for a scalar or column function. For more details on referencing a table function, see the description of the FROM clause on “from-clause” on page 335.

Function resolution

A function is invoked by its function name, which is implicitly or explicitly qualified with a schema name, followed by parentheses that enclose the arguments to the function. Within the database, each function is uniquely identified by its function signature, which is its schema name, function name, the number of parameters, and the data types of the parameters. Thus, a schema can contain several functions that have the same name but each of which have a different number of parameters, or parameters with different data types. Or, a function with the same name, number of parameters, and types of parameters can exist in multiple schemas. When any function is invoked, the database manager must determine which function to execute. This process is called *function resolution*.

Function resolution is similar for functions that are invoked with a qualified or unqualified function name with the exception that for an unqualified name, the database manager needs to search more than one schema.

Qualified function resolution: When a function is invoked with a function name and a schema name, the database manager only searches the specified schema to

resolve which function to execute. The database manager finds the appropriate function instance when all of the following conditions are true:

- The name of the function instance matches the name in the function invocation.
- The number of input parameters in the function instance matches the number of arguments in the function invocation.
- The data type of each input argument of the function invocation matches or is *promotable* to the data type of the corresponding parameter of the function instance.

This comparison of data types results in one best fit, which is the choice for execution (see “Determining the Best Fit”). For information on the promotion of data types, see “Promotion of Data Types” on page 75.

If no function in the schema meets these criteria, an error occurs.

Unqualified function resolution: When a function is invoked with only a function name, the database manager needs to search more than one schema to resolve the function instance to execute. The SQL path contains the list of schemas to search. For each schema in the path (for information on paths see “SQL Path” on page 53), the database manager selects a candidate function based on the following criteria:

- The name of the function instance matches the name in the function invocation.
- The number of input parameters in the function instance matches the number of function arguments in the function invocation.
- The data type of each input argument of the function invocation matches or is *promotable* to the data type of the corresponding parameter of the function instance.

This comparison of data types results in one best fit, which is the choice for execution (see “Determining the Best Fit”). For information on the promotion of data types, see “Promotion of Data Types” on page 75.

If no function in the schema meets these criteria, an error occurs.

A candidate function is not selected for a schema if one or more of the criteria is not met.

After the database manager identifies the candidate functions, it selects the candidate with the best fit as the function instance to execute (see “Determining the Best Fit”). If more than one schema contains the function instance with the best fit (the function signatures are identical except for the schema name), the database manager selects the function whose schema is earliest in the SQL path.

Function resolution applies to all functions, including built-in functions. Built-in functions logically exist in schema QSYS2. If schema QSYS2 is not explicitly specified in the SQL path, the schema is implicitly assumed at the front of the path. Therefore, when an unqualified function name is specified, ensure that the path is specified so that the intended function is selected.

Determining the Best Fit

There might be more than one function with the same name that is a candidate for execution. In that case, the database manager determines which function is the best fit for the invocation by comparing the argument and parameter data types. Note that neither the data type of the result of the function nor the type of function (column or scalar) under consideration enters into this determination.

If the data types of all the parameters for a given function are the same as those of the arguments in the function invocation, that function is the best fit. If there is no

Functions

exact match, the database manager compares the data types in the parameter lists from left to right, using the following method:

1. Compare the data type of the first argument in the function invocation to the data type of the first parameter in each function. (Any length, precision, scale, and CCSID attributes of the data types are not considered in the comparison.)
2. For this argument, if one function has a data type that fits the function invocation better than the data types in the other functions, that function is the best fit. The precedence list for the promotion of data types in "Promotion of Data Types" on page 75 shows the data types that fit each data type in best-to-worst order.
3. If the data type of the first parameter for more than one candidate function fits the function invocation equally well, repeat this process for the next argument of the function invocation. Continue for each argument until a best fit is found.

The following examples illustrate function resolution.

Example 1: Assume that MYSCHEMA contains two functions, both named FUNA, that were created with these partial CREATE FUNCTION statements.

```
CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), INT, DOUBLE) ...  
CREATE FUNCTION MYSCHEMA.FUNA (VARCHAR(10), REAL, DOUBLE) ...
```

Also assume that a function with three arguments of data types VARCHAR(10), SMALLINT, and DECIMAL is invoked with a qualified name:

```
MYSCHEMA.FUNA( VARCHARCOL, SMALLINTCOL, DECIMALCOL ) ...
```

Both MYSCHEMA.FUNA functions are candidates for this function invocation because they meet the criteria specified in "Function resolution" on page 124. The data types of the first parameter for the two function instances in the schema, which are both VARCHAR, fit the data type of the first argument of the function invocation, which is VARCHAR, equally well. However, for the second parameter, the data type of the first function (INT) fits the data type of the second argument (SMALLINT) better than the data type of second function (REAL). Therefore, the database manager selects the first MYSCHEMA.FUNA function as the function instance to execute.

Example 2: Assume that functions were created with these partial CREATE FUNCTION statements:

1. **CREATE FUNCTION** SMITH.ADDIT (**CHAR**(5), **INT**, **DOUBLE**) ...
2. **CREATE FUNCTION** SMITH.ADDIT (**INT**, **INT**, **DOUBLE**) ...
3. **CREATE FUNCTION** SMITH.ADDIT (**INT**, **INT**, **DOUBLE**, **INT**) ...
4. **CREATE FUNCTION** JOHNSON.ADDIT (**INT**, **DOUBLE**, **DOUBLE**) ...
5. **CREATE FUNCTION** JOHNSON.ADDIT (**INT**, **INT**, **DOUBLE**) ...
6. **CREATE FUNCTION** TODD.ADDIT (**REAL**) ...
7. **CREATE FUNCTION** TAYLOR.SUBIT (**INT**, **INT**, **DECIMAL**) ...

Also assume that the SQL path at the time an application invokes a function is "TAYLOR", "JOHNSON", "SMITH". The function is invoked with three data types (INT, INT, DECIMAL) as follows:

```
SELECT ... ADDIT(INTCOL1, INTCOL2, DECIMALCOL) ...
```

Function 5 is chosen as the function instance to execute based on the following evaluation:

- Function 6 is eliminated as a candidate because schema TODD is not in the SQL path.

- Function 7 in schema TAYLOR is eliminated as a candidate because it does not have the correct function name.
- Function 1 in schema SMITH is eliminated as a candidate because the INT data type is not promotable to the CHAR data type of the first parameter of Function 1.
- Function 3 in schema SMITH is eliminated as a candidate because it has the wrong number of parameters.
- Function 2 is a candidate because the data types of its parameters match or are promotable to the data types of the arguments.
- Both Function 4 and 5 in schema JOHNSON are candidates because the data types of their parameters match or are promotable to the data types of the arguments. However, Function 5 is chosen as the better candidate because although the data types of the first parameter of both functions (INT) match the first argument (INT), the data type of the second parameter of Function 5 (INT) is a better match of the second argument (INT) than the data type of Function 4 (DOUBLE).
- Of the remaining candidates, Function 2 and 5, the database manager selects Function 5 because schema JOHNSON comes before schema SMITH in the SQL path.

Example 3: Assume that functions were created with these partial CREATE FUNCTION statements:

1. **CREATE FUNCTION** BESTGEN.MYFUNC (**INT**, **DECIMAL**(9,0)) ...
2. **CREATE FUNCTION** KNAPP.MYFUNC (**INT**, **NUMERIC**(8,0))...
3. **CREATE FUNCTION** ROMANO.MYFUNC (**INT**, **FLOAT**) ...

Also assume that the SQL path at the time an application invokes a function is "ROMANO", "KNAPP", "BESTGEN". The function is invoked with two data types (SMALLINT, DECIMAL) as follows:

```
SELECT ... MYFUNC(SINTCOL1, DECIMALCOL) ...
```

Function 2 is chosen as the function instance to execute based on the following evaluation:

- All three functions are candidates for this function invocation because they meet the criteria specified in "Function resolution" on page 124.
- Function 3 in schema ROMANO is eliminated because the second parameter (FLOAT) is not as good a fit for the second argument (DECIMAL) as the second parameter of either Function 1 (DECIMAL) or Function 2 (NUMERIC).
- The second parameters of Function 1 (DECIMAL) and Function 2 (NUMERIC) are equally good fits for the second argument (DECIMAL).
- Function 2 is finally chosen because "KNAPP" precedes "BESTGEN" in the SQL path.

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 within the context in which the function is invoked, an error will occur. For example, given functions named STEP defined 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):

Functions

```
SELECT ... 3 +STEP(S)
```

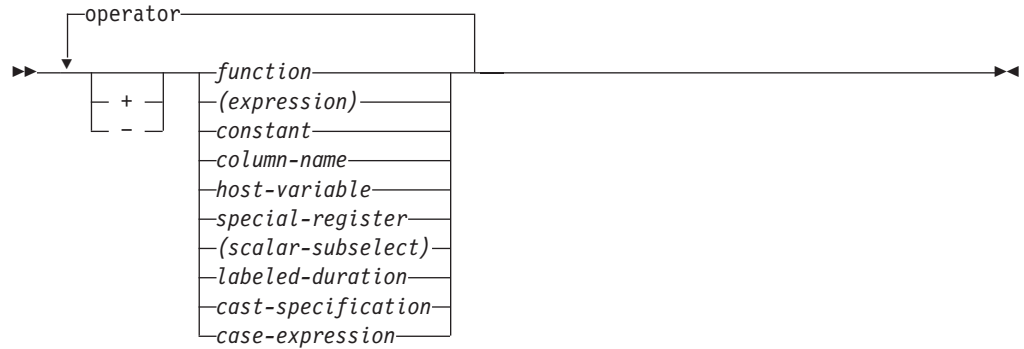
then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

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 80). This includes the case where precision, scale, length, or CCSID differs between the argument and the parameter.

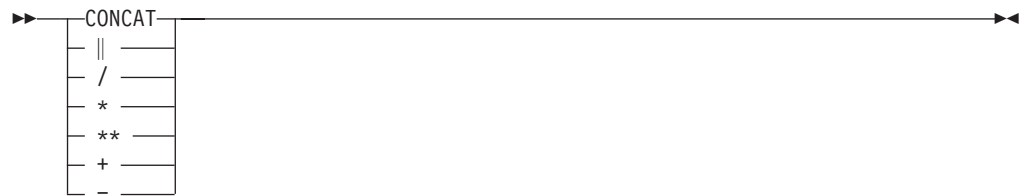
Expressions

An expression specifies a value.

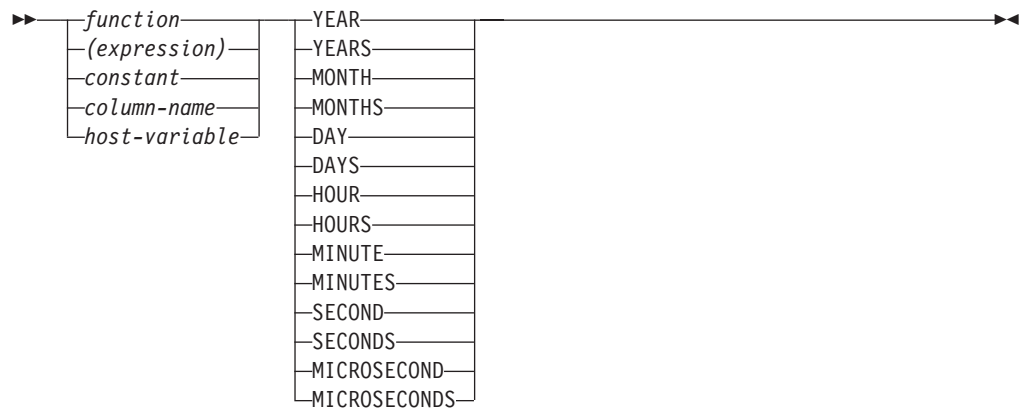
1



operator:



labeled-duration:



Without Operators

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

Example

SALARY :SALARY 'SALARY' MAX(SALARY)

With Arithmetic Operators

If arithmetic operators are used, the result of the expression is a number derived from the application of the operators to the values of the operands.

Expressions

If any operand can be null, 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. If the data type of `A` is small integer, the data type of `- A` is large integer. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators*, `+`, `-`, `*`, `/`, and `**`, specify addition, subtraction, multiplication, division, and exponentiation, respectively. The value of the second operand of division must not be zero.

In COBOL, blanks must precede and follow a minus sign to avoid any ambiguity with COBOL host variable names (which allow use of a dash).

The result of an exponentiation (`**`) operator is a double-precision floating-point number. The result of the other operators depends on the type of the operand.

Two Integer Operands

If both operands of an arithmetic operator are integers with zero scale, 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 large integers. If either integer operand has nonzero scale, it is converted to a decimal operand with the same precision and scale.

Integer and Decimal Operands

If one operand is an integer with zero scale and the other is decimal, the operation is performed in decimal using a temporary copy of the integer that has been converted to a decimal number with precision and scale 0 as defined in the following table:

Operand	Precision of Decimal Copy
Column or variable: big integer	19
Column or variable: large integer	11
Column or variable: small integer	5
Constant (including leading zeros)	Same as the number of digits in the constant

If one operand is an integer with nonzero scale, it is first converted to a decimal operand with the same precision and scale.

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.

Unless specified otherwise, all functions and operations that accept decimal numbers allow a precision of up to 31 digits. The result of a decimal operation must not have a precision greater than 31.

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 scale of the result of addition and subtraction is $\max(s, s')$. The precision is $\min(31, \max(p-s, p'-s') + \max(s, s') + 1)$.

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 $31-p+s'$. 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 are first 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 converted to double-precision floating point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number that has been converted to double-precision floating point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can slightly affect results because floating-point operands are approximate representations of real numbers. Since the order in which operands are processed may be implicitly modified by the optimizer (for example, the optimizer may decide what degree of parallelism to use and what access plan to use), an application should not depend on the results being precisely the same each time an SQL statement is executed that uses floating-point operands.

Distinct Types as Operands

A distinct type cannot be used with arithmetic operators even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, 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)
```

With the Concatenation Operator

The concatenation operator (CONCAT or ||) combines two strings. The result of the expression is a string.

Expressions

The operands of concatenation must be compatible strings that are not distinct types. Note that a binary string cannot be concatenated with a character string, including character strings defined as FOR BIT DATA.

The data type of the result is determined by the data types of the operands. The data type of the result is summarized in the following table:

Table 21. Result Data Types With Concatenation

If one operand column is ...	And the other operand is ...	The data type of the result column is ...
DBCLOB(x)	CHAR(y)* or VARCHAR(y)* or CLOB(y)* or GRAPHIC(y) or VARGRAPHIC(y) or DBCLOB(y)	DBCLOB(z) where $z = \text{MIN}(x + y, \text{maximum length of a DBCLOB})$
VARGRAPHIC(x)	CHAR(y)* or VARCHAR(y)* or GRAPHIC(y) or VARGRAPHIC(y)	VARGRAPHIC(z) where $z = \text{MIN}(x + y, \text{maximum length of a VARGRAPHIC})$
GRAPHIC(x)	CHAR(y)* mixed data	VARGRAPHIC(z) where $z = \text{MIN}(x + y, \text{maximum length of a VARGRAPHIC})$
GRAPHIC(x)	CHAR(y)* SBCS data or GRAPHIC(y)	GRAPHIC(z) where $z = \text{MIN}(x + y, \text{maximum length of a GRAPHIC})$
CLOB(x)*	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where $z = \text{MIN}(x + y, \text{maximum length of a DBCLOB})$
VARCHAR(x)*	GRAPHIC(y)	VARGRAPHIC(z) where $z = \text{MIN}(x + y, \text{maximum length of a VARGRAPHIC})$
CLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y)	CLOB(z) where $z = \text{MIN}(x + y, \text{maximum length of a CLOB})$
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where $z = \text{MIN}(x + y, \text{maximum length of a VARCHAR})$
CHAR(x) mixed data	CHAR(y)	VARCHAR(z) where $z = \text{MIN}(x + y, \text{maximum length of a VARCHAR})$
CHAR(x) SBCS data	CHAR(y)	CHAR(z) where $z = \text{MIN}(x + y, \text{maximum length of a CHAR})$
BLOB(x)	BLOB(y)	BLOB(z) where $z = \text{MIN}(x + y, \text{maximum length of a BLOB})$
Note:		
* Character strings are only allowed when the other operand is a graphic string if the graphic string is Unicode.		

Table 22. Result Encoding Schemes With Concatenation

If one operand column is ...	And the other operand is ...	The data type of the result column is ...
Unicode data	Unicode or DBCS or mixed or SBCS data	Unicode data
DBCS data	DBCS data	DBCS data
bit data	mixed or SBCS or bit data	bit data

Table 22. Result Encoding Schemes With Concatenation (continued)

If one operand column is ...	And the other operand is ...	The data type of the result column is ...
mixed data	mixed or SBCS data	mixed data
SBCS data	SBCS data	SBCS data

If the sum of the lengths of the operands exceeds the maximum length attribute of the resulting data type:

- The length attribute of the result is the maximum length of the resulting data type.²⁶
- If only blanks are truncated no warning or error occurs.
- If non-blanks are truncated, an error occurs.

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.

With mixed data this result will not have redundant shift codes “at the seam”. Thus, if the first operand is a string ending with a “shift-in” character (X'0F’), while the second operand is a character string beginning with a “shift-out” character (X'0E’), these two bytes are eliminated from the result.

The actual length of the result is the sum of the lengths of the operands unless redundant shifts are eliminated; in which case, the actual length is two less than the sum of the lengths of the operands.

The CONCAT operator should be used instead of the || operator. The code point for the | character varies, depending on the CCSID.

The CCSID of the result is determined by the CCSID of the operands as explained under “Conversion Rules for Operations That Combine Strings” on page 97. Note that as a result of these rules:

- If any operand is bit data, the result is bit data.
- If one operand is mixed data and the other is SBCS data, the result is mixed data. However, this does not necessarily mean that the result is well-formed mixed data.

Example

Concatenate the column FIRSTNME with a blank and the column LASTNAME.

```
FIRSTNME CONCAT ' ' CONCAT LASTNAME
```

Scalar Subselect

A scalar subselect as supported in an expression is a subselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the subselect 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, the result column name is based on the name of the column. See “subselect” on page 330 for more information.

26. If the expression is in the select-list, the length attribute may be further reduced in order to fit within the maximum record size. For more information, see “Maximum row sizes” on page 550.

Datetime Operands and Durations

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

Labeled Durations (see diagram on page 125)

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 MICROSECONDS²⁷. 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 *yyyymmdd*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. The result of subtracting one date value from another, as in the expression HIREDATE - 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. 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 *yyyymmddhhmmsszzzzzz*, 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.

27. Note that the singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

- 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 $RESULT = DATE1 - DATE2$.

If $DAY(DATE2) \leq DAY(DATE1)$
 then $DAY(RESULT) = DAY(DATE1) - DAY(DATE2)$.

If $DAY(DATE2) > DAY(DATE1)$
 then $DAY(RESULT) = N + DAY(DATE1) - DAY(DATE2)$
 where $N =$ the last day of $MONTH(DATE2)$.
 $MONTH(DATE2)$ is then incremented by 1.

If $MONTH(DATE2) \leq MONTH(DATE1)$
 then $MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2)$.

If $MONTH(DATE2) > MONTH(DATE1)$
 then $MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2)$.
 $YEAR(DATE2)$ is then incremented by 1.

$YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2)$.

Expressions

For example, the result of `DATE('3/15/2000') - '12/31/1999'` is 215 (or, a duration of 0 years, 2 months, and 15 days).

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 `SQLWARN6` in the `SQLCA` is set to 'W' to indicate the end-of-month adjustment.

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 `SQLWARN6` in the `SQLCA` is set to 'W' to indicate the end-of-month adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year. Adding a labeled duration of `DAYS` will not cause an end-of-month adjustment.

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.

Also note that logically equivalent expressions may not produce the same result. For example:

```
(DATE('2002-01-31') + 1 MONTH) + 1 MONTH will result in a date of  
2002-03-28.
```

does not produce the same result as

`DATE('2002-01-31') + 2 MONTHS` will result in a date of 2002-03-31.

The order in which labeled date durations are added to and subtracted from dates can affect the results. For compatibility with the results of adding or subtracting date durations, a specific order must be used. When labeled date durations are added to a date, specify them in the order of YEARS + MONTHS + DAYS. When labeled date durations are subtracted from a date, specify them in the order of DAYS - MONTHS - YEARS. For example, to add one year and one day to a date, specify:

`DATE1 + 1 YEAR + 1 DAY`

To subtract one year, one month, and one day from a date, specify:

`DATE1 - 1 DAY - 1 MONTH - 1 YEAR`

Time Arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting Times: The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0). If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1. If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation `RESULT = TIME1 - TIME2`.

If `SECOND(TIME2) <= SECOND(TIME1)`
 then `SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2)`.

If `SECOND(TIME2) > SECOND(TIME1)`
 then `SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2)`.
`MINUTE(TIME2)` is then incremented by 1.

If `MINUTE(TIME2) <= MINUTE(TIME1)`
 then `MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2)`.

If `MINUTE(TIME2) > MINUTE(TIME1)`
 then `MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2)`.
`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.

Expressions

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. $\text{TIME1} + X$, where "X" is a DECIMAL(6,0) number, is equivalent to the expression:

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

Timestamp Arithmetic

Timestamps can be subtracted, incremented, or decremented.

Subtracting Timestamps: The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps. The data type of the result is DECIMAL(20,6). If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation $\text{RESULT} = \text{TS1} - \text{TS2}$.

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. When the order of evaluation is not specified by parentheses, exponentiation is applied after prefix operators (such as -, unary minus) and before multiplication and division. Multiplication and division are applied before addition and subtraction. Operators at the same precedence level are applied from left to right. The following table shows the priority of all operators.

Priority	Operators
1	+, - (when used for signed numeric values)
2	**
3	*, /, CONCAT,
4	+, - (when used between two operands)

Example 1: In this example, the first operation is the addition in (SALARY + BONUS) because it is within parenthesis. The second operation is multiplication because it is at a higher precedence level than the second addition operator and it is to the left of the division operator. The third operation is division because it is at a higher precedence level than the second addition operator. Finally, the remaining addition is performed.

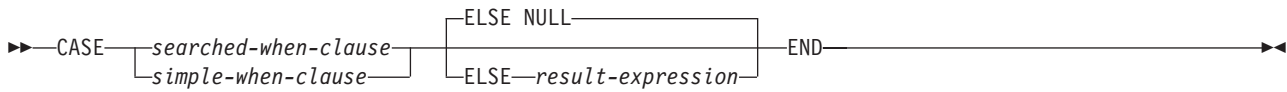
1.10 * (SALARY + BONUS) + SALARY / :VAR3

Example 2: In this example, the first operation (CONCAT) combines the character strings in the variables YYYYMM and DD into a string representing a date. The second operation (-) then subtracts that date from the date being processed in DATECOL. The result is a date duration that indicates the time elapsed between the two dates.

DATECOL - :YYYYMM CONCAT :DD

Expressions

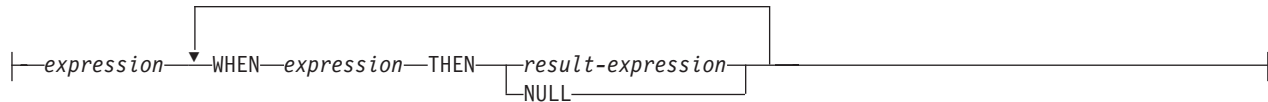
CASE Expressions



searched-when-clause:



simple-when-clause:



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) when-clause that evaluates to true. If no when-clause evaluates to true and the ELSE keyword is present then the result is the value of the ELSE *result-expression* or NULL. If no when-clause evaluates to true and the ELSE keyword is not present then the result is NULL. Note that when a when-clause evaluates to unknown (because of nulls), the when-clause is not true and hence is treated the same way as a when-clause that evaluates to false.

searched-when-clause

Specifies a *search-condition* that is applied to each row or group of table data presented for evaluation, and the result when that condition is true.

simple-when-clause

Specifies that the value of the *expression* prior to the first WHEN keyword is tested for equality with the value of the *expression* that follows each WHEN keyword. It also specifies the result when that condition is true.

The data type of the *expression* prior to the first WHEN keyword:

- must be compatible with the data types of the *expression* that follows each WHEN keyword.
- must not include a function that is non-deterministic or has an external action.

result-expression or NULL

Specifies the value that follows the THEN keyword and ELSE keywords. There must be at least one *result-expression* in the CASE expression with a defined data type. NULL cannot be specified for every case.

All *result-expressions* must have compatible data types, where the attributes of the result are determined based on the “Rules for Result Data Types” on page 93.

search-condition

Specifies a condition that is true, false, or unknown about a row or group of table data. The *search-condition* must not include a subquery in an EXISTS or IN predicate.

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

Table 23. Equivalent CASE Expressions

CASE Expression	Equivalent Expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)

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:

```
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
```

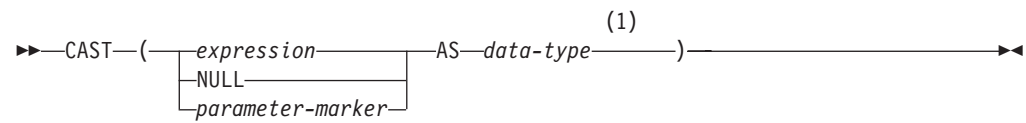
- The following CASE expressions are equivalent:

```
SELECT LASTNAME,
       CASE
       WHEN LASTNAME = 'Haas' THEN 'President'
       ...
       ELSE 'Unknown'
       END
FROM EMPLOYEE
```

Expressions

```
SELECT LASTNAME,  
CASE LASTNAME  
WHEN 'Haas' THEN 'President'  
...  
ELSE 'Unknown'  
END  
FROM EMPLOYEE
```

CAST Specification



Notes:

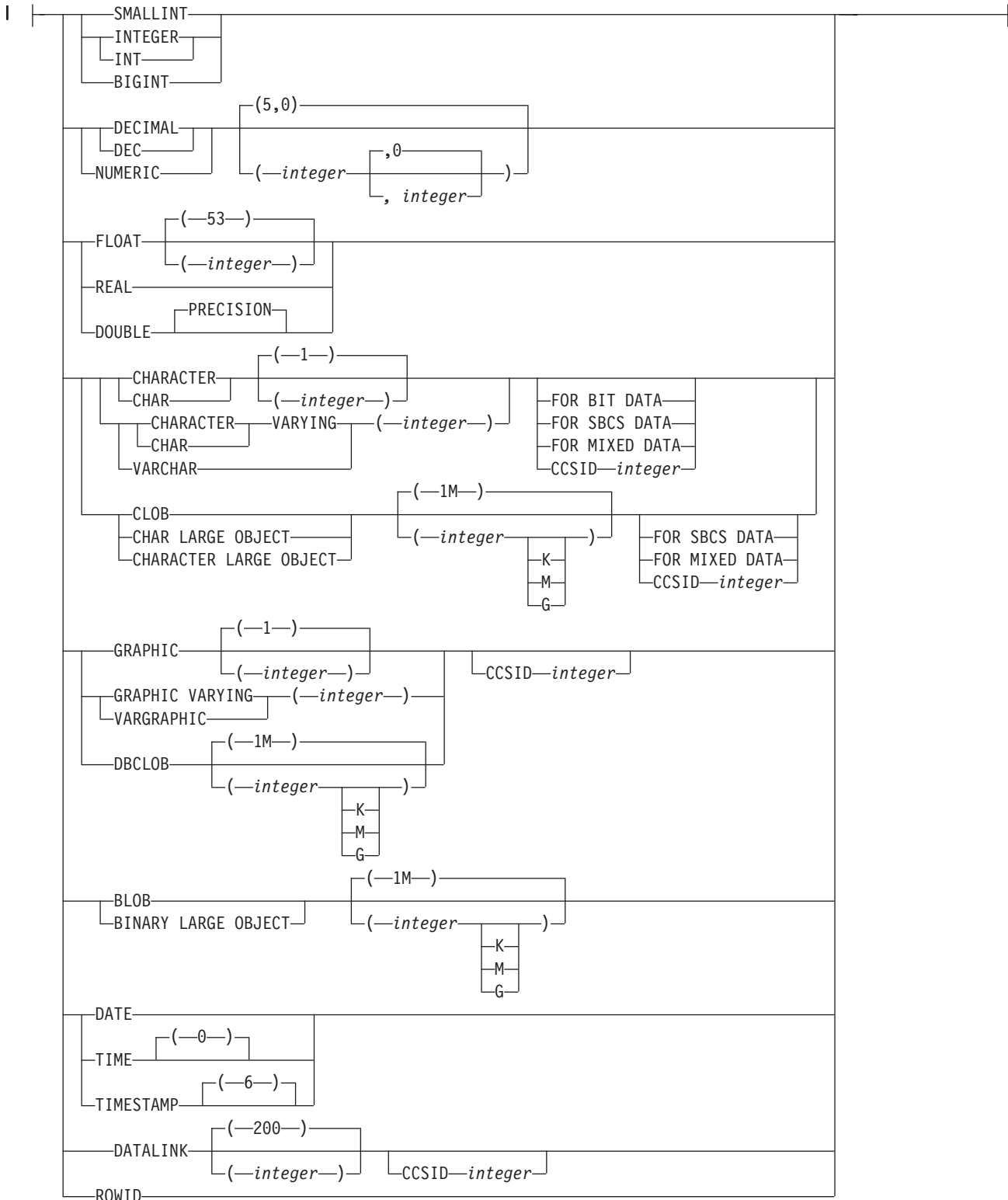
- 1 The data type names may be qualified. For more information see “Naming Conventions” on page 45.

Expressions

data-type:



built-in-type:



The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data type*. If the data type of either operand is a distinct type, the privileges held by the authorization ID of the statement must include USAGE authority on the distinct type.

expression

Specifies that the cast operand is an expression other than NULL or a parameter marker. The result is the argument value converted to the specified target data type.

The supported casts are shown in Table 12 on page 78, 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 is returned.

When casting character or graphic strings to a character or graphic string with a different length, a warning is returned if truncation of other than trailing blanks occurs.

NULL

Specifies that the cast operand is the null value. 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 the same rules as assignment to a column). Such a parameter marker is called a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of DESCRIBE of a select list or for column assignment.

data-type

Specifies the data type of the result. If the data type is not qualified, the SQL path is used to find the appropriate data type. For more information, see “Unqualified Function, Procedure, Specific, and Distinct Type Names” on page 54. For a description of *data-type*, see “CREATE TABLE” on page 525. (For portability across operating systems, when specifying a floating-point data type, use REAL or DOUBLE instead of FLOAT.)

Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an *expression*, see Table 12 on page 78 for the target data types that are supported based on the data type of the cast operand.
- For a cast operand that is the keyword NULL, the target data type can be any data type.
- For a cast operand that is a parameter marker, the target data type can be any data type. If the data type is a distinct type, the application that uses the parameter marker will use the source data type of the distinct type.

If the CCSID attribute is not specified, then:

- If the *data-type* is BLOB, a CCSID of 65535 is used.
- If the *expression* is a character string, and the *data-type* is CHAR, VARCHAR, or CLOB; the CCSID of the *expression* is used.
- If the *expression* is a graphic string, and the *data-type* is GRAPHIC, VARGRAPHIC, or DBCLOB; the CCSID of the *expression* is used.
- Otherwise, the default CCSID for the *data-type* is used.

Expressions

For information on which casts between data types are supported and the rules for casting to a data type see “Casting Between Data Types” on page 77.

Examples

- An application is only interested in the integer portion of the SALARY column (defined as DECIMAL(9,2)) from the EMPLOYEE table. The following CAST specification will convert the SALARY column to INTEGER.

```
SELECT EMPNO, CAST(SALARY AS INTEGER)
FROM EMPLOYEE
```

- Assume that two distinct types exist. T_AGE was sourced on SMALLINT and is the data type for the AGE column in the PERSONNEL table. R_YEAR was sourced on INTEGER and is the data type for the RETIRE_YEAR column in the same table. The following UPDATE statement could be prepared.

```
UPDATE PERSONNEL SET RETIRE_YEAR = ?
WHERE AGE = CAST( ? AS T_AGE )
```

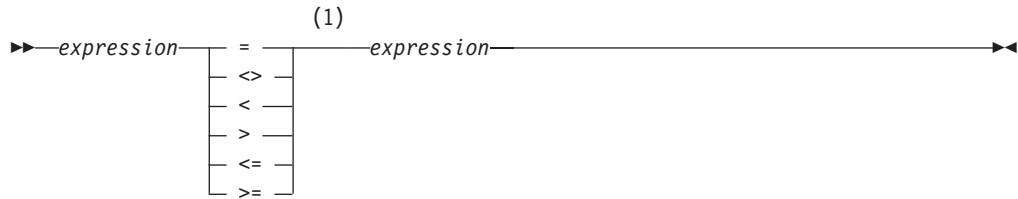
Predicates

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.
- The CCSID conversion of operands of predicates involving two or more operands are done according to “Conversion Rules for Comparison” on page 90.
- Use of a DataLink value is limited to the NULL predicate.

Basic Predicate



Notes:

1 Other comparison operators are also supported. ²⁸

A *basic predicate* compares two values.

If the operands of the predicate contain SBCS data or mixed data, and if the sort sequence in effect at the time the statement is executed is not *HEX, then the comparison of the operands is performed using weighted values for the operands. The weighted values are based on the sort sequence.

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*:

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

Examples

```

EMPNO = '528671'

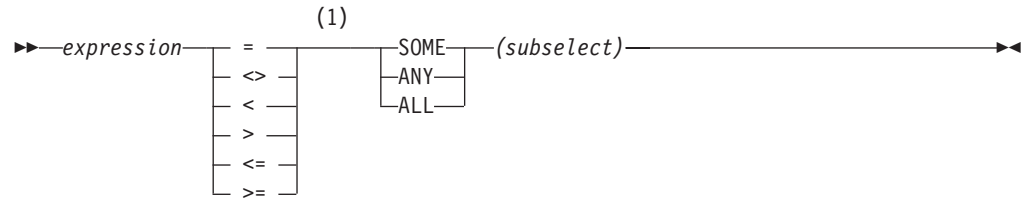
PRTSTAFF <> :VAR1

SALARY + BONUS + COMM < 20000

SALARY > (SELECT AVG(SALARY)
          FROM EMPLOYEE)
    
```

28. The following forms of the comparison operators are also supported in basic and quantified predicates: !=, !<, !>, !=, !=, and != are supported. All these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators and are not recommended for use when writing new SQL statements. Some keyboards must use the hex values for the not (!) symbol. The hex value varies and is dependent on the keyboard that is used. A not sign (!) or the character that must be used in its place in certain countries, can cause parsing errors in statements passed from one database server to another. The problem occurs if the statement undergoes character conversion with certain combinations of source and target CCSIDs. To avoid this problem, substitute an equivalent operator for any operator that includes a not sign. For example, substitute '<>' for '!<=', '<=' for '!>', and '>=' for '!<'.

Quantified Predicate



Notes:

1 Other comparison operators are also supported. ²⁸

A *quantified predicate* compares a value with a set of values.

The subselect must specify a single result column and can return any number of values, whether null or not null. If the operands of the predicate contain SBCS data or mixed data, and if the sort sequence in effect at the time the statement is executed is not *HEX, then the comparison is performed using weighted values for the operands. The weighted values are based on the sort sequence.

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

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

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

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

Examples

Table TBLA

```
COLA
-----
  1
  2
  3
  4
null
```

Table TBLB

Quantified Predicate

```
COLA  
-----  
  2  
  3
```

Example 1

```
SELECT * FROM TBLA WHERE COLA = ANY(SELECT COLB FROM TBLB)
```

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 * FROM TBLA WHERE COLA > ANY(SELECT COLB FROM TBLB)
```

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 * FROM TBLA WHERE COLA > ALL(SELECT COLB FROM TBLB)
```

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 * FROM TBLA WHERE COLA > ALL(SELECT COLB FROM TBLB WHERE COLB<0)
```

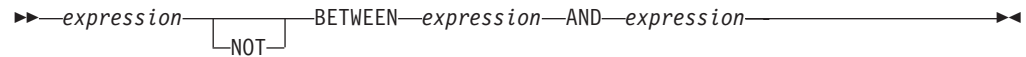
Results in 1,2,3,4, and null. The subselect returns no values. Thus, the predicate is true for all rows in TBLA.

Example 5

```
SELECT * FROM TBLA WHERE COLA > ANY(SELECT COLB FROM TBLB WHERE COLB<0)
```

Results in the empty set. The subselect returns no values. Thus, the predicate is false for all rows in TBLA.

BETWEEN Predicate



The BETWEEN predicate compares a value with a range of values.

If a sort sequence other than *HEX is in effect when the statement is executed and the BETWEEN predicate involves SBCS data or mixed data, the weighted values of the strings are compared instead of the values. The weighted value is based on the sort sequence.

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is logically 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.
```

If the operands of the BETWEEN predicate are strings with different CCSIDs, operands are converted as if the above logically-equivalent search conditions were specified.

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

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000
```

```
SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

EXISTS Predicate

EXISTS Predicate

►►—EXISTS—(*subselect*)—◄◄

The EXISTS predicate tests for the existence of certain rows. The subselect may specify any number of columns, and

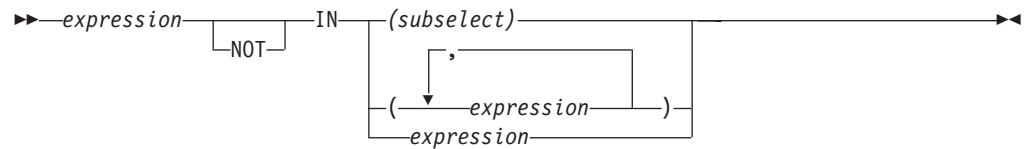
- The result is true only if the number of rows specified by the subselect is not zero.
- The result is false only if the number of rows specified by the subselect is zero.
- The result cannot be unknown.

The values returned by the subselect are ignored.

Example

```
EXISTS (SELECT * FROM EMPLOYEE WHERE SALARY > 60000)
```

IN Predicate



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

If a sort sequence other than *HEX is in effect when the statement is executed and the IN predicate involves SBCS data or mixed data, the weighted values of the strings are compared instead of the actual values. The weighted values are based on the sort sequence.

In the subselect form, the subselect must identify a single result column and may return any number of values, whether null or not null.

An IN predicate of the form:

```
expression IN (subselect)
```

is equivalent to a quantified predicate of the form:

```
expression = ANY (subselect)
```

An IN predicate of the form:

```
expression NOT IN (subselect)
```

is equivalent to a quantified predicate of the form:

```
expression <> ALL (subselect)
```

An IN predicate of the form:

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

is logically equivalent to:

```
expression IN (SELECT * FROM R)
```

where T is a table with a single row and R is a temporary table formed by the following fullselect:

```
SELECT value1 FROM T
UNION
SELECT value2 FROM T
UNION
.
.
.
UNION
SELECT valuen FROM T
```

Each host variable must identify a structure or variable that is described in accordance with the rule for declaring host structures or variables.

IN Predicate

If the operands of the IN predicate have different data types or attributes, the rules used to determine the data type for evaluation of the IN predicate are those for UNION and UNION ALL. For a description, see “Rules for Result Data Types” on page 93.

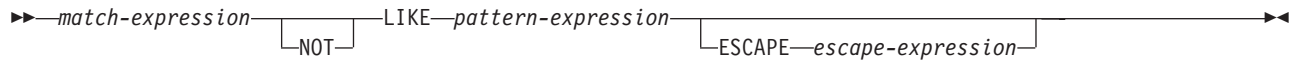
If the operands of the IN predicate are strings with different CCSIDs, the rules used to determine which operands are converted are those for operations that combine strings. For a description, see “Conversion Rules for Operations That Combine Strings” on page 97.

Examples

```
DEPTNO IN ('D01', 'B01', 'C01')
```

```
EMPNO IN(SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

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 have special meanings. Trailing blanks in a pattern are a part of the pattern.

If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The *match-expression*, *pattern-expression*, and *escape-expression* must identify strings. The values for *match-expression*, *pattern-expression*, and *escape-expression* must either all be binary strings or none can be binary strings. The three arguments can include a mixture of character strings and graphic strings.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

If a sort sequence other than *HEX is in effect when the statement is executed and the LIKE predicate involves SBCS data or mixed data, the weighted values of the strings are compared instead of the actual values. The weighted values are based on the sort sequence.

With character strings, the terms **character**, **percent sign**, and **underscore** in the following discussion refer to single-byte characters. With graphic strings, the terms refer to double-byte or UCS-2 characters. With binary strings, the terms refer to the code points of those single-byte characters.

match-expression

An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

LIKE *pattern-expression*

An expression that specifies the string that is to be matched.

A simple description of the pattern

The pattern is used to specify the conformance criteria for values in the *match-expression* where:

- The underscore sign (`_`) 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 pattern

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

LIKE Predicate

Let m denote a value of *match-expression* and 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 nonempty 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 of the predicate is either true or false. The result is true either if both m and p are 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 consisting of other than percent signs, the result is false.

The predicate m NOT LIKE p is equivalent to the search condition NOT(m LIKE p).

If necessary, the CCSID of the *match-expression*, *pattern-expression*, and *escape-expression* are converted to the compatible CCSID between the *match-expression* and *pattern-expression*.

Mixed data

If the expression is mixed data, the expression might contain double-byte characters, and the pattern can include both SBCS and DBCS characters. In that case the special characters in p are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one DBCS character.
- A percent sign (either SBCS or DBCS) refers to any number of characters of any type, either SBCS or DBCS.
- Redundant shifts in *match-expression* and *pattern-expression* are ignored.²⁹

UCS-2 data

If the expression is UCS-2 graphic data, the pattern can include either or both of the supported code points for the UCS-2 underscore and percent sign. The

29. Redundant shifts are normally ignored. To guarantee that they are ignored, however, specify the IGNORE_LIKE_REDUNDANT_SHIFTS query attribute.

supported code points for the UCS-2 underscore are X'005F' and X'FF3F'. The supported code points for the UCS-2 percent sign are X'0025' and X'FF05'.

Parameter Marker

When the pattern specified in a LIKE predicate is a parameter marker, and a fixed-length character host variable is used to replace the parameter marker; specify a value for the host variable that is the correct length. If a correct length is not specified, the select will not return the intended results.

For example, if the host variable is defined as CHAR(10), and the value WYSE% is assigned to that host variable, the host variable is padded with blanks on assignment. The pattern used is

```
'WYSE%      '
```

This pattern requests the database manager to search for all values that start with WYSE and end with five blank spaces. If you intended to search for only the values that start with 'WYSE' you should assign the value 'WYSE%%%%%%%%%' to the host variable.

ESCAPE *escape-expression*

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 following rules apply the use of the ESCAPE clause and the *escape-expression*:

- The *escape-expression* must be a string of length 1.³⁰
- The *pattern-expression* must not contain the escape character except when followed by the escape character, percent, or underscore.
For example, if '+' is the escape character, any occurrences of '+' other than '++', '+_', or '+%' in the *pattern-expression* is an error.
- The *escape-expression* can be a parameter marker.

The following example shows the effect of successive occurrences of the escape character, which in this case is the plus sign (+).

When the pattern string is...	The actual pattern is...
+%	A percent sign
++%	A plus sign followed by zero or more arbitrary characters
+++%	A plus sign followed by a percent sign

Examples

Example 1: Search for the string 'SYSTEMS' appearing anywhere within the PROJNAME column in the PROJECT table.

```
SELECT PROJNAME
FROM PROJECT
WHERE PROJECT.PROJNAME LIKE '%SYSTEMS%'
```

Example 2: Search for a string with a first character of 'J' that is exactly two characters long in the FIRSTNME column of the EMPLOYEE table.

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

LIKE Predicate

```
SELECT FIRSTNME
   FROM EMPLOYEE
  WHERE EMPLOYEE.FIRSTNME LIKE 'J_'
```

Example 3: In this example:

```
SELECT *
   FROM TABLEY
  WHERE C1 LIKE 'AAAA+%BBB%' ESCAPE '+'
```

'+' is the escape character and indicates that the search is for a string that starts with 'AAAA%BBB'. The '+%' is interpreted as a single occurrence of '%' in the pattern.

Example 4: Assume that a distinct type named ZIP_TYPE with a source data type of CHAR(5) exists and an ADDRZIP column with data type ZIP_TYPE exists in some table TABLEY. The following statement selects the row if the zip code (ADDRZIP) begins with '9555'.

```
SELECT *
   FROM TABLEY
  WHERE CHAR(ADDRZIP) LIKE '9555%'
```

Example 5: The RESUME column in sample table EMP_RESUME is defined as a CLOB. If the host variable LASTNAME has a value of 'JONES', the following statement selects the RESUME column when the string JONES appears anywhere in the column.

```
SELECT RESUME
   FROM EMP_RESUME
  WHERE RESUME LIKE '%' || LASTNAME || '%'
```

Example 6: In the following table of EBCDIC examples, assume COL1 is mixed data. The table shows the results when the predicates in the first column are evaluated using the COL1 values from the second column:

LIKE Predicate

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa %AB%CS _I '	'aaa %ABDZCS _I '	True
WHERE COL1 LIKE 'aaa %AB % _I %CS _I '	'aaa %AB % _I dzx %CS _I '	True
WHERE COL1 LIKE 'a %CS _I '	'a %CS _I '	True
	'ax %CS _I '	True
	'ab %DE %fg %CS _I '	True
WHERE COL1 LIKE 'a %CS _I '	'a %CS _I '	True
	'a %XCS _I '	False
WHERE COL1 LIKE 'a %__CS _I '	'a %XCS _I '	True
	'ax %CS _I '	False
WHERE COL1 LIKE '%S _I '	Empty string	True
WHERE COL1 LIKE 'ab %CS _I _'	'ab %CS _I d'	True
	'ab % _I %CS _I d'	True

RV3F001-0

NULL Predicate

NULL Predicate

►► *expression* IS NOT NULL ◀◀

The NULL predicate tests for null values.

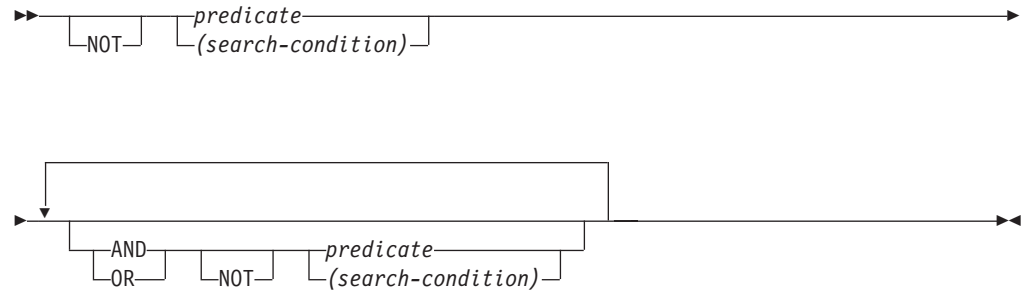
The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

Examples

EMPLOYEE.PHONE IS NULL

SALARY IS NOT NULL

Search Conditions



A *search condition* specifies a condition that is true, false, or unknown about a given row or group.

The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in the following table in which P and Q are any predicates:

Table 24. 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

Examples

In the examples, the numbers on the second line indicate the order in which the operators are evaluated.

Example 1

```
MAJPROJ = 'MA2100' AND DEPTNO = 'D11' OR DEPTNO = 'B03' OR DEPTNO = 'E11'  
1 2 or 3 2 or 3
```

Example 2

```
MAJPROJ = 'MA2100' AND (DEPTNO = 'D11' OR DEPTNO = 'B03') OR DEPTNO = 'E11'  
2 1 3
```

Chapter 3. Built-In Functions

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

The *built-in functions* are part of schema QSYS2. A *built-in function* can be invoked with or without its schema name. Regardless of whether a schema name qualifies the function name, the database manager uses function resolution to determine which function to use. For more information on functions and the process of function resolution, see “Function resolution” on page 124.

Functions are classified as *column functions* or *scalar functions*. The argument of a column function is a set of values. An argument of a scalar function is a single value.

In the syntax of SQL, the term *function* is used only in the definition of an expression. Thus, a function can be used only where an expression can be used. Additional restrictions apply to the use of column functions as specified in the following section and in Chapter 4, “Queries” on page 329.

The following tables list the different types of *built-in functions*:

Table 25. Column Functions

“AVG” on page 169	Returns the average of a set of numbers
“COUNT” on page 170	Returns the number of rows or values in a set of rows or values
“COUNT_BIG” on page 172	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.
“MAX” on page 173	Returns the maximum value in a set of values in a group
“MIN” on page 174	Returns the minimum value in a set of values in a group
“STDDEV_POP or STDDEV” on page 175	Returns the biased standard deviation (/n) of a set of numbers.
“SUM” on page 176	Returns the sum of a set of numbers
“VAR_POP or VARIANCE or VAR” on page 177	Returns the biased variance (/n) of a set of numbers.

Table 26. Cast Scalar Functions

“BIGINT” on page 186	Returns a big integer representation of a number
“BLOB” on page 187	Returns a BLOB representation of a string of any type
“CHAR” on page 190	Returns a CHARACTER representation of a value
“CLOB” on page 196	Returns a CLOB representation of a value
“DATE” on page 207	Returns a DATE from a value
“DBCLOB” on page 215	Returns a DBCLOB representation of a string expression
“DECIMAL or DEC” on page 217	Returns a DECIMAL representation of a number

Built-In Functions

Table 26. Cast Scalar Functions (continued)

"DOUBLE_PRECISION or DOUBLE" on page 232	Return a DOUBLE_PRECISION representation of a number
"FLOAT" on page 235	Return a FLOAT representation of a number
"GRAPHIC" on page 237	Returns a GRAPHIC representation of a string expression
"INTEGER or INT" on page 248	Returns an INTEGER representation of a number
"REAL" on page 283	Returns a REAL representation of a number
"ROWID" on page 286	Returns a Row ID from a value.
"SMALLINT" on page 293	Returns a SMALLINT representation of a number
"TIME" on page 303	Returns a TIME from a value
"TIMESTAMP" on page 304	Returns a TIMESTAMP from a value or a pair of values
"VARCHAR" on page 316	Returns a VARCHAR representative of a value
"VARGRAPHIC" on page 320	Returns a VARGRAPHIC representation of a string expression
"ZONED" on page 327	Returns a zoned decimal representation of a number

Table 27. Datalink Scalar Functions

"DLCOMMENT" on page 223	Returns the comment value from a DataLink value
"DLLINKTYPE" on page 224	Returns the link type value from a DataLink value
"DLURLCOMPLETE" on page 225	Returns the complete URL value from a DataLink value with a link type of URL
"DLURLPATH" on page 226	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 returned includes a file access token.
"DLURLPATHONLY" on page 227	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.
"DLURLSCHEME" on page 228	Returns the scheme from a DataLink value with a linktype of URL
"DLURLSERVER" on page 229	Returns the file server from a DataLink value with a linktype of URL
"DLVALUE" on page 230	Returns a DataLink value

Table 28. Datetime Scalar Functions

"CURDATE" on page 205	Returns a date based on a reading of the time-of-day clock
"CURTIME" on page 206	Returns a time based on a reading of the time-of-day clock
"DAY" on page 209	Returns the day part of a value
"DAYOFMONTH" on page 210	Returns the day part of a value
"DAYOFWEEK" on page 211	Returns an integer that represents the day of the week, where 1 is Sunday and 7 is Saturday
"DAYOFWEEK_ISO" on page 212	Returns an integer that represents the day of the week, where 1 is Monday and 7 is Sunday
"DAYOFYEAR" on page 213	Returns an integer that represents the day of the year

Table 28. Datetime Scalar Functions (continued)

"DAYS" on page 214	Returns an integer representation of a date
"HOUR" on page 242	Returns the hour part of a value
"JULIAN_DAY" on page 250	Returns an integer value representing a number of days from January 1, 4712 B.C. to the date specified in the argument
"MICROSECOND" on page 264	Returns the microsecond part of a value
"MIDNIGHT_SECONDS" on page 265	Returns an integer value representing the number of seconds between midnight and a specified time value
"MINUTE" on page 267	Returns the minute part of a value
"MONTH" on page 270	Returns the month part of a value
"NOW" on page 273	Returns a timestamp based on a reading of the time-of-day clock
"QUARTER" on page 280	Returns an integer that represents the quarter of the year in which the date resides
"SECOND" on page 289	Returns the seconds part of a value
"TIMESTAMPDIFF" on page 306	Returns an estimated number of intervals based on the difference between two timestamps
"WEEK" on page 323	Returns an integer that represents the week of the year. The week starts with Sunday.
"WEEK_ISO" on page 324	Returns an integer that represents the week of the year. The week starts with Monday.
"YEAR" on page 326	Returns the year part of a value

Table 29. Partitioning Scalar Functions

"HASH" on page 240	Returns the partition number of a set of values
"NODENAME" on page 271	Returns the relational database name of where a row is located
"NODENUMBER" on page 272	Returns the node number of a row
"PARTITION" on page 275	Returns the partition number of a row

Table 30. Miscellaneous Scalar Functions

"COALESCE" on page 200	Returns the first argument that is not null
"HEX" on page 241	Returns a hexadecimal representation of a value
"IDENTITY_VAL_LOCAL" on page 243	Returns the most recently assigned value for an identity column
"IFNULL" on page 247	Returns the first argument that is not null
"LENGTH" on page 255	Returns the length of a value
"MAX" on page 263	Returns the maximum value in a set of values
"MIN" on page 266	Returns the minimum value in a set of values
"NULLIF" on page 274	Returns a null value if the arguments are equal, otherwise it returns the value of the first argument
"RRN" on page 287	Returns the relative record number of a row
"VALUE" on page 315	Returns the first argument that is not null

Built-In Functions

Table 31. Numeric Scalar Functions

"ABS" on page 179	Return the absolute value of a number
"ACOS" on page 180	Returns the arc cosine of a number, in radians
"ANTILOG" on page 181	Returns the anti-logarithm (base 10) of a number
"ASIN" on page 182	Returns the arc sine of a number, in radians
"ATAN" on page 183	Returns the arc tangent of a number, in radians
"ATANH" on page 184	Returns the hyperbolic arc tangent of a number, in radians
"ATAN2" on page 185	Returns the arc tangent of x and y coordinates as an angle expressed in radians
"CEILING" on page 189	Returns the smallest integer value that is greater than or equal to a numeric-expression
"COS" on page 202	Returns the cosine of a number
"COSH" on page 203	Returns the hyperbolic cosine of a number
"COT" on page 204	Returns the cotangent of a number
"DEGREES" on page 220	Returns the number of degrees of an angle
"DIGITS" on page 222	Returns a character-string representation of the absolute value of a number
"EXP" on page 234	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument
"FLOOR" on page 236	Returns the largest integer value less than or equal to a numeric-expression
"LN" on page 256	Returns the natural logarithm of a number
"LOG10" on page 259	Return the common logarithm (base 10) of a number
"MOD" on page 268	Divides the first argument by the second argument and returns the remainder
"PI" on page 276	Returns the value of PI
"POWER" on page 279	Returns the result of raising the first argument to the power of the second argument
"RADIANS" on page 281	Returns the number of radians for an argument that is expressed in degrees
"RAND" on page 282	Returns a random number
"ROUND" on page 284	Returns a numeric value that has been rounded to the specified number of decimal places
"SIGN" on page 290	Returns an indicator of the sign of an expression
"SIN" on page 291	Returns the sine of a number
"SINH" on page 292	Returns the hyperbolic sine of a number
"SQRT" on page 296	Returns the square root of a number
"TAN" on page 301	Returns the tangent of a number
"TANH" on page 302	Returns the hyperbolic tangent of a number
"TRUNCATE or TRUNC" on page 311	Returns a number value that as been truncated at a specified number of decimal places

Table 32. String Scalar Functions

"CHARACTER_LENGTH" on page 195	Returns the length of a string expression.
--------------------------------	--------------------------------------------

Table 32. String Scalar Functions (continued)

"CONCAT" on page 201	Concatenates two strings.
"DIFFERENCE" on page 221	Returns a value representing the difference between the sounds of two strings
"LAND" on page 251	Returns a string that is the logical 'AND' of the argument strings
"LCASE" on page 252	Returns a string in which all the characters have been converted to lowercase characters
"LEFT" on page 253	Returns the leftmost characters from the string
"LNOT" on page 257	Returns a string that is the logical NOT of the argument string
"LOCATE" on page 258	Returns the starting position of one string within another string
"LOR" on page 260	Returns a string that is the logical OR of the argument strings
"LOWER" on page 261	Returns a string in which all the characters have been converted to lowercase characters
"LTRIM" on page 262	Removes blanks or hexadecimal zeros from the beginning of a string expression
"POSITION or POSSTR" on page 277	Return the starting position of one string within another string
"RTRIM" on page 288	Removes blanks or hexadecimal zeroes from the end of a string expression
"SOUNDEX" on page 294	Returns a character code representing the sound of the words in the argument
"SPACE" on page 295	Returns a character string that consists of the number of blanks that the argument specifies
"STRIP" on page 297	Removes blanks or another specified character from the end or beginning of a string expression
"SUBSTRING or SUBSTR" on page 298	Returns a substring of a string
"TRANSLATE" on page 307	Translates one or more characters in a string
"TRIM" on page 309	Removes blanks or another specified character from the end or beginning of a string expression
"UCASE" on page 313	Returns a string in which all the characters have been converted to uppercase characters
"UPPER" on page 314	Returns a string in which all the characters have been converted to uppercase characters
"XOR" on page 325	Returns a string that is the logical XOR of the argument strings

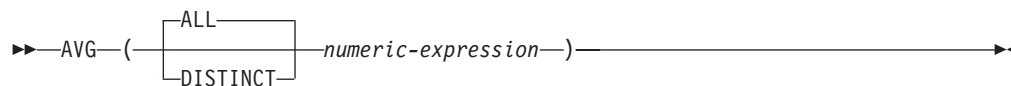
Column Functions

The following information applies to all column functions other than COUNT(*) and COUNT_BIG(*).

- The argument of a column function is a set of values derived from an *expression*. The *expression* may include columns but cannot include another column function. The scope of the set is a group or an intermediate result table as explained in Chapter 4, "Queries".
- If a GROUP BY clause is specified in a query and the intermediate result of 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 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 JOB for employees in department D01:

```
SELECT COUNT(DISTINCT JOB)
FROM 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.
- A column function can be used in a WHERE clause only if that clause is part of a subquery of a HAVING clause and the column name specified in the expression is a correlated reference to a group. If the expression includes more than one column name, each column name must be a correlated reference to the same group.

AVG



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

The argument values must be any built-in numeric data type and their sum must be within the range of the data type of the result.

The data type of the result is the same as the data type of the argument values, except that:

- The result is double-precision floating point if the argument values are single-precision floating point.
- The result is large integer if the argument values are small integers.
- The result is decimal with precision 31 and scale $31-p+s$ if the argument values are decimal or nonzero scale binary with precision p and scale s .

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is used, duplicate values are eliminated.

The result can be null. If set of values is empty, the result is the null value. Otherwise, the result is the average value of 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.

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).

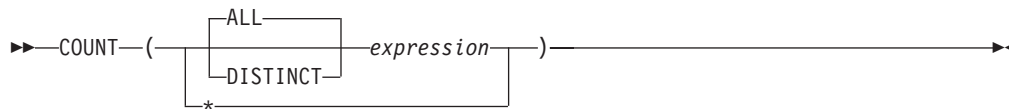
- Using the PROJECT table, set the host variable ANY_CALC to the average of each unique staffing value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
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).

COUNT

COUNT



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

The result of the function is a large integer and it must be within the range of large integers. The result cannot be null. If the table is a distributed table, then the result is DECIMAL(15,0). For more information about distributed tables, see the DB2 Multisystem book.

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 values can be of any built-in data type other than a BLOB, CLOB, DBCLOB, or DataLink. If DISTINCT is used, the resulting *expression* must not have a length attribute greater than 2000 for a character column or 1000 for a graphic column.

The argument of COUNT(*expression*) or COUNT(ALL *expression*) is a set of values. The function is applied to the set 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.

The argument of COUNT(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values and duplicate values. The result is the number of values in the set.

If a sort sequence other than *HEX is in effect when the statement that contains the COUNT(DISTINCT *expression*) is executed and the arguments contain SBCS, UCS-2, or mixed data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the sort sequence.

Examples

- Using the EMPLOYEE table, set the host variable FEMALE (INTEGER) 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.

- Using the EMPLOYEE table, set the host variable FEMALE_IN_DEPT (INTEGER) to the number of departments (WORKDEPT) that have at least one female as a member.

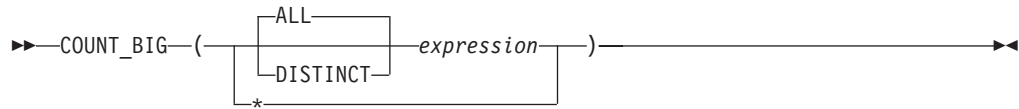
```
SELECT COUNT(DISTINCT WORKDEPT)
  INTO :FEMALE_IN_DEPT
  FROM EMPLOYEE
  WHERE SEX='F'
```

COUNT

Results in FEMALE_IN_DEPT being set to 5. (There is at least one female in departments A00, C01, D11, D21, and E11.)

COUNT_BIG

COUNT_BIG



The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of integer.

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(*expression*) is a set of values. The function is applied to the set derived from the argument values by the elimination of null values. The result is the number of values in the set.

The argument values can be of any built-in data type other than a BLOB, CLOB, DBCLOB, or DataLink. If DISTINCT is used, the resulting *expression* must not have a length attribute greater than 2000 for a character column or 1000 for a graphic column.

If a sort sequence other than *HEX is in effect when the statement that contains the COUNT_BIG(DISTINCT *expression*) is executed and the arguments contain SBCS, UCS-2, or mixed data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the sort sequence.

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.
- To count on a specific column, a sourced function must specify the type of the column. In this example, the CREATE FUNCTION statement creates a sourced function that takes any column defined as CHAR, uses COUNT_BIG to perform the counting, and returns the result as a double precision floating-point number. The query shown counts the number of unique departments in the sample employee table.

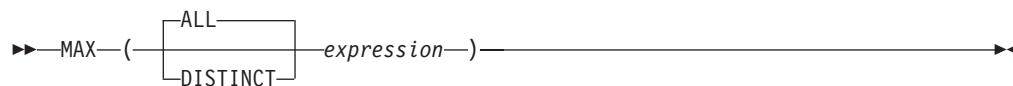
```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE  
SOURCE QSYS2.COUNT_BIG(CHAR());
```

```
SET CURRENT PATH RICK, SYSTEM PATH
```

```
SELECT COUNT(DISTINCT WORKDEPT FROM EMPLOYEE;
```

The empty parenthesis in the parameter list for the new function (RICK.COUNT) means that the input parameter for the new function is the same type as the input parameter for the function named in the SOURCE clause. The empty parenthesis in the parameter list in the SOURCE clause (COUNT_BIG) means that the length attribute of the CHAR parameter of the COUNT_BIG function is ignored when DB2 locates the COUNT_BIG function.

MAX



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

The argument values can be any built-in data types except LOB and DataLink values.

The data type and length attribute of the result are the same as the data type and length attribute of the argument values. When the argument is a string, the result has the same CCSID as the argument.

If a sort sequence other than *HEX is in effect when the statement that contains the MAX function is executed and the arguments contain SBCS, UCS-2, or mixed data, then the result is obtained by comparing weighted values for each value in the set. The weighted values are based on the sort sequence.

The function is applied to the set of values derived from the argument values by the elimination of null values.

The result can be null. If the function is applied to the 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 is not advised.

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.

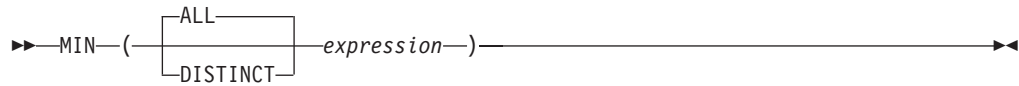
- Using the PROJECT table, set the host variable LAST_PROJ (CHAR(24)) to the project name (PROJNAME) that comes last in the sort sequence.

```
SELECT MAX(PROJNAME)
INTO :LAST_PROJ
FROM PROJECT
```

Results in LAST_PROJ being set to 'WELD LINE PLANNING'.

MIN

MIN



The MIN column function returns the minimum value in a set of values in a group.

The argument values can be any built-in data types except LOB and DataLink values.

The data type and length attribute of the result are the same as the data type and length attribute of the argument values. When the argument is a string, the result has the same CCSID as the argument. The result can be null.

If a sort sequence other than *HEX is in effect when the statement that contains the MIN function is executed and the arguments contain SBCS, UCS-2, or mixed data, then the result is obtained by comparing weighted values for each value in the set.

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 the empty set, the result 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 is not advised.

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'
```

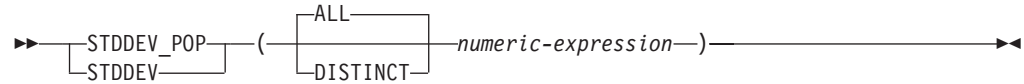
Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462).

- 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'.

STDDEV_POP or STDDEV



The STDDEV_POP function returns the biased standard deviation ($/n$) of a set of numbers. The formula used to calculate the biased standard deviation is:

$$\text{STDDEV_POP} = \text{SQRT}(\text{VAR_POP})$$

where $\text{SQRT}(\text{VAR_POP})$ is the square root of the variance.

The argument values must be any built-in numeric data type.

The data type of the result is double-precision floating point.

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.

The result can be null. If the function is applied to the 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 added is undefined, but every intermediate result must be within the range of the result data type.

Note

Syntax alternatives: STDDEV is a synonym for STDDEV_POP.

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 A00.

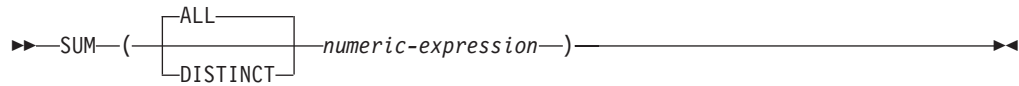
```

SELECT STDDEV_POP(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';

```

Results in DEV being set to approximately 9938.00.

SUM



The SUM function returns the sum of a set of numbers.

The argument values must be any built-in numeric data type.

The data type of the result is the same as the data type of the argument values except that the result is:

- A double-precision floating point if the argument values are single-precision floating point
- A large integer if the argument values are small integers
- A decimal with precision 31 and scale s if the argument values are decimal or nonzero scale binary numbers with precision p and scale s .

If the data type of the argument values is decimal or nonzero scale binary, the precision of the result is 31 and the scale is the same as the scale of the argument values.

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.

The result can be null. If the function is applied to the empty set, the result is a null value. Otherwise, the result is the sum 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.

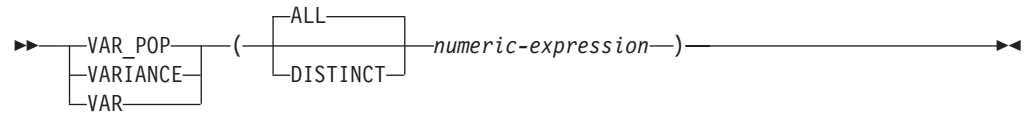
Example

- Using the EMPLOYEE table, set the host variable 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.

VAR_POP or VARIANCE or VAR



The VAR_POP function returns the biased variance ($/n$) of a set of numbers. The formula used to calculate the biased variance is:

$$\text{VAR_POP} = \text{SUM}(X**2) / \text{COUNT}(X) - (\text{SUM}(X) / \text{COUNT}(X))**2$$

The argument values must be any built-in numeric data type.

The data type of the result is double-precision floating point.

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.

The result can be null. If the function is applied to the 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.

Note

Syntax alternatives: VARIANCE and VAR are synonyms for VAR_POP.

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 A00.

```

SELECT VAR_POP(SALARY)
  INTO :VARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00';
  
```

Results in VARNCE being set to approximately 98763888.88.

Scalar Functions

A *scalar function* can be used wherever an expression can be used. The restrictions on the use of column functions do not apply to scalar functions, because a scalar function is applied to single parameter values rather than to sets of values. The argument of a scalar function can be a function. However, the restrictions that apply to the use of expressions and 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.

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 - BIRTHDATE)
FROM EMPLOYEE
WHERE WORKDEPT = 'D01'
```

ABS

►► ABS (*numeric-expression*) ◀◀

The ABS function returns the absolute value of a number.

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

The data type and length attribute of the result are the same as the data type and length attribute of the argument value, except that the result is a large integer if the argument value is a small integer, and the result is double-precision floating point if the argument value is single-precision floating point.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: ABSVAL is a synonym for ABS. It is supported only for compatibility with previous DB2 releases.

Example

- Assume the host variable PROFIT is a large integer with a value of -50000.

```
SELECT ABS (:PROFIT)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 50000.

ACOS

►► ACOS(*numeric-expression*) ◄◄

The ACOS function returns the arc cosine of the argument as an angle expressed in radians. The ACOS and COS functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. The value must be greater than or equal to -1 and less than or equal to 1.

The data type of the result is double-precision floating point. 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 greater than or equal to 0 and less than or equal to PI.

Example

- Assume the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```
SELECT ACOS(:ACOSINE)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 1.49.

ANTILOG

►►—ANTILOG—(*numeric-expression*)—◄◄

The ANTILOG function returns the anti-logarithm (base 10) of a number. The ANTILOG and LOG functions are inverse operations.

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

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable ALOG is a DECIMAL(10,9) host variable with a value of 1.499961866.

```
SELECT ANTILOG(:ALOG)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 31.62.

ASIN

►► ASIN(*numeric-expression*) ◀◀

The ASIN function returns the arc sine of the argument as an angle expressed in radians. The ASIN and SIN functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. The value must be greater than or equal to -1 and less than or equal to 1.

The data type of the result is double-precision floating point. 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 greater than or equal to $-\text{PI} / 2$ and less than or equal to $\text{PI} / 2$.

Example

- Assume the host variable ASINE is a DECIMAL(10,9) host variable with a value of 0.997494987.

```
SELECT ASIN(:ASINE)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 1.50.

ATAN

►► ATAN(*numeric-expression*) ◀◀

The ATAN function returns the arc tangent of the argument as an angle expressed in radians. The ATAN and TAN functions are inverse operations.

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

The data type of the result is double-precision floating point. 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 greater than or equal to $-\pi/2$ and less than or equal to $\pi/2$.

Example

- Assume the host variable ATANGENT is a DECIMAL(10,8) host variable with a value of 14.10141995.

```
SELECT ATAN(:ATANGENT)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 1.50.

ATANH

►► ATANH(*—numeric-expression—*) ◀◀

The ATANH function returns the hyperbolic arc tangent of a number, in radians. The ATANH and TANH functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. The value must be greater than -1 and less than 1.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable HATAN is a DECIMAL(10,9) host variable with a value of 0.905148254.

```
SELECT ATANH(:HATAN)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 1.50.

ATAN2

►►—ATAN2—(—*numeric-expression1*—,—*numeric-expression2*—)——————►◄

The ATAN2 function returns the arc tangent of x and y coordinates as an angle expressed in radians. The first and second arguments specify the x and y coordinates, respectively.

Each argument is an expression that returns the value of any built-in numeric data type. Both arguments must not be 0.

The data type of the result is double-precision floating point. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Example

- Assume that host variables HATAN2A and HATAN2B are DOUBLE host variables with values of 1 and 2, respectively.

```
SELECT ATAN2 (:HATAN2A, :HATAN2B)
FROM SYSIBM.SYSDUMMY1
```

Returns a double precision floating-point number with an approximate value of 1.1071487.

BIGINT

Numeric to Big Integer

►► `BIGINT`—(*numeric-expression*)—►►

Character to Big Integer

►► `BIGINT`—(*character-expression*)—►►

The `BIGINT` function returns a big integer representation of:

- A number
- A character string representation of a decimal number
- A character string representation of an integer
- A character string representation of a floating-point number

Numeric to Big Integer

numeric-expression

An expression that returns a numeric 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 big integers, an error occurs. The fractional part of the argument is truncated.

Character to Big Integer

character-expression

An expression that returns a value that is a character-string representation of an integer. The expression must not be a CLOB.

If the argument is a *character-expression*, the result is the same number that would result from `CAST(character-expression AS BIGINT)`. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, integer, or decimal constant. If the whole part of the argument is not within the range of integers, an error occurs. Any fractional part of the argument is truncated.

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

Note

Syntax alternatives: The `CAST` specification should be used for maximal portability. For more information, see “`CAST` Specification” on page 143.

Example

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

```
SELECT BIGINT(SALARY)
FROM EMPLOYEE
```

BLOB

►► BLOB (—*string-expression* [, —*integer*]) ►►

The BLOB function returns a BLOB representation of a string of any type.

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

string-expression

A *string-expression* whose value can be a character string, graphic string, binary string, or row ID.

integer

Specifies the length attribute for the resulting binary string. The value must be between 1 and 2 147 483 647.

If *integer* is not specified:

- If the *string-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument, unless the argument is a graphic string. In this case, the length attribute of the result is twice the length attribute of the argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of the expression (or twice the length of the expression when the input is graphic data). If the length of the *string-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the first input argument is a character string and all the truncated characters are blanks, or the first input argument is a graphic string and all the truncated characters are double-byte blanks.

Note

Syntax alternatives: When the length is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Example

- The following function returns a BLOB for the string ‘This is a BLOB’.

```
SELECT BLOB('This is a BLOB')
FROM SYSIBM.SYSDUMMY1
```

- The following function returns a BLOB for the large object that is identified by locator myclob_locator.

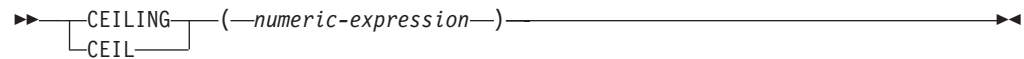
```
SELECT BLOB(:myclob_locator)
FROM SYSIBM.SYSDUMMY1
```

- Assume that a table has a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME. Locate any maps that contain the string ‘Pellow Island’ and return a single binary string with the map name concatenated in front of the actual map. The following function returns a BLOB for the large object that is identified by locator myclob_locator.

BLOB

```
SELECT BLOB( MAP_NAME CONCAT ': ' CONCAT TOPOGRAPHIC_MAP )  
FROM ONTARIO_SERIES_4  
WHERE TOPOGRAPHIC_MAP LIKE '%Pellow Island%'
```


CEILING



The CEIL or CEILING function returns the smallest integer value that is greater than or equal to *numeric-expression*.

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

The result of the function has the same data type and length attribute of the argument except that the scale is 0 if the argument is DECIMAL or NUMERIC. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(5,0).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Find the highest monthly salary for all the employees. Round the result up to the next integer. The SALARY column has a decimal data type

```
SELECT CEIL(MAX(SALARY))/12
FROM EMPLOYEE
```

This example returns 4396.00 because the highest paid employee is Christine Haas who earns \$52750.00 per year. Her average monthly salary before applying the CEIL function is 4395.83.

- Use CEILING on both positive and negative numbers.

```
SELECT CEILING( 3.5),
       CEILING( 3.1),
       CEILING(-3.1),
       CEILING(-3.5),
FROM SYSIBM.SYSDUMMY1
```

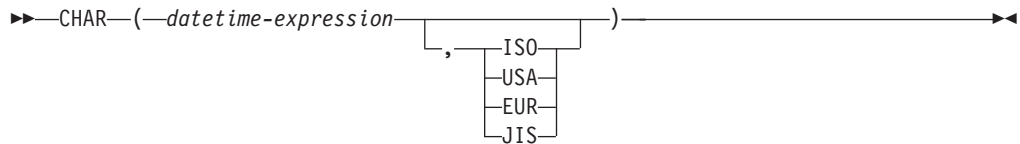
This example returns:

```
04.  04.  -03.  -03.
```

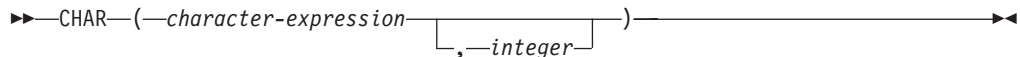
respectively.

CHAR

Datetime to Character



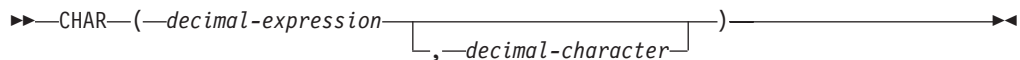
Character to Character



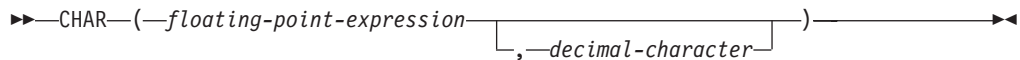
Integer to Character



Decimal to Character



Floating-point to Character



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

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT.
- A decimal number if the first argument is a decimal number.
- A double-precision floating-point number if the first argument is a DOUBLE or REAL.
- A character string if the first argument is any type of character string.
- A date value if the first argument is a DATE.
- A time value if the first argument is a TIME.
- A timestamp value if the first argument is a TIMESTAMP.
- A row ID value if the first argument is a ROWID.

The first argument must be a built-in data type other than a BLOB, GRAPHIC, VARGRAPHIC, or DBCLOB.

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 built-in data types

date The result is the character-string representation of the date in the format specified by the second argument. If the second argument is not specified, the format used is the default date format. If the format is ISO, USA, EUR, or JIS, the length of the result is 10. Otherwise the length of the result is the length of the default date format. For more information see “String Representations of Datetime Values” on page 68.

time The result is the character-string representation of the time in the format specified by the second argument. If the second argument is not specified, the format used is the default time format. The length of the result is 8. For more information see “String Representations of Datetime Values” on page 68.

timestamp

The second argument is not applicable and must not be specified.

The result is the character-string representation of the timestamp. The length of the result is 26.

The CCSID of the string is the default SBCS CCSID at the current server.

ISO, EUR, USA, or JIS

Specifies the date or time format of the resulting character string. For more information, see “String Representations of Datetime Values” on page 68.

Character to Character*character-expression*

An expression that returns a value that is a built-in character-string data type.

integer

Specifies the length attribute for the resulting fixed length character string. The value must be between 1 and 32766 (32765 if nullable). If the first argument is mixed data, the second argument cannot be less than 4.

If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length is the same as the length attribute of the result. If the length of the *character-expression* is less than the length 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 (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

The CCSID of the string is the CCSID of the *character-expression*.

Integer to Character*integer-expression*

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

CHAR

The result is the fixed-length character-string representation of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. The result is left justified.

- If the 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.
- If the 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.
- If the 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.

The CCSID of the string is the default SBCS CCSID at the current server.

Decimal to Character

decimal-expression

An expression that returns a value that is a built-in decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is desired, the DECIMAL scalar function can be used 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 must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see “Decimal Point” on page 102.

The result is a fixed-length character string representation of the argument. The result includes a decimal character and up to p digits, where p is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

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 CCSID of the string is the default SBCS CCSID at the current server.

Floating-point to Character

floating-point expression

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

decimal-character

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see “Decimal Point” on page 102.

The result is a 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 is zero, the result is 0E0. Otherwise, the result includes the smallest number of characters that can be used to 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.

The CCSID of the string is the default SBCS CCSID at the current server.

Note

Syntax alternatives: When the first argument is numeric, or the first argument is a string and the length argument is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Examples

- Assume the column PRSTDATE has an internal value equivalent to 1988-12-25. The date format is *MDY and the date separator is a slash (/).

```
|      SELECT CHAR(PRSTDATE, USA)
|      FROM PROJECT
```

Results in the value '12/25/1988'.

```
|      SELECT CHAR(PRSTDATE)
|      FROM PROJECT
```

Results in the value '12/25/88'.

- 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).

```
|      SELECT CHAR(STARTING, USA)
|      FROM CL_SCHED
```

Results in the value '5:12 PM'.

```
|      SELECT CHAR(STARTING + :HOUR_DUR, JIS)
|      FROM CL_SCHED
```

Results in the value '10:12:00'.

- Assume the column RECEIVED (timestamp) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns.

```
|      SELECT CHAR(RECEIVED)
|      FROM IN_TRAY
```

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 (SQLSTATE 01004) that the value is truncated is returned.

- Use the CHAR function to return the values for EDLEVEL (defined as SMALLINT) as a fixed length string.

```
|      SELECT CHAR(EDLEVEL)
|      FROM EMPLOYEE
```

CHAR

An EDLEVEL of 18 would be returned as the CHAR(6) value '18bbbb' (18 followed by 4 blanks).

- Assume that the STAFF table 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 returned with a comma as the decimal character (18357,50).

```
SELECT CHAR(SALARY, ',')  
FROM EMPLOYEE
```

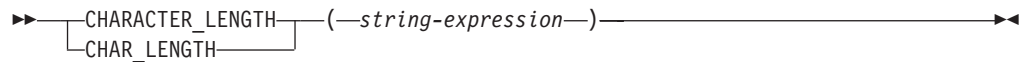
returns the value '18357,50bbb' (18357,50 followed by 3 blanks).

- Assume a host variable, DOUBLE_NUM, has a double precision floating-point data type and a value of -987.654321E-35.

```
SELECT CHAR(:DOUBLE_NUM)  
FROM SYSIBM.SYSDUMMY1
```

Results in the character value '-9.8765432100000002E-33 '.

CHARACTER_LENGTH



The CHARACTER_LENGTH or CHAR_LENGTH function returns the length of a string expression. See “LENGTH” on page 255 for a similar function.

The argument must be an expression that returns a value of any built-in string 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 number of characters in the argument (not the number of bytes). A single character is either an SBCS or DBCS character. The length of strings includes trailing blanks. The length of a varying-length string is the actual length, not the maximum length.

Example

- Assume the host variable ADDRESS is a varying-length character string with a value of ‘895 Don Mills Road’.

```
SELECT CHARACTER_LENGTH(:ADDRESS)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 18.

CLOB

CLOB

Character to CLOB

►► CLOB (*—character-expression* , *—length* DEFAULT , *—integer*)

Graphic to CLOB

►► CLOB (*—graphic-expression* , *—length* DEFAULT , *—integer*)

Integer to CLOB

►► CLOB (*—integer-expression*)

Decimal to CLOB

►► CLOB (*—decimal-expression* , *—decimal-character*)

Floating-point to CLOB

►► CLOB (*—floating-point-expression* , *—decimal-character*)

The CLOB function returns a character-string representation of:

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number if the first argument is a packed or zoned decimal number
- A double-precision floating-point number if the first argument is a DOUBLE or REAL
- A character string if the first argument is any type of character string
- A graphic string if the first argument is an UCS-2 graphic string

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

Character to CLOB

character-expression

An expression that returns a value that is a built-in character-string data type.

length

Specifies the length attribute for the resulting varying length character string. The value must be between 1 and 2 147 483 647. If the first argument is mixed data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of the *character-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer

Specifies the CCSID of the result. It must be a valid SBCS CCSID or mixed data CCSID. If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. If the third argument is a SBCS CCSID, then the first argument cannot be a DBCS-either or DBCS-only string. The third argument cannot be 65535.

If the third argument is not specified, the first argument must not have a CCSID of 65535:

- If the first argument is bit data, an error occurs.
- If the first argument is SBCS data, then the result is SBCS data. The CCSID of the result is the same as the CCSID of the first argument.
- If the first argument is mixed data (DBCS-open, DBCS-only, or DBCS-either), then the result is mixed data. The CCSID of the result is the same as the CCSID of the first argument.

Graphic to CLOB

graphic-expression

An expression that returns a value that is a built-in graphic-string data type. It must not be DBCS-graphic data.

length

Specifies the length attribute for the resulting varying length character string. The value must be between 1 and 2 147 483 647. If the result is mixed data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified, the length attribute of the result is determined as follows (where *n* is the length attribute of the first argument):

- If the *graphic-expression* is the empty graphic string constant, the length attribute of the result is 1.
- If the result is SBCS data, the result length is *n*.
- If the result is mixed data, the result length is $(2.5*(n-1)) + 4$.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer

Specifies the CCSID of the result. It must be a valid SBCS CCSID or mixed data CCSID. If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. The third argument cannot be 65535.

CLOB

If the third argument is not specified, the CCSID of the result is the default CCSID at the current server. If the default CCSID is mixed data, then the result is mixed data. If the default CCSID is SBCS data, then the result is SBCS data.

Integer to CLOB

integer-expression

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

The result is a varying-length character string of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. The result is left justified.

- If the argument is a small integer, the length attribute of the result is 6.
- If the argument is a large integer, the length attribute of the result is 11.
- If the argument is a big integer, the length attribute of the result is 20.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit.

The CCSID of the result is the default SBCS CCSID at the current server.

Decimal to CLOB

decimal-expression

An expression that returns a value that is a built-in decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is desired, the DECIMAL scalar function can be used 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 must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see “Decimal Point” on page 102.

The result is a varying-length character string representation of the argument. The result includes a decimal character and up to p digits, where p is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is $2+p$ where p is the precision of the *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing characters are included. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit.

The CCSID of the result is the default SBCS CCSID at the current server.

Floating-point to CLOB

floating-point expression

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

decimal-character

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see “Decimal Point” on page 102.

The result is a varying-length character string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If the argument is zero, the result is 0E0.

The CCSID of the result is the default SBCS CCSID at the current server.

Note

Syntax alternatives: When the first argument is numeric, or the first argument is a string and the length argument is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

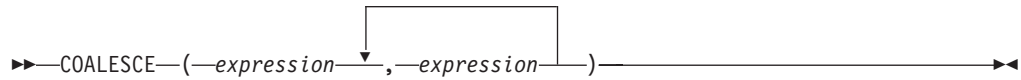
Example

- The following function returns a CLOB for the string 'This is a CLOB'.

```
SELECT CLOB('This is a CLOB')
FROM SYSIBM.SYSDUMMY1
```

COALESCE

COALESCE



The COALESCE function returns the value of the first non-null expression.

The arguments must be compatible. Character-string arguments are compatible with datetime values. For more information about data type compatibility, see “Assignments and Comparisons” on page 80. The arguments can be of either a built-in data type or a distinct type.³¹

The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. The result can be null only if all arguments can be null, and the result is null only if all arguments are null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands as explained in “Rules for Result Data Types” on page 93.

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
```

31. This function cannot be used as a source function when creating a user-defined function. Because it accepts any compatible data types as arguments, it is not necessary to create additional signatures to support distinct types.

CONCAT

►►—CONCAT—(—*string-expression-1*—,—*string-expression-2*—)—————►►

The CONCAT function combines two string arguments. The arguments must be compatible strings. For more information about data type compatibility, see “Assignments and Comparisons” on page 80.

The result of the function is a string that consists of the first argument string followed by the second. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The CONCAT function is identical to the CONCAT operator. For more information, see “With the Concatenation Operator” on page 131.

Example

- Concatenate the column FIRSTNAME with the column LASTNAME.

```
SELECT CONCAT(FIRSTNAME, LASTNAME)
FROM EMPLOYEE
WHERE EMPNO = '000010'
```

Returns the value 'CHRISTINEHAAS'.

COS

►► `COS` (*—numeric-expression—*) ◀◀

The COS function returns the cosine of the argument, where the argument is an angle expressed in radians. The COS and ACOS functions are inverse operations.

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

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable COSINE is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT COS(:COSINE)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 0.07.

COSH

►► `COSH` (*—numeric-expression—*) ◀◀

The COSH function returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.

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

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable HCOS is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT COSH (:HCOS)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 2.35.

COT

►► COT (*numeric-expression*) ◀◀

The COT function returns the cotangent of the argument, where the argument is an angle expressed in radians.

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

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable COTAN is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT COT(:COTAN)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 0.07.

CURDATE

►►—CURDATE—(—)—◄◄

The CURDATE function returns a date based on a reading of the time-of-day clock when the SQL statement is executed at the current server. The value returned by the CURDATE function is the same as the value returned by the CURRENT DATE special register.

The data type of the result is a date. The result cannot be null.

If this function is used more than once within a single SQL statement, or used with the CURTIME or NOW scalar functions or the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers within a single statement, all values are based on a single clock reading.

Example

- Return the current date based on the time-of-day clock.

```
SELECT CURDATE()  
FROM SYSIBM.SYSDUMMY1
```

CURTIME

CURTIME

►►—CURTIME—(—)—◄◄

The CURTIME function returns a time based on a reading of the time-of-day clock when the SQL statement is executed at the current server. The value returned by the CURTIME function is the same as the value returned by the CURRENT TIME special register.

The data type of the result is a time. The result cannot be null.

If this function is used more than once within a single SQL statement, or used with the CURDATE or NOW scalar functions or the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers within a single statement, all values are based on a single clock reading.

Example

- Return the current time based on the time-of-day clock.

```
SELECT CURTIME()  
FROM SYSIBM.SYSDUMMY1
```

DATE

►►—DATE—(—*expression*—)——►►

The DATE function returns a date from a value.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or any numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be one of the following:
 - A valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.
 - A character string with an actual length of 7 that represents a valid date in the form *yyyymmm*, where *yyyy* are digits denoting a year, and *mmm* are digits between 001 and 366 denoting a day of that year.
- If *expression* is a number, it must be a positive number less than or equal to 3652059.

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

The other rules depend on the data type of the argument:

- If the argument is a timestamp:

The result is the date part of the timestamp.
- If the argument is a date:

The result is that date.
- If the argument is a number:

The result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.
- If the argument is a character string:

The result is the date represented by the string or the date part of the timestamp value represented by the string.

When a string representation of a date is SBCS data with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a date value.

When a string representation of a date is mixed data with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a date value.

Note

Syntax alternatives: When the argument is a date, timestamp, or character string, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

DATE

Examples

- Assume that the column RECEIVED (TIMESTAMP) has an internal value equivalent to '1988-12-25-17.12.30.000000'.

```
SELECT DATE(RECEIVED)
FROM IN_TRAY
```

Results in an internal representation of '1988-12-25'.

- The following DATE scalar function applied to an ISO string representation of a date:

```
SELECT DATE('1988-12-25')
FROM SYSIBM.SYSDUMMY1
```

Results in an internal representation of '1988-12-25'.

- The following DATE scalar function applied to an EUR string representation of a date:

```
SELECT DATE('25.12.1988')
FROM SYSIBM.SYSDUMMY1
```

Results in an internal representation of '1988-12-25'.

- The following DATE scalar function applied to a positive number:

```
SELECT DATE(35)
FROM SYSIBM.SYSDUMMY1
```

Results in an internal representation of '0001-02-04'.

DAY

►►—DAY—(—*expression*—)—————►►

The DAY function returns the day part of a value.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.
- If *expression* is a number, it must be a date duration or timestamp duration. For the valid formats of datetime durations, see “Datetime Operands and Durations” on page 134.

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 character-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.

- Return the day part of the difference between two dates:

```
SELECT DAY( DATE('2000-03-15') - DATE('1999-12-31') )
  FROM SYSIBM.SYSDUMMY1
```

Results in the value 15.

DAYOFMONTH

DAYOFMONTH

►►—DAYOFMONTH—(—*expression*—)—————►►

The DAYOFMONTH function returns an integer between 1 and 31 that represents the day of the month.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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 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 DAYOFMONTH(PRENDATE)
  INTO :END_DAY
  FROM PROJECT
  WHERE PROJNAME = 'WELD LINE PLANNING'
```

Results in END_DAY being set to 15.

DAYOFWEEK

►—DAYOFWEEK—(—*expression*—)—————►

The DAYOFWEEK function returns an integer between 1 and 7 that represents the day of the week, where 1 is Sunday and 7 is Saturday. For another alternative, see “DAYOFWEEK_ISO” on page 212.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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

Example

- Using the EMPLOYEE table, set the host variable DAY_OF_WEEK (INTEGER) to the day of the week that Christine Haas (EMPNO='000010') started (HIREDATE).

```
SELECT DAYOFWEEK(HIREDATE)
      INTO :DAY_OF_WEEK
      FROM EMPLOYEE
      WHERE EMPNO = '000010'
```

Results in DAY_OF_WEEK being set to 6, which represents Friday.

- The following query returns four values: 1, 2, 1, and 2.

```
SELECT DAYOFWEEK(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK(TIMESTAMP('10/12/1998','01.02')),
       DAYOFWEEK(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK(CAST(TIMESTAMP('10/12/1998','01.02') AS CHAR(20))),
      FROM SYSIBM.SYSDUMMY1
```

DAYOFWEEK_ISO

DAYOFWEEK_ISO

►►—DAYOFWEEK_ISO—(—*expression*—)—————►►

The DAYOFWEEK_ISO function returns an integer between 1 and 7 that represents the day of the week, where 1 is Monday and 7 is Sunday. For another alternative, see “DAYOFWEEK” on page 211.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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, set the host variable DAY_OF_WEEK (INTEGER) to the day of the week that Christine Haas (EMPNO='000010') started (HIREDATE).

```
SELECT DAYOFWEEK_ISO(HIREDATE)
       INTO :DAY_OF_WEEK
       FROM EMPLOYEE
       WHERE EMPNO = '000010'
```

Results in DAY_OF_WEEK being set to 5, which represents Friday.

- The following query returns four values: 7, 1, 7, and 1.

```
SELECT DAYOFWEEK_ISO(CAST('10/11/1998' AS DATE)),
       DAYOFWEEK_ISO(TIMESTAMP('10/12/1998','01.02')),
       DAYOFWEEK_ISO(CAST(CAST('10/11/1998' AS DATE) AS CHAR(20))),
       DAYOFWEEK_ISO(CAST(TIMESTAMP('10/12/1998','01.02') AS CHAR(20))),
       FROM SYSIBM.SYSDUMMY1
```


DAYOFYEAR

►►—DAYOFYEAR—(*expression*)—◄◄

The DAYOFYEAR function returns an integer between 1 and 366 that represents the day of the year where 1 is January 1.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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

Example

- Using the EMPLOYEE table, set the host variable AVG_DAY_OF_YEAR (INTEGER) to the average of the day of the year that employees started on (HIREDATE).

```
SELECT AVG(DAYOFYEAR(HIREDATE))
INTO :AVG_DAY_OF_YEAR
FROM EMPLOYEE
```

Results in AVG_DAY_OF_YEAR being set to 202.

DAYS

►►—DAYS—(—*expression*—)—————►►

The DAYS function returns an integer representation of a date.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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 (INTEGER) 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'
```

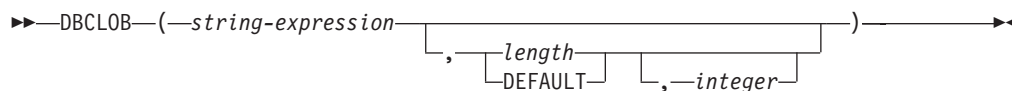
Results in EDUCATION_DAYS being set to 396.

- Using the PROJECT table, set the host variable TOTAL_DAYS (INTEGER) 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 1484.

DBCLOB



The DBCLOB function returns a DBCLOB representation of a string expression.

string-expression

An expression that returns a value that is a character-string or graphic-string. It cannot be a BLOB. It cannot be CHAR or VARCHAR bit data. It cannot be GRAPHIC or VARGRAPHIC with a CCSID of 65535 unless a third argument is specified.

length

Specifies the length attribute for the resulting varying-length graphic string. The value must be between 1 and 1 073 741 823.

If the second argument is not specified or DEFAULT is specified:

- If the *expression* is an empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

integer

Specifies the CCSID for the resulting varying-length graphic string. It must be a DBCS or UCS-2 CCSID. The CCSID cannot be 65535.

In the following rules, S denotes one of the following:

- If the string expression is a host variable containing data in a foreign encoding scheme, S is the result of the expression after converting the data to a CCSID in a native encoding scheme. (See “Character Conversion” on page 32 for more information.)
- If the string expression is data in a native encoding scheme, S is that string expression.

If the third argument is not specified and the first argument is character, then the CCSID of the result is determined by a mixed CCSID. Let M denote that mixed CCSID. M is determined as follows:

- If the CCSID of S is a mixed CCSID, M is that CCSID.
- If the CCSID of S is an SBCS CCSID:
 - If the CCSID of S has an associated mixed CCSID, M is that CCSID.
 - Otherwise the operation is not allowed.

The following table summarizes the result CCSID based on M.

M	Result CCSID	Description	DBCS Substitution Character
930	300	Japanese EBCDIC	X'FEFE'
933	834	Korean EBCDIC	X'FEFE'
935	837	S-Chinese EBCDIC	X'FEFE'
937	835	T-Chinese EBCDIC	X'FEFE'
939	300	Japanese EBCDIC	X'FEFE'

DBCLOB

M	Result CCSID	Description	DBCS Substitution Character
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'

If the third argument is not specified and the first argument is not character, then the CCSID of the result is the same as the CCSID of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *expression*. If the length attribute of the resulting DBCLOB is less than the actual length of the first argument, truncation is performed and no warning is returned.

The result of the function is a DBCLOB string. If the expression can be null, the result can be null. If the expression is null, the result is the null value. If the expression is an empty string or the EBCDIC string X'0E0F', the result is an empty string.

If the result is DBCS-graphic data, the equivalence of SBCS and DBCS characters depends on M. Regardless of the CCSID, every double-byte code point in the argument is considered a DBCS character, and every single-byte code point in the argument is considered an SBCS character with the exception of the EBCDIC mixed data shift codes X'0E' and X'0F'.

- If the nth character of the argument is a DBCS character, the nth character of the result is that DBCS character.
- If the nth character of the argument is an SBCS character that has an equivalent DBCS character, the nth character of the result is that equivalent DBCS character.
- If the nth character of the argument is an SBCS character that does not have an equivalent DBCS character, the nth character of the result is the DBCS substitution character.

If the result is UCS-2 graphic data, each character of the argument determines a character of the result. The nth character of the result is the UCS-2 equivalent of the nth character of the argument.

Note

Syntax alternatives: When the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

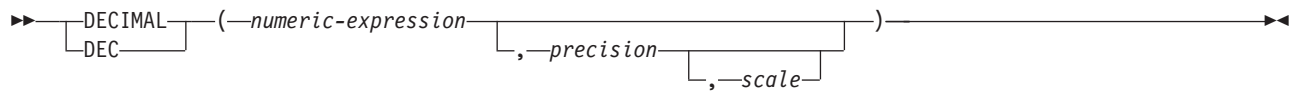
Example

- Using the EMPLOYEE table, set the host variable VAR_DESC (VARGRAPHIC(24)) to the DBCLOB equivalent of the first name (FIRSTNME) for employee number (EMPNO) '000050'.

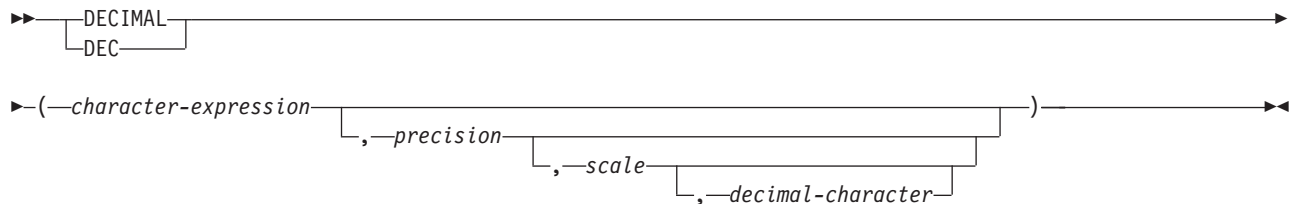
```
SELECT DBCLOB(FIRSTNME)
  INTO :VAR_DESC
 FROM EMPLOYEE
 WHERE EMPNO = '000050'
```

DECIMAL or DEC

Numeric to Decimal



Character to Decimal



The DECIMAL function returns a decimal representation of:

- A number
- A character string representation of a decimal number
- A character string representation of an integer
- A character string representation of a floating-point number

Numeric to Decimal

numeric-expression

An expression that returns a value of any built-in numeric data type.

precision

An integer constant with a value greater than or equal to 1 and less than or equal to 31.

The default for *precision* depends on the data type of the *numeric-expression*:

- 15 for floating point, decimal, numeric, or nonzero scale binary
- 19 for big integer
- 11 for large integer
- 5 for small integer

scale

An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

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*. 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 must contain a character-string representation of a number.

DECIMAL

Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer or decimal constant. The expression must not be a CLOB.

precision

An integer constant that is greater than or equal to 1 and less than or equal to 31. If not specified, the default is 15.

scale

An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

decimal-character

Specifies the single-byte character constant that was used to delimit the decimal digits in *character-expression* from the whole part of the number. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal separator character. For more information, see "Decimal Point" on page 102.

The result is the same number that would result from `CAST(character-expression AS DECIMAL(p,s))`. Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *s*. An error occurs if the number of significant digits to the left of the decimal character (the whole part of the number) in *character-expression* is greater than *p-s*. The default decimal character is not valid in the substring if the *decimal-character* argument is specified.

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.

Note

Syntax alternatives: When the precision is specified, the CAST specification should be used for maximal portability. For more information, see "CAST Specification" on page 143.

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
```

- Using the PROJECT table, select all of the starting dates (PRSTDATE) that have been incremented by a duration that is specified in a host variable. 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 SALARY becomes 21400.50.

DEGREES

DEGREES

►►—DEGREES—(*—numeric-expression—*)——————►◄

The DEGREES function returns the number of degrees of the argument which is an angle expressed in radians.

The argument must be an expression that returns the value of any built-in numeric data type.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable RAD is a DECIMAL(4,3) host variable with a value of 3.142.

```
SELECT DEGREES(:RAD)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 180.0.

DIFFERENCE

►►—DIFFERENCE—(—*string-expression-1*—,—*string-expression-2*—)—————►◄

The DIFFERENCE function 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 must be a built-in string data types, but not BLOBs, CLOBs, and DBCLOBs.

The data type of the result is INTEGER. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- Assume the following statement:

```
SELECT DIFFERENCE('CONSTRAINT','CONSTANT'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONSTANT')
FROM SYSIBM.SYSDUMMY1
```

Returns 4, C523, and C523. Since the two strings return the same SOUNDEX value, the difference is 4 (the highest value possible).

- Assume the following statement:

```
SELECT DIFFERENCE('CONSTRAINT','CONTRITE'),
       SOUNDEX('CONSTRAINT'),
       SOUNDEX('CONTRITE')
FROM SYSIBM.SYSDUMMY1
```

Returns 2, C523, and C536. In this case, the two strings return different SOUNDEX values, and hence, a lower difference value.

DIGITS

►►—DIGITS—(—*numeric-expression*—)—————►►

The DIGITS function returns a character-string representation of the absolute value of a number.

The argument must be a built-in numeric data type of 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 point. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5, if the argument is a small zero scale integer
- 10, if the argument is a large zero scale integer
- 19, if the argument is a big integer
- p , if the argument is a decimal or nonzero scale integer with a precision of p

The CCSID of the character string is the default SBCS CCSID at the current server.

Examples

- Assume that a table called TABLEX contains an INTEGER column called INTCOL containing 10-digit numbers. List all 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.

```
SELECT DIGITS(COLUMNX)
FROM TABLEX
```

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—(—*DataLink-expression*—)—————►►

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 a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is VARCHAR(254).

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- 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:

```
INSERT INTO TBLA
VALUES (DLVALUE('http://d1fs.almaden.ibm.com/x/y/a.b', 'URL', 'A comment'))
```

then the following function operating on that value:

```
SELECT DLCOMMENT(COLA)
FROM TBLA
```

Returns the value 'A comment'.

DLLINKTYPE

DLLINKTYPE

►—DLLINKTYPE—(—*DataLink-expression*—)——►

The DLLINKTYPE function returns the link type value from a DataLink value.

The argument must be an expression that results in a value with a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is VARCHAR(4).

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
VALUES( DLVALUE('http://d1fs.almaden.ibm.com/x/y/a.b', 'URL', 'A comment') )
```

then the following function operating on that value:

```
SELECT DLLINKTYPE(COLA)
FROM TBLA
```

Returns the value 'URL'.

DLURLCOMPLETE

►►—DLURLCOMPLETE—(—*DataLink-expression*—)—————►►

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, and then DLURLPATH. If the DataLink has an attribute of FILE LINK CONTROL and READ PERMISSION DB, the value includes a file access token.

The argument must be an expression that results in a value with a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string. The length attribute depends on the attributes of the DataLink:

- If the DataLink has an attribute of FILE LINK CONTROL and READ PERMISSION DB, the length attribute of the result is the length attribute of the argument plus 19.
- Otherwise, the length attribute of the result is the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b', 'URL', 'A comment') )
```

then the following function operating on that value:

```
SELECT DLURLCOMPLETE(COLA)
FROM TBLA
```

Returns the value

'HTTP://DLFS.ALMADEN.IBM.COM/x/y/*****;a.b', where
***** represents the access token.

DLURLPATH

DLURLPATH

►►—DLURLPATH—(—*DataLink-expression*—)—————►►

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 a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string. The length attribute depends on the attributes of the DataLink:

- If the DataLink has an attribute of FILE LINK CONTROL and READ PERMISSION DB, the length attribute of the result is the length attribute of the argument plus 19.
- Otherwise, the length attribute of the result is the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
VALUES( DLVALUE('http://d1fs.almaden.ibm.com/x/y/a.b', 'URL', 'A comment') )
```

then the following function operating on that value:

```
SELECT DLURLPATH(COLA)
FROM TBLA
```

Returns the value '/x/y/*****;a.b', where ***** represents the access token.

DLURLPATHONLY

►►—DLURLPATHONLY—(—*DataLink-expression*—)—————►►

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 a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string with a length attribute of that is equal to the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
VALUES( DLVALUE('http://d1fs.almaden.ibm.com/x/y/a.b', 'URL', 'A comment') )
```

then the following function operating on that value:

```
SELECT DLURLPATHONLY(COLA)
FROM TBLA
```

Returns the value '/x/y/a.b'.

DLURLSCHEME

DLURLSCHEME

►►—DLURLSCHEME—(—*DataLink-expression*—)—————►►

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 a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is VARCHAR(20).

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
VALUES( DLVALUE('http://d1fs.almaden.ibm.com/x/y/a.b', 'URL', 'A comment') )
```

then the following function operating on that value:

```
SELECT DLURLSCHEME(COLA)
FROM TBLA
```

Returns the value 'HTTP'.

DLURLSERVER

►►—DLURLSERVER—(—*DataLink-expression*—)—————►►

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 a built-in DataLink data type.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a varying-length string with a length attribute of that is equal to the length attribute of the argument.

If the DataLink value only includes the comment, the result returned is a zero length string.

The CCSID of the character string is the same as that of *DataLink-expression*.

Examples

- Given a DataLink value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
INSERT INTO TABLA
VALUES( DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b', 'URL', 'A comment') )
```

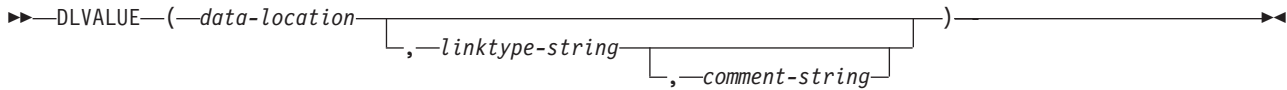
then the following function operating on that value:

```
SELECT DLURLSERVER(COLA)
FROM TBLA
```

Returns the value 'DLFS.ALMADEN.IBM.COM'.

DLVALUE

DLVALUE



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 a character string expression that contains a complete URL value. If the expression is not an empty string, it must include the URL scheme and URL server. The actual length of the character string expression must be less than or equal to 32718 characters.

linktype-string

An optional character string expression that specifies the link type of the DataLink value. The only valid value is 'URL'.

comment-string

An optional character string expression that provides a comment or additional location information. The actual length of the character string expression must be less than or equal to 254 characters.

The *comment-string* cannot be the null value. If a *comment-string* is not specified, the *comment-string* is the empty string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The result of the function is a DataLink value.

The CCSID of the DataLink is the same as that of *data-location* except in the following cases:

- If the *comment string* is mixed data and *data-location* is not mixed data, the CCSID of the result will be the CCSID of the *comment string*.³²
- If the *data-location* has a CCSID of bit data (65535), UCS-2 graphic data (13488), Turkish data (905 or 1026), or Japanese data (290, 930, or 5026); the CCSID of the result is described in the following table:

CCSID of <i>data-location</i>	CCSID of <i>comment-string</i>	Result CCSID
65535	65535	Job Default CCSID
65535	non-65535	<i>comment-string</i> CCSID (unless the CCSID is 290, 930, 5026, 905, 1026, or 13488 where the CCSID will then be further modified as described in the following rows.)
290	any	4396
930 or 5026	any	939

32. If the CCSID of *comment string* is 5026 or 930, the CCSID of the results will be 939.

CCSID of <i>data-location</i>	CCSID of <i>comment-string</i>	Result CCSID
905 or 1026	any	500
13488	any	500

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.

Examples

- 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`.

```

INSERT INTO HOCKEY_GOALS
VALUES('Maurice Richard',
      'Montreal canadian',
      '?',
      '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) )

```


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. To eliminate the possibility of out-of-range results, DOUBLE_PRECISION is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, DOUBLE_PRECISION(SALARY)/COMM
FROM EMPLOYEE
WHERE COMM > 0
```

EXP

►►—EXP—(*numeric-expression*)—◄◄

The EXP function returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument. The EXP and LN functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable E is a DECIMAL(10,9) host variable with a value of 3.453789832.

```
SELECT EXP(:E)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 31.62.

FLOAT

Numeric to Float

►►—FLOAT—(*—numeric-expression—*)—►►

Character to Float

►►—FLOAT—(*—character-expression—*)—►►

The FLOAT function returns a floating point representation of a number.

FLOAT is a synonym for the DOUBLE_PRECISION and DOUBLE functions. For more information, see “DOUBLE_PRECISION or DOUBLE” on page 232.

FLOOR

►►—FLOOR—(*numeric-expression*)—◄◄

The FLOOR function returns the largest integer value less than or equal to *numeric-expression*.

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

The result of the function has the same data type and length attribute of the argument except that the scale is 0 if the argument is a decimal number. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(5,0).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Use the FLOOR function to truncate any digits to the right of the decimal point.

```
SELECT FLOOR(SALARY)
FROM EMPLOYEE
```

- Use FLOOR on both positive and negative numbers.

```
SELECT FLOOR( 3.5),
       FLOOR( 3.1),
       FLOOR(-3.1),
       FLOOR(-3.5),
FROM SYSIBM.SYSDUMMY1
```

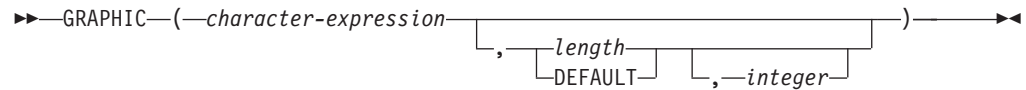
This example returns:

```
3.    3.   -4.  -4.
```

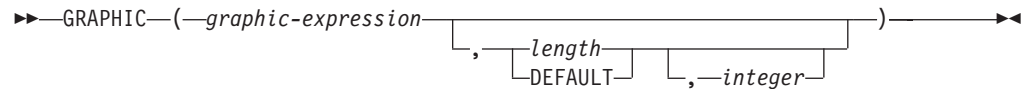
respectively.

GRAPHIC

Character to Graphic



Graphic to Graphic



The `GRAPHIC` function returns a fixed-length graphic-string representation of a string expression.

The result of the function is a fixed-length graphic string (`GRAPHIC`).

If the expression can be null, the result can be null. If the expression is null, the result is the null value.

Character to Graphic

character-expression

Specifies a character string expression. It cannot be a `CHAR` or `VARCHAR` bit data. If the expression is an empty string or the EBCDIC string `X'0E0F'`, the result is an empty string.

length

Specifies the length attribute of the result and must be an integer constant between 1 and 16383 if the first argument is not nullable or between 1 and 16382 if the first argument is nullable. If the length of *character-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If *length* is not specified, or if `DEFAULT` is specified, the length attribute of the result is the same as the length attribute of the first argument.

Each character of the argument determines a character of the result. If the length attribute of the resulting fixed-length string is less than the actual length of the first argument, truncation is performed and no warning is returned.

integer

Specifies the `CCSID` of the result. It must be a `DBCS` or `UCS-2 CCSID`. The `CCSID` cannot be 65535. If the `CCSID` represents `UCS-2` graphic data, each character of the argument determines a character of the result. The *n*th character of the result is the `UCS-2` equivalent of the *n*th character of the argument.

If *integer* is not specified then the `CCSID` of the result is determined by a mixed `CCSID`. Let *M* denote that mixed `CCSID`.

In the following rules, *S* denotes one of the following:

GRAPHIC

- If the string expression is a host variable containing data in a foreign encoding scheme, S is the result of the expression after converting the data to a CCSID in a native encoding scheme. (See “Character Conversion” on page 32 for more information.)
- If the string expression is data in a native encoding scheme, S is that string expression.

M is determined as follows:

- If the CCSID of S is a mixed CCSID, M is that CCSID.
- If the CCSID of S is an SBCS CCSID:
 - If the CCSID of S has an associated mixed CCSID, M is that CCSID.
 - Otherwise the operation is not allowed.

The following table summarizes the result CCSID based on M.

M	Result CCSID	Description	DBCS Substitution Character
930	300	Japanese EBCDIC	X'FEFE'
933	834	Korean EBCDIC	X'FEFE'
935	837	S-Chinese EBCDIC	X'FEFE'
937	835	T-Chinese EBCDIC	X'FEFE'
939	300	Japanese EBCDIC	X'FEFE'
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'

The equivalence of SBCS and DBCS characters depends on M. Regardless of the CCSID, every double-byte code point in the argument is considered a DBCS character, and every single-byte code point in the argument is considered an SBCS character with the exception of the EBCDIC mixed data shift codes X'0E' and X'0F'.

- If the nth character of the argument is a DBCS character, the nth character of the result is that DBCS character.
- If the nth character of the argument is an SBCS character that has an equivalent DBCS character, the nth character of the result is that equivalent DBCS character.
- If the nth character of the argument is an SBCS character that does not have an equivalent DBCS character, the nth character of the result is the DBCS substitution character.

Graphic to Graphic

graphic-expression

Specifies a graphic string expression.

length

Specifies the length attribute of the result and must be an integer constant between 1 and 16383 if the first argument is not nullable or between 1 and 16382 if the first argument is nullable. If the length of *graphic-expression* is less than the length specified, the result is padded with double-byte blanks to the length of the result.

If the second argument is not specified, or if DEFAULT is specified, the length attribute of the result is the same as the length attribute of the first argument.

GRAPHIC

If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer

Specifies the CCSID of the result. It must be a DBCS or UCS-2 CCSID. The CCSID cannot be 65535.

If *integer* is not specified then the CCSID of the result is the CCSID of the first argument.

Note

Syntax alternatives: If the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Example

- Using the EMPLOYEE table, set the host variable DESC (GRAPHIC(24)) to the GRAPHIC equivalent of the first name (FIRSTNME) for employee number (EMPNO) '000050'.

```
SELECT GRAPHIC( VARGRAPHIC(FIRSTNME))
  INTO :DESC
  FROM EMPLOYEE
 WHERE EMPNO = '000050'
```

HASH

HASH



The HASH function returns the partition number of a set of values. Also see the PARTITION function. For more information about partition numbers, see the DB2 Multisystem book.

The arguments can be any built-in data type except date, time, timestamp, floating-point, or DataLink values.

The result of the function is a large integer with a value between 0 and 1023.

If any of the arguments are null, the result is zero. The result cannot be null.

Example

- Use the HASH function to determine what the partitions would be if the partitioning key was composed of EMPNO and LASTNAME. This query returns the partition number for every row in EMPLOYEE.

```
SELECT HASH(EMPNO, LASTNAME)  
FROM EMPLOYEE
```

HEX

►►—HEX—(*expression*)—◄◄

The HEX function returns a hexadecimal representation of a value.

The argument can be of any built-in data type.

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

The result is a string of hexadecimal digits, the first two digits represent the first byte of the argument, the next two digits represent the second byte of the argument, and so forth. If the argument is a datetime value, the result is the hexadecimal representation of the internal form of the argument.³³

The length attribute of the result is twice the storage length attribute of the argument. For information on the storage length attribute see “CREATE TABLE” on page 525. The length attribute of the result cannot be greater than 32766 for fixed-length results or greater than 32740 for varying-length results.

If the argument is a varying-length string, the result is a varying-length string. Otherwise, the result is a fixed-length string.

The CCSID of the string is the default SBCS CCSID at the current server.

Example

- Use the HEX function to return a hexadecimal representation of the education level for each employee.

```
SELECT FIRSTNAME, MIDDLENAME, LASTNAME, HEX(EDLEVEL)
FROM EMPLOYEE
```

33. This hexadecimal representation for DATE, TIMESTAMP, and NUMERIC data types is different from other database products because the internal form for these data types is different.

HOURL

▶▶—HOURL—(—*expression*—)—————▶▶

The HOURL function returns the hour part of a value.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string or a numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see “String Representations of Datetime Values” on page 68.
- If *expression* is a number, it must be a time duration or timestamp duration. For the valid formats of datetime durations, see “Datetime Operands and Durations” on page 134.

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 character-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 HOURL(STARTING) BETWEEN 12 AND 17
```

IDENTITY_VAL_LOCAL

►►—IDENTITY_VAL_LOCAL—(—)—◄◄

IDENTITY_VAL_LOCAL is a non-deterministic function that returns the most recently assigned value for an identity column.

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

The value returned is the value that was assigned to the identity column of the table identified in the most recent INSERT statement for a table containing an identity column. The INSERT statement has to be issued at the same level; that is, the value has to be available locally within the level at which it was assigned until replaced by the next assigned value. A new level is initiated when a trigger, function, or stored procedure is invoked. A trigger condition is at the same level as the associated triggered action.

The assigned value can be a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT) or an identity value that was generated by the database manager.

The result can be null. The result is null if an INSERT statement has not been issued for a table containing an identity column at the current processing level. This includes invoking the function in a before or after insert trigger.

The result of the IDENTITY_VAL_LOCAL function is not affected by the following statements:

- An INSERT statement for a table which does not contain an identity column
- An UPDATE statement
- A COMMIT statement
- A ROLLBACK statement

Notes

The following notes explain the behavior of the function when it is invoked in various situations:

Invoking the function within the VALUES clause of an INSERT statement

Expressions in an INSERT statement are evaluated before values are assigned to the target columns of the INSERT statement. Thus, when you invoke IDENTITY_VAL_LOCAL in an INSERT statement, the value that is used is the most recently assigned value for an identity column from a previous INSERT statement. The function returns the null value if no such INSERT statement had been executed within the same level as the invocation of the IDENTITY_VAL_LOCAL function.

Invoking the function following a failed INSERT statement

The function returns an unpredictable result when it is invoked after the unsuccessful execution of an INSERT statement for a table with an identity column. The value might be the value that would have been returned from the function had it been invoked before the failed INSERT or the value that would have been assigned had the INSERT succeeded. The actual value returned depends on the point of failure and is therefore unpredictable.

IDENTITY_VAL_LOCAL

Invoking the function within the SELECT statement of a cursor

Because the results of the IDENTITY_VAL_LOCAL function are not deterministic, the result of an invocation of the IDENTITY_VAL_LOCAL function from within the SELECT statement of a cursor can vary for each FETCH statement.

Invoking the function within the trigger condition of an insert trigger

The result of invoking the IDENTITY_VAL_LOCAL function from within the condition of an insert trigger is the null value.

Invoking the function within a triggered action of an insert trigger

Multiple before or after insert triggers can exist for a table. In such cases, each trigger is processed separately, and identity values generated by SQL statements issued within a triggered action are not available to other triggered actions using the IDENTITY_VAL_LOCAL function. This is the case even though the multiple triggered actions are conceptually defined at the same level.

Do not use the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger. The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of a before insert trigger is the null value. The value for the identity column of the table for which the trigger is defined cannot be obtained by invoking the IDENTITY_VAL_LOCAL function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.

The result of invoking the IDENTITY_VAL_LOCAL function in the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent INSERT statement invoked in the same triggered action for a table containing an identity column. If an INSERT statement for a table containing an identity column was not executed within the same triggered action before invoking the IDENTITY_VAL_LOCAL function, then the function returns a null value.

Invoking the function following an INSERT with triggered actions

The result of invoking the function after an INSERT that activates triggers is the value actually assigned to the identity column (that is, the value that would be returned on a subsequent SELECT statement). This value is not necessarily the value provided in the INSERT statement or a value generated by the database manager. The assigned value could be a value that was specified in a SET transition variable statement within the triggered action of a before insert trigger for a trigger transition variable associated with the identity column.

Examples

- Set the variable IVAR to the value assigned to the identity column in the EMPLOYEE table. The value returned from the function in the VALUES statement should be 1.

```
CREATE TABLE EMPLOYEE
  (EMPNO INTEGER GENERATED ALWAYS AS IDENTITY,
   NAME CHAR(30),
   SALARY DECIMAL(5,2),
   DEPT SMALLINT)
```

```
INSERT INTO EMPLOYEE
  (NAME, SALARY, DEPTNO)
```



```
VALUES('Rupert', 989.99, 50)
```

```
VALUES IDENTITY_VAL_LOCAL() INTO :IVAR
```

- Assume two tables, T1 and T2, have an identity column named C1. The database manager generates values 1, 2, 3,...for the C1 column in table T1, and values 10, 11, 12,...for the C1 column in table T2.

```
CREATE TABLE T1
(C1 SMALLINT GENERATED ALWAYS AS IDENTITY,
 C2 SMALLINT)
```

```
CREATE TABLE T2
(C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY ( START WITH 10 ) ,
 C2 SMALLINT)
```

```
INSERT INTO T1 ( C2 ) VALUES(5)
```

```
INSERT INTO T1 ( C2 ) VALUES(5)
```

```
SELECT * FROM T1
```

C1	C2
1	5
2	5

```
VALUES IDENTITY_VAL_LOCAL() INTO :IVAR
```

At this point, the IDENTITY_VAL_LOCAL function would return a value of 2 in IVAR. The following INSERT statement inserts a single row into T2 where column C2 gets a value of 2 from the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T2 ( C2 ) VALUES( IDENTITY_VAL_LOCAL() )
```

```
SELECT * FROM T2
WHERE C1 = DECIMAL( IDENTITY_VAL_LOCAL(), 15, 0)
```

C1	C2
10	2

Invoking the IDENTITY_VAL_LOCAL function after this INSERT would result in a value of 10, which is the value generated by the database manager for column C1 of T2. Assume another single row is inserted into T2. For the following INSERT statement, the database manager assigns a value of 13 to identity column C1 and gives C2 a value of 10 from IDENTITY_VAL_LOCAL. Thus, C2 is given the last identity value that was inserted into T2.

```
INSERT INTO T2 ( C2, C1 ) VALUES( IDENTITY_VAL_LOCAL(), 13 )
```

```
SELECT * FROM T2
WHERE C1 = DECIMAL( IDENTITY_VAL_LOCAL(), 15, 0)
```

C1	C2
13	10

- The IDENTITY_VAL_LOCAL function can also be invoked in an INSERT statement that both invokes the IDENTITY_VAL_LOCAL function and causes a new value for an identity column to be assigned. The next value to be returned is thus established when the IDENTITY_VAL_LOCAL function is invoked after the INSERT statement completes. For example, consider the following table definition:

IDENTITY_VAL_LOCAL

```
CREATE TABLE T3  
(C1 SMALLINT GENERATED BY DEFAULT AS IDENTITY,  
C2 SMALLINT)
```

For the following INSERT statement, specify a value of 25 for the C2 column, and the database manager generates a value of 1 for C1, the identity column. This establishes 1 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T3 ( C2 ) VALUES( 25 )
```

In the following INSERT statement, the IDENTITY_VAL_LOCAL function is invoked to provide a value for the C2 column. A value of 1 (the identity value assigned to the C1 column of the first row) is assigned to the C2 column, and the database manager generates a value of 2 for C1, the identity column. This establishes 2 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T3 ( C2 ) VALUES( IDENTITY_VAL_LOCAL() )
```

In the following INSERT statement, the IDENTITY_VAL_LOCAL function is again invoked to provide a value for the C2 column, and the user provides a value of 11 for C1, the identity column. A value of 2 (the identity value assigned to the C1 column of the second row) is assigned to the C2 column. The assignment of 11 to C1 establishes 11 as the value that will be returned on the next invocation of the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T3 ( C2, C1 ) VALUES( IDENTITY_VAL_LOCAL(), 11 )
```

After the 3 INSERT statements have been processed, table T3 contains the following:

C1	C2
1	25
2	1
11	2

The contents of T3 illustrate that the expressions in the VALUES clause are evaluated before the assignments for the columns of the INSERT statement. Thus, an invocation of an IDENTITY_VAL_LOCAL function invoked from a VALUES clause of an INSERT statement uses the most recently assigned value for an identity column in a previous INSERT statement.

IFNULL

►►—IFNULL—(—*expression*—,—*expression*—)——————►◄

The IFNULL function returns the value of the first non-null expression.

The IFNULL function is identical to the COALESCE scalar function with two arguments. For more information, see “COALESCE” on page 200.

Example

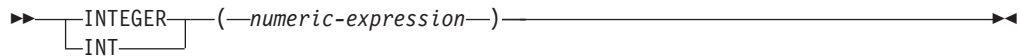
- When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is, null), then return a value of zero.

```
SELECT EMPNO, IFNULL(SALARY,0)
FROM EMPLOYEE
```

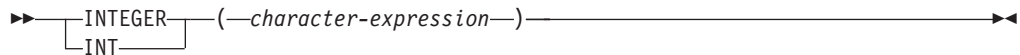
INTEGER

INTEGER or INT

Numeric to Integer



Character to Integer



The `INTEGER` function returns an integer representation of:

- A number
- A character string representation of a decimal number
- A character string representation of an integer
- A character string representation of a floating-point number

Numeric to Integer

numeric-expression

An expression that returns a numeric 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 fractional part of the argument is truncated.

Character to Integer

character-expression

An expression that returns a value that is a character-string representation of an integer. The expression must not be a CLOB.

The result is the same number that would result from `CAST(character-expression AS INTEGER)`. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, integer, or decimal constant. If the whole part of the argument is not within the range of integers, an error occurs. Any fractional part of the argument is truncated.

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.

Note

Syntax alternatives: The `CAST` specification should be used for maximal portability. For more information, see “`CAST` Specification” on page 143.

Example

- Using the `EMPLOYEE` table, select a list containing salary (`SALARY`) divided by education level (`EDLEVEL`). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and the employee number (`EMPNO`).

INTEGER

```
SELECT INTEGER(SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO  
FROM EMPLOYEE
```

JULIAN_DAY

JULIAN_DAY

►►—JULIAN_DAY—(—*expression*—)—————►◄

The JULIAN_DAY function returns an integer value representing a number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date specified in the argument.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a valid character-string representation of a date or timestamp. An argument with a character string data type must not be a CLOB. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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

Examples

- Using sample table EMPLOYEE, set the integer host variable JDAY to the Julian day of the day that Christine Haas (EMPNO = '000010') was employed (HIREDATE = '1965-01-01').

```
SELECT JULIAN_DAY(HIREDATE)
      INTO :JDAY
      FROM EMPLOYEE
      WHERE EMPNO = '000010'
```

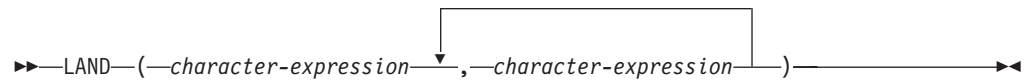
The result is that JDAY is set to 2438762.

- Set integer host variable JDAY to the Julian day for January 1, 1998.

```
SELECT JULIAN_DAY('1998-01-01')
      INTO :JDAY
      FROM SYSIBM.SYSDUMMY1
```

The result is that JDAY is set to 2450815.

LAND



The LAND function returns a string that is the logical 'AND' of the argument strings. This function takes the first argument string, does an AND comparison with the next string, and then continues to do AND comparisons with each successive argument using the previous result. If an argument is encountered that is shorter than the previous result, it is padded with blanks.

The arguments must be character strings but cannot be LOBs. The arguments cannot be mixed data character strings or graphic strings.

The arguments are converted, if necessary, to the attributes of the result. The attributes of the result are determined as follows:

- If all the arguments are fixed-length strings, the result is a fixed-length string of length n , where n is the length of the longest argument.
- If any argument is a varying-length string, the result is a varying-length string with length attribute n , where n is the length attribute of the argument with greatest length attribute. The actual length of the result is m , where m is the actual length of the longest argument.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'A1B1', host variable L2 is a CHARACTER(3) host variable with a value of X'F0F040', and host variable L3 is a CHARACTER(4) host variable with a value of X'A1B10040'.

```
SELECT LAND(:L1,:L2,:L3)
FROM SYSIBM.SYSDUMMY1
```

Returns the value X'A0B00000'.

- Likewise,

```
SELECT LAND(:L3,:L2,:L1)
FROM SYSIBM.SYSDUMMY1
```

Returns the value X'A0B00040'. In this case, the shorter arguments are padded with blanks (X'40'), so the logical AND result differs from the first example.

LCASE

LCASE

►►—LCASE—(*—string-expression—*)—◄◄

The LCASE function returns a string in which all the characters have been converted to lowercase characters, based on the CCSID of the argument.

The LCASE function is identical to the LOWER function. For more information, see “LOWER” on page 261.

LEFT

▶▶—LEFT—(*—string-expression—*, *—integer—*)—▶▶

The LEFT function returns the leftmost *integer* bytes of *string-expression*.

If *string-expression* is a character string, the result is a character string, and each character is one byte. If *string-expression* is a graphic string, the result is a graphic string, and each character is a DBCS or UCS-2 character. If *string-expression* is a binary string, the result is a binary string, and each character is one byte.

string-expression

An expression that specifies the string from which the result is derived. *string-expression* must be a character string, graphic string, or a binary string with a built-in data type.

A substring of *string-expression* is zero or more contiguous characters of *string-expression*. If *string-expression* is a graphic string, a character is a DBCS or UCS-2 character. If *string-expression* is a character string or binary string, a character is a byte.³⁴

integer

An expression that specifies the length of the result. *integer* must be an integer greater than or equal to 0 and less than or equal to *n*, where *n* is the length attribute of *string-expression*.

The *string-expression* is effectively padded on the right with the necessary number of blank characters (or hexadecimal zeroes for binary strings) so that the specified substring of *string-expression* always exists.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *string-expression* and a data type that depends on the data type of *string-expression*:

- VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC
- VARCHAR if *string-expression* is CHAR or VARCHAR
- DBCLOB if *string-expression* is DBCLOB
- CLOB if *string-expression* is CLOB
- BLOB if *string-expression* is BLOB

If *integer* is an integer constant and the argument is not a BLOB, CLOB, or DBCLOB, the result of the function is a fixed-length string.

The actual length of the result is *integer*.

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

The CCSID of the result is the same as that of *string-expression*.

34. The LEFT function accepts mixed data strings. However, because LEFT operates on a strict byte-count basis, the result will not necessarily be a properly formed mixed data string.

LEFT

Example

- Assume the host variable NAME (VARCHAR(50)) has a value of 'KATIE AUSTIN' and the host variable FIRSTNAME_LEN (int) has a value of 5.

```
SELECT LEFT(:NAME, :FIRSTNAME_LEN)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'KATIE'

LENGTH

►—LENGTH—(—*expression*—)—————►

The LENGTH function returns the length of a value. See “CHARACTER_LENGTH” on page 195 for a similar function.

The argument must 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 of strings includes blanks. The length of a varying-length string is the actual length, not the length attribute.

The length of a graphic string is the number of double-byte characters (the number of bytes divided by 2). The length of all other values is the number of bytes used to represent the value:

- 2 for small integer
- 4 for large integer
- 8 for big integer
- The integral part of $(p/2)+1$ for packed decimal numbers with precision p
- p for zoned decimal numbers with precision p
- 4 for single-precision float
- 8 for double-precision float
- The length of the string for strings
- 3 for time
- 4 for date
- 10 for timestamp
- The actual number of bytes used to store the DataLink value (plus 19 if the DataLink is FILE LINK CONTROL and READ PERMISSION DB) for datalinks
- 26 for row ID

Examples

- Assume the host variable ADDRESS is a varying-length character string with a value of ‘895 Don Mills Road’.

```
SELECT LENGTH(:ADDRESS)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 18.

- Assume that PRSTDATE is a column of type DATE.

```
SELECT LENGTH(PRSTDATE)
FROM PROJECT
```

Returns the value 4.

- Assume that PRSTDATE is a column of type DATE.

```
SELECT LENGTH(CHAR(PRSTDATE, EUR))
FROM PROJECT
```

Returns the value 10.

LN

►►—LN—(*numeric-expression*)—◄◄

The LN function returns the natural logarithm of a number. The LN and EXP functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. The value of the argument must be greater than zero.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable NATLOG is a DECIMAL(4,2) host variable with a value of 31.62.

```
SELECT LN(:NATLOG)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 3.45.

LNOT

►►—LNOT—(*—character-expression—*)—►►

The LNOT function returns a string that is the logical NOT of the argument string.

The argument must be a character string but cannot be a LOB. The argument cannot be a MIXED character string or a graphic string.

The data type and length attribute of the result is the same as the data type and length attribute of the argument value. If the argument is a varying-length string, the actual length of the result is the same as the actual length of the argument value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'F0F0'.

```
SELECT LNOT(:L1)
FROM SYSIBM.SYSDUMMY1
```

Returns the value X'0F0F'.

LOCATE

LOCATE

►►—LOCATE—(—*search-string*—,—*source-string*—,—*start*—)—►►

The LOCATE function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*. If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin.

search-string

An expression that specifies the string that is to be searched for. *Search-string* may be a character-string, a graphic-string, or a binary-string expression. It must be compatible with the *source-string*.

source-string

An expression that specifies the source string in which the search is to take place. *Source-string* may be a character-string, a graphic-string, or a binary-string expression.

start

An expression that specifies the position within *source-string* at which the search is to start. It must be an integer that is greater than or equal to zero.

If *start* is specified, the function is similar to:

```
POSSTR( SUBSTR(source-string,start) , search-string )
```

If *start* is not specified, the function is equivalent to:

```
POSSTR( source-string , search-string )
```

For more information, see “POSITION or POSSTR” on page 277.

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

If the CCSID of the *search-string* is different than the CCSID of the *source-string*, it is converted to the CCSID of the *source-string*.

Example

- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
FROM IN_TRAY
WHERE LOCATE('GOOD', NOTE_TEXT) <> 0
```

LOG10

►►—LOG10—(*—numeric-expression—*)—◄◄

The LOG10 function returns the common logarithm (base 10) of a number. The LOG10 and ANTILOG functions are inverse operations.

The argument value can be of any built-in numeric data type.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: LOG is a synonym for LOG10. It is supported only for compatibility with previous DB2 releases. LOG10 should be used instead of LOG because some database managers and applications implement LOG as the natural logarithm of a number instead of the common logarithm of a number.

Example

- Assume the host variable L is a DECIMAL(4,2) host variable with a value of 31.62.

```
SELECT LOG10(:L)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 1.49.

LOR

LOR

►► LOR(-character-expression, -character-expression) ◄◄

The LOR function returns a string that is the logical OR of the argument strings. This function takes the first argument string, does an OR comparison with the next string, and then continues to do OR comparisons for each successive argument using the previous result. If an argument is encountered that is shorter than the previous result, it is padded with blanks.

The arguments must be character strings but cannot be LOBs. The arguments cannot be mixed data character strings or graphic strings.

The arguments are converted, if necessary, to the attributes of the result. The attributes of the result are determined as follows:

- If all the arguments are fixed-length strings, the result is a fixed-length string of length n , where n is the length of the longest argument.
- If any argument is a varying-length string, the result is a varying-length string with length attribute n , where n is the length attribute of the argument with greatest length attribute. The actual length of the result is m , where m is the actual length of the longest argument.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'0101', host variable L2 is a CHARACTER(3) host variable with a value of X'F0F000', and host variable L3 is a CHARACTER(4) host variable with a value of X'0000000F'.

```
SELECT LOR(:L1,:L2,:L3)
FROM SYSIBM.SYSDUMMY1
```

Returns the value X'F1F1000F'.

- Likewise,

```
SELECT LOR(:L3,:L2,:L1)
FROM SYSIBM.SYSDUMMY1
```

Returns the value X'F1F1404F'. In this case, the shorter arguments are padded with blanks (X'40'), so the logical OR result differs from the first example.

LOWER

►►—LOWER—(—*string-expression*—)——►►

The LOWER function returns a string in which all the characters have been converted to lowercase characters, based on the CCSID of the argument. Only SBCS and UCS-2 graphic characters are converted. The characters A-Z are converted to a-z, and characters with diacritical marks are converted to their lowercase equivalent, if any. Refer to the UCS-2 level 1 mapping tables section of the Globalization topic in the iSeries Information Center for a description of the monocasing tables that are used for this translation.

string-expression

An expression that specifies the string to be converted. String-expression must be a character or UCS-2 graphic string.

The result of the function has the same data type, length attribute, actual length, and CCSID as the argument. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Note

Syntax alternatives: LCASE is a synonym for LOWER.

Examples

- Ensure that the characters in the value of host variable NAME are lowercase. NAME has a data type of VARCHAR(30) and a value of 'Christine Smith'.

```
SELECT LOWER(:NAME)
FROM SYSIBM.SYSDUMMY1
```

The result is the value 'christine smith'.

LTRIM

►► LTRIM(*—string-expression—*) ◀◀

The LTRIM function removes blanks or hexadecimal zeros from the beginning of a string expression.³⁵

The argument must be a string expression.

- If the argument is a binary string, then the leading hexadecimal zeros (X'00') are removed.
- If the argument is a DBCS graphic string, then the leading DBCS blanks are removed.
- If the first argument is a UCS-2 graphic string, then the leading UCS-2 blanks are removed
- Otherwise, leading SBCS blanks are removed.

The data type of the result depends on the data type of *expression*:

Data type of <i>expression</i>	Data type of the Result
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC
BLOB	BLOB
CLOB	CLOB
DBCLOB	DBCLOB

The length attribute of the result is the same as the length attribute of *string-expression*. The actual length of the result is the length of *string-expression* minus the number of bytes removed. If all characters are removed, the result is an empty string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The CCSID of the result is the same as that of the string.

Example

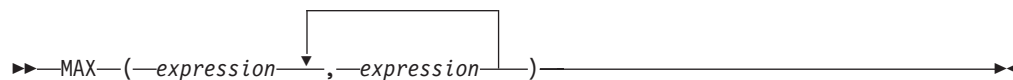
- Assume the host variable HELLO of type CHAR(9) has a value of ' Hello'.

```
SELECT LTRIM(:HELLO)
FROM SYSIBM.SYSDUMMY1
```

Results in: 'Hello'.

³⁵ The LTRIM function returns the same results as: STRIP(expression,LEADING)

MAX



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

The arguments must be compatible. Character-string arguments are compatible with datetime values, but are not compatible with graphic strings. The arguments cannot be DataLink values.

The result of the function is the largest argument value. The result can be null if at least one argument can be null; the result is the null value if one of the arguments is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands as explained in “Rules for Result Data Types” on page 93.

If a sort sequence other than *HEX is in effect when the statement is executed and SBCS, UCS-2, or mixed data is involved, the weighted values of the strings are compared instead of the actual values. The weighted values are based on the sort sequence.

Examples

- Assume the host variable M1 is a DECIMAL(2,1) host variable with a value of 5.5, host variable M2 is a DECIMAL(3,1) host variable with a value of 4.5, and host variable M3 is a DECIMAL(3,2) host variable with a value of 6.25.

```
SELECT MAX(:M1, :M2, :M3)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 6.25.

- Assume the host variable M1 is a CHARACTER(2) host variable with a value of 'AA', host variable M2 is a CHARACTER(3) host variable with a value of 'AA ', and host variable M3 is a CHARACTER(4) host variable with a value of 'AA A'.

```
SELECT MAX(:M1, :M2, :M3)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'AA A'.

|
|

MICROSECOND

MICROSECOND

►► MICROSECOND (—*expression*—) ◀◀

The MICROSECOND function returns the microsecond part of a value.

The argument must be an expression that returns a value of one of the following built-in data types: a timestamp, a character string, or a numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a timestamp. For the valid formats of string representations of timestamps, see “String Representations of Datetime Values” on page 68.
- If *expression* is a number, it must be a timestamp duration. For the valid formats of datetime durations, see “Datetime Operands and Durations” on page 134.

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 character-string representation of a timestamp:
The result is the microsecond part of the value, which is an integer between 0 and 999999.
- If the argument is a duration:
The result is the microsecond part of the value, which is an integer between -999999 and 999999. A nonzero result has the same sign as the argument.

Example

- Assume 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

►—MIDNIGHT_SECONDS—(*expression*)—◄

The MIDNIGHT_SECONDS function returns an integer value that is greater than or equal to 0 and less than or equal to 86 400 representing the number of seconds between midnight and the time value specified in the argument.

The argument must be an expression that returns a value of one of the following built-in data types: time, a timestamp, or a valid character-string representation of a time or timestamp. An argument with a character-string data type must not be a CLOB. For the valid formats of string representations of timestamps, see “String Representations of Datetime Values” on page 68.

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

Examples

- Find the number of seconds between midnight and 00:01:00, and midnight and 13:10:10. Assume that host variable XTIME1 has a value of '00:01:00', and that XTIME2 has a value of '13:10:10'.

```
SELECT MIDNIGHT_SECONDS(:XTIME1), MIDNIGHT_SECONDS(:XTIME2)
FROM SYSIBM.SYSDUMMY1
```

This example returns 60 and 47410. Because there are 60 seconds in a minute and 3600 seconds in an hour, 00:01:00 is 60 seconds after midnight $((60 * 1) + 0)$, and 13:10:10 is 47410 seconds $((3600 * 13) + (60 * 10) + 10)$.

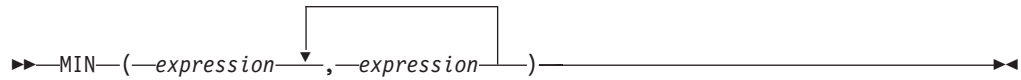
- Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
SELECT MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00')
FROM SYSIBM.SYSDUMMY1
```

This example returns 86400 and 0. Although these two values represent the same point in time, different values are returned.

MIN

MIN



The MIN scalar function returns the minimum value in a set of values.

The arguments must be compatible. Character-string arguments are compatible with datetime values, but are not compatible with graphic strings. The arguments cannot be DataLink values.

The result of the function is the smallest argument value. The result can be null if at least one argument can be null; the result is the null value if one of the arguments is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands as explained in “Rules for Result Data Types” on page 93.

If a sort sequence other than *HEX is in effect when the statement is executed and SBCS, UCS-2, or mixed data is involved, the weighted values of the strings are compared instead of the actual values. The weighted values are based on the sort sequence.

Examples

- Assume the host variable M1 is a DECIMAL(2,1) host variable with a value of 5.5, host variable M2 is a DECIMAL(3,1) host variable with a value of 4.5, and host variable M3 is a DECIMAL(3,2) host variable with a value of 6.25.

```
SELECT MIN(:M1, :M2, :M3)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 4.50.

- Assume the host variable M1 is a CHARACTER(2) host variable with a value of 'AA', host variable M2 is a CHARACTER(3) host variable with a value of 'AAA', and host variable M3 is a CHARACTER(4) host variable with a value of 'AAAA'.

```
SELECT MIN(:M1, :M2, :M3)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'AA'.

MINUTE

►►—MINUTE—(—*expression*—)—————►►

The MINUTE function returns the minute part of a value.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string or a numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see “String Representations of Datetime Values” on page 68.
- If *expression* is a number, it must be a time duration or timestamp duration. For the valid formats of datetime durations, see “Datetime Operands and Durations” on page 134.

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, a timestamp, or a valid character-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—(—*numeric-expression-1*—,—*numeric-expression-2*—)————►►

The MOD function divides the first argument by the second argument and returns the remainder.

The formula used to calculate the remainder is:

$$\text{MOD}(x,y) = x - (x/y) * y$$

where x/y is the truncated integer result of the division. The result is negative only if first argument is negative.

The arguments must each be an expression that returns a built-in numeric data type. *numeric-expression-2* cannot be zero.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The attributes of the result are determined as follows:

- If both arguments are large or small integers with zero scale, the data type of the result is large integer.
- If both arguments are integers with zero scale and at least one of the arguments is a big integer, the data type of the result is big integer.
- If one argument is an integer with zero scale and the other is decimal, the result is decimal with the same precision and scale as the decimal argument.
- If both arguments are decimal or integer with scale numbers, the result is decimal. The precision of the result is $\min(p-s, p'-s') + \max(s, s')$, and the scale of the result is $\max(s, s')$, where the symbols p and s denote the precision and scale of the first operand, and the symbols p' and s' denote the precision and scale of the second operand.
- If either argument is floating point, the data type of the result is double-precision floating point.

The operation is performed in floating point; the operands having been first converted to double-precision floating-point numbers, if necessary.

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

Examples

- Assume the host variable M1 is an integer host variable with a value of 5, and host variable M2 is an integer host variable with a value of 2.

```
SELECT MOD(:M1, :M2)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 1.

- Assume the host variable M1 is an integer host variable with a value of 5, and host variable M2 is a DECIMAL(3,2) host variable with a value of 2.20.


```
SELECT MOD(:M1, :M2)  
FROM SYSIBM.SYSDUMMY1
```

Returns the value 0.60.

- Assume the host variable M1 is a DECIMAL(4,2) host variable with a value of 5.50, and host variable M2 is a DECIMAL(4,1) host variable with a value of 2.0.

```
SELECT MOD(:M1, :M2)  
FROM SYSIBM.SYSDUMMY1
```

Returns the value 1.50.

MONTH

►► MONTH (—*expression*—) ◀◀

The MONTH function returns the month part of a value.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.
- If *expression* is a number, it must be a date duration or timestamp duration. For the valid formats of datetime durations, see “Datetime Operands and Durations” on page 134.

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, a timestamp, or a valid character-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
```

NODENAME

►►—NODENAME—(—*table-designator*—)—————►►

The NODENAME function returns the relational database name of where a row is located. If the argument identifies a non-distributed table, the value of the CURRENT SERVER special register is returned. For more information about nodes, see the DB2 Multisystem book.

The argument must be a table designator of the subselect. For more information about table designators, see “Table Designators” on page 110.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or derived table, the function returns the relational database name of its base table. If the argument identifies a view, common table expression, or derived table derived from more than one base table, the function returns the relational database name of the first table in the outer subselect of the view, common table expression, or derived table.

The argument must not identify a view, common table expression, or derived table whose outer subselect includes a column function, a GROUP BY clause, a HAVING clause, a UNION clause, or DISTINCT clause. If the subselect contains a GROUP BY or HAVING clause, the NODENAME function can only be specified in the WHERE clause or as an operand of a column function. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is VARCHAR(18). The result can be null.

The CCSID of the result is the default CCSID of the current server.

Example

- Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO) and determine the node from which each row involved in the join originated.

```
SELECT EMPNO, NODENAME(X), NODENAME(Y)
FROM EMPLOYEE X, DEPARTMENT Y
WHERE X.DEPTNO=Y.DEPTNO
```

NODENUMBER

NODENUMBER

►►—NODENUMBER—(—*table-designator*—)—◄◄

The NODENUMBER function returns the node number of a row. If the argument identifies a non-distributed table, the value 0 is returned.³⁶ For more information about nodes and node numbers, see the DB2 Multisystem book.

The argument must be a table designator of the subselect. For more information about table designators, see “Table Designators” on page 110.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or derived table, the function returns the node number of its base table. If the argument identifies a view, common table expression, or derived table derived from more than one base table, the function returns the node number of the first table in the outer subselect of the view, common table expression, or derived table.

The argument must not identify a view, common table expression, or derived table whose outer subselect includes a column function, a GROUP BY clause, a HAVING clause, a UNION clause, or DISTINCT clause. If the subselect contains a GROUP BY or HAVING clause, the NODENUMBER function can only be specified in the WHERE clause or as an operand of a column function. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a large integer. The result can be null.

Example

- Determine the node number and employee name for each row in the EMPLOYEE table. If this is a distributed table, the number of the node where the row exists is returned.

```
SELECT NODENUMBER(EMPLOYEE), LASTNAME  
FROM EMPLOYEE
```

³⁶ If the argument identifies a DDS created logical file that is based on more than one physical file member, NODENUMBER will not return 0, but instead will return the underlying physical file member number.

NOW



The NOW function returns a timestamp based on a reading of the time-of-day clock when the SQL statement is executed at the current server. The value returned by the NOW function is the same as the value returned by the CURRENT TIMESTAMP special register. If this function is used more than once within a single SQL statement, or used with the CURDATE or CURTIME scalar functions or the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers within a single statement, all values are based on a single clock reading.

The data type of the result is a timestamp. The result cannot be null.

Example

- Return the current timestamp based on the time-of-day clock.

```
SELECT NOW()  
FROM SYSIBM.SYSDUMMY1
```

NULLIF

►►—NULLIF—(—*expression*—,—*expression*—)———►►

The NULLIF function returns a null value if the arguments compare equal, otherwise it returns the value of the first argument.

The arguments must be compatible and comparable data types. Character-string arguments are compatible with datetime values. If one operand is a distinct type, the other operand must be the same distinct type. The arguments cannot be DataLink values.

The attributes of the result are the attributes of the first argument. The result can be null. The result is null if the first argument is null or if both arguments are equal.

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 operand, e1.

Example

- Assume host variables PROFIT, CASH, and LOSSES have DECIMAL data types with the values 4500.00, 500.00, and 5000.00 respectively:

```
SELECT NULLIF (:PROFIT + :CASH, :LOSSES )  
FROM SYSIBM.SYSDUMMY1
```

Returns the null value.

PARTITION

►►PARTITION—(*—table-designator—*)—►►

The PARTITION function returns the partition number of a row obtained by applying the hashing function on the partitioning key value of the row. Also see the HASH function. If the argument identifies a non-distributed table, the value 0 is returned. For more information about partition numbers and partitioning keys, see the DB2 Multisystem book.

The argument must be a table designator of the subselect. For more information about table designators, see “Table Designators” on page 110.

In SQL naming, the table name may be qualified. In system naming, the table name cannot be qualified.

If the argument identifies a view, common table expression, or derived table, the function returns the partition number of its base table. If the argument identifies a view, common table expression, or derived table derived from more than one base table, the function returns the partition number of the first table in the outer subselect of the view, common table expression, or derived table.

The argument must not identify a view, common table expression, or derived table whose outer subselect includes a column function, a GROUP BY clause, a HAVING clause, a UNION clause, or DISTINCT clause. If the subselect contains a GROUP BY or HAVING clause, the PARTITION function can only be specified in the WHERE clause or as an operand of a column function. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a large integer with a value between 0 and 1023. The result can be null.

Example

- Select the employee number (EMPNO) from the EMPLOYEE table for all rows where the partition number is equal to 100.

```
SELECT EMPNO
FROM EMPLOYEE
WHERE PARTITION(EMPLOYEE) = 100
```

PI

PI

►►—PI—(—)—►►

Returns the value of PI 3.141592653589793. There are no arguments.

The result of the function is double-precision floating-point. The result cannot be null.

Example

- The following returns the circumference of a circle with diameter 10:

```
SELECT PI()*10  
FROM SYSIBM.SYSDUMMY1
```


POSITION or POSSTR

- Otherwise, the result returned by the function is 0.³⁷

Example

- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, POSSTR(NOTE_TEXT, 'GOOD')
FROM IN_TRAY
WHERE POSSTR(NOTE_TEXT, 'GOOD') <> 0
```

³⁷. This includes the case where the *search-string* is longer than the *source-string*.

POWER

►►—POWER—(*—numeric-expression-1—*,*—numeric-expression-2—*)——————►►

The POWER function returns the result of raising the first argument to the power of the second argument.³⁸

Each argument must be an expression that returns the value of any built-in numeric data type. If the value of *numeric-expression-1* is equal to zero, then *numeric-expression-2* must be greater than or equal to zero. If both arguments are 0, the result is 1. If the value of *numeric-expression-1* is less than zero, then *numeric-expression-2* must be an integer value.

The result of the function is a double-precision floating-point number. If an argument can be null, the result can be null; if an argument is null, the result is the null value.

Example

- Assume the host variable HPOWER is an integer with value 3.

```
SELECT POWER(2, :HPOWER)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 8.

38. The result of the POWER function is exactly the same as the result of exponentiation: *numeric-expression-1* ** *numeric-expression-2*.

QUARTER

QUARTER

►►—QUARTER—(—*expression*—)—————►►

The `QUARTER` function returns an integer between 1 and 4 that represents the quarter of the year in which the date resides. For example, any dates in January, February, or March will return the integer 1.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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

Example

- Using the `PROJECT` table, set the host variable `QUART` (`INTEGER`) to the quarter in which project 'PL2100' ended (`PRENDATE`).

```
SELECT QUARTER(PRENDATE)
      INTO :QUART
      FROM PROJECT
      WHERE PROJNO = 'PL2100'
```

Results in `QUART` being set to 3.

RADIANS

►►—RADIANS—(*—numeric-expression—*)—◄◄

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

The argument must be an expression that returns the value of any built-in numeric data type. If the argument is not a double precision floating-point number, it is converted to one for processing by the function.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

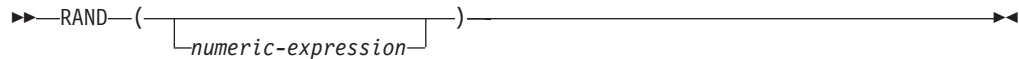
Example

- Assume that host variable HDEG is an INTEGER with a value of 180. The following statement:

```
SELECT RADIANS(:HDEG)
FROM SYSIBM.SYSDUMMY1
```

Returns a double precision floating-point number with an approximate value of 3.1415926536.

RAND



The RAND function returns a floating point value between 0 and 1.

If an expression is specified, it is used as the seed value. The expression must be a SMALLINT or INTEGER.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume that host variable HRAND is an INTEGER with a value of 100. The following statement:

```
SELECT RAND(:HRAND)
FROM SYSIBM.SYSDUMMY1
```

Returns a random floating-point number between 0 and 1, such as the approximate value .0121398.

- To generate values in a numeric interval other than 0 to 1, multiply the RAND function by the size of the desired interval. For example, to get a random number between 0 and 10, such as the approximate value 5.8731398, multiply the function by 10:

```
SELECT RAND(:HRAND) * 10
FROM SYSIBM.SYSDUMMY1
```

REAL

►► REAL (*numeric-expression* | *character-expression*) ►►

The REAL function returns a single-precision floating-point representation of:

- A number
- A character string representation of a decimal number
- A character string representation of an integer
- A character string representation of a floating-point number

Numeric to Real

numeric-expression

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

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable. If the numeric value of the argument is not within the range of single-precision floating-point, an error occurs.

Character to Real

character-expression

An expression that returns a character string value.

The result is the same number that would result from CAST(*character-expression* AS REAL). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an floating-point, integer, or decimal constant. If the numeric value of the argument is not within the range of single-precision floating-point, an error occurs.

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.

Note

Syntax alternatives: The CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

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. To eliminate the possibility of out-of-range results, REAL is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, REAL(SALARY)/COMM
FROM EMPLOYEE
WHERE COMM > 0
```

ROUND

ROUND

►►—ROUND—(—*numeric-expression-1*—,—*numeric-expression-2*—)—►►

The ROUND function returns *numeric-expression-1* rounded to some number of places to the right or left of the decimal point.

numeric-expression-1

An expression that returns a value of any built-in numeric data type.

numeric-expression-2

An expression that returns a small or large integer. The absolute value of integer specifies the number of places to the right of the decimal point for the result if *numeric-expression-2* is not negative, or to left of the decimal point if *numeric-expression-2* is negative.

If *numeric-expression-2* is not negative, *numeric-expression-1* is rounded to the *numeric-expression-2* number of places to the right of the decimal point. A digit value of 5 is rounded to the next higher positive number.

If *numeric-expression-2* is negative, *numeric-expression-1* is rounded to the absolute value of (*numeric-expression-2*+1) number of places to the left of the decimal point. A digit value of 5 is rounded to the next lower negative number. If the absolute value of *numeric-expression-2* is greater than the number of digits to the left of the decimal point, the result is 0.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument, except that precision is increased by one if *numeric-expression-1* is DECIMAL or NUMERIC and the precision is less than 31. For example, an argument with a data type of DECIMAL(5,2) will result in DECIMAL(6,2). An argument with a data type of DECIMAL(31,2) will result in DECIMAL(31,2).

If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

- Calculate the number 873.726 rounded to 2, 1, 0, -1, -2, -3, and -4 decimal places respectively.

```
SELECT ROUND(873.726, 2),  
       ROUND(873.726, 1),  
       ROUND(873.726, 0),  
       ROUND(873.726, -1),  
       ROUND(873.726, -2),  
       ROUND(873.726, -3),  
       ROUND(873.726, -4)  
FROM SYSIBM.SYSDUMMY1
```

Returns the following values, respectively:

```
0873.730  0873.700  0874.000  0870.000  0900.000  1000.000  0000.000
```

- Calculate both positive and negative numbers.

```
SELECT ROUND( 3.5, 0),  
       ROUND( 3.1, 0),  
       ROUND(-3.1, 0),  
       ROUND(-3.5, 0)  
FROM SYSIBM.SYSDUMMY1
```


Returns the following examples, respectively:

04.0 03.0 -03.0 -04.0

respectively.

ROWID

►►—ROWID—(—*string-expression*—)—————►

The ROWID function casts a character string to a row ID.

string-expression

An expression that returns a character string or binary string value. The *string-expression* must not be a CLOB. Although the string can contain any value, it is recommended that it contain a ROWID value that was previously generated by DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries to ensure a valid ROWID value is returned. For example, the function can be used to convert a ROWID that value that was cast to CHAR value back to a ROWID value.

If the actual length of *string-expression* is less than 40, the result is not padded. If the actual length of *string-expression* is greater than 40, the result is truncated. If non-blank characters are truncated, a warning is returned.

The length attribute of the result is 40. The actual length of the result is the length of *string-expression*.

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

Note

Syntax alternatives: The CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Example

- Assume that table EMPLOYEE contains a ROWID column EMP_ROWID. Also assume that the table contains a row that is identified by a row ID value that is equivalent to X'F0DFD230E3C0D80D81C201AA0A28010000000000203'. Using direct row access, select the employee number for that row.

```
SELECT EMPNO
FROM EMPLOYEE
WHERE EMP_ROWID = ROWID(X'F0DFD230E3C0D80D81C201AA0A28010000000000203')
```

RRN

►►—RRN—(—*table-designator*—)———►►

The RRN function returns the relative record number of a row.

The argument must be a table designator of the subselect. For more information about table designators, see “Table Designators” on page 110.

In SQL naming, the table name may be qualified. In system naming, the table name can not be qualified.

If the argument identifies a view, common table expression, or derived table, the function returns the relative record number of its base table. If the argument identifies a view, common table expression, or derived table derived from more than one base table, the function returns the relative record number of the first table in the outer subselect of the view, common table expression, or derived table.

If the argument identifies a distributed table, the function returns the relative record number of the row on the node where the row is located. This means that RRN will not be unique for each row of a distributed table.

The argument must not identify a view, common table expression, or derived table whose outer subselect includes a column function, a GROUP BY clause, a HAVING clause, a UNION clause, or DISTINCT clause. The RRN function cannot be specified in a SELECT clause if the subselect contains a column function, a GROUP BY clause, or a HAVING clause. If the argument is a correlation name, the correlation name must not identify a correlated reference.

The data type of the result is a decimal with precision 15 and scale 0. The result can be null.

Example

- Return the relative record number and employee name from table EMPLOYEE for those employees in department 20.

```
SELECT RRN(EMPLOYEE), LASTNAME
FROM EMPLOYEE
WHERE DEPTNO = 20
```

RTRIM

►► RTRIM(*—string-expression—*) ◀◀

The RTRIM function removes blanks or hexadecimal zeroes from the end of a string expression.³⁹

The argument must be a string expression.

- If the argument is a binary string, then the trailing hexadecimal zeros (X'00') are removed.
- If the argument is a DBCS graphic string, then the trailing DBCS blanks are removed.
- If the first argument is a UCS-2 graphic string, then the trailing UCS-2 blanks are removed.
- Otherwise, trailing SBCS blanks are removed.

The data type of the result depends on the data type of *string-expression*:

Data type of <i>expression</i>	Data type of the Result
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC
BLOB	BLOB
CLOB	CLOB
DBCLOB	DBCLOB

The length attribute of the result is the same as the length attribute of *string-expression*. The actual length of the result is the length of the expression minus the number of bytes removed. If all characters are removed, the result is an empty string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The CCSID of the result is the same as that of the string.

Example

- Assume the host variable HELLO of type CHAR(9) has a value of 'Hello '.

```
SELECT RTRIM(:HELLO)
FROM SYSIBM.SYSDUMMY1
```

Results in: 'Hello'.

³⁹ The RTRIM function returns the same results as: STRIP(expression,TRAILING)

SECOND

►►—SECOND—(—*expression*—)—————►►

The SECOND function returns the seconds part of a value.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, a character string, or a numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a time or timestamp. For the valid formats of string representations of times and timestamps, see “String Representations of Datetime Values” on page 68.
- If *expression* is a number, it must be a time duration or timestamp duration. For the valid formats of datetime durations, see “Datetime Operands and Durations” on page 134.

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, a timestamp, or a valid character-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.

```
SELECT SECOND(:TIME_DUR)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 45.

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

```
SELECT SECOND(RECEIVED)
FROM IN_TRAY
```

Returns the value 30.

SIGN

►►—SIGN—(*numeric-expression*)—◀◀

The SIGN function returns an indicator of the sign of expression. The returned value is:

- 1 if the argument is less than zero
- 0 if the argument is zero
- 1 if the argument is greater than zero

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

The result has the same data type and length attribute as the argument, except that precision is increased by one if the argument is DECIMAL or NUMERIC and the scale of the argument is equal to its precision. For example, an argument with a data type of DECIMAL(5,5) will result in DECIMAL(6,5). If the precision is already 31, the scale will be decreased by one. For example, DECIMAL(31,31) will result in DECIMAL(31,30).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume that host variable PROFIT is a large integer with a value of 50000.

```
SELECT SIGN(:PROFIT)
FROM EMPLOYEE
```

Returns the value 1.

SIN

►►—SIN—(*numeric-expression*)—◄◄

The SIN function returns the sine of the argument, where the argument is an angle expressed in radians. The SIN and ASIN functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable SINE is a decimal (2,1) host variable with a value of 1.5.

```
SELECT SIN(:SINE)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 0.99.

SINH

►►—SINH—(*—numeric-expression—*)——————►◄

The SINH function returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

The argument must be an expression that returns the value of any built-in numeric data type.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable HSINE is a decimal (2,1) host variable with a value of 1.5.

```
SELECT SINH(:HSINE)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 2.12.

SMALLINT

Numeric to Smallint

►►—SMALLINT—(*—numeric-expression—*)—►►

Character to Smallint

►►—SMALLINT—(*—character-expression—*)—►►

The SMALLINT function returns a small integer representation of

- A number
- A character string representation of a decimal number
- A character string representation of an integer
- A character string representation of a floating-point number

Numeric to Smallint

numeric-expression

An expression that returns a numeric value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. The fractional part of the argument is truncated.

Character to Smallint

character-expression

An expression that returns a value that is a character-string representation of an integer. The expression must not be a CLOB.

The result is the same number that would result from CAST(*character-expression* AS SMALLINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a floating-point, integer, or decimal constant. If the whole part of the argument is not within the range of small integers, an error occurs. Any fractional part of the argument is truncated. The fractional part of the argument is truncated.

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.

Note

Syntax alternatives: The CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Example

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

```
SELECT SMALLINT(SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
FROM EMPLOYEE
```

SOUNDEX

SOUNDEX

►►—SOUNDEX—(—*string-expression*—)—————►►

The SOUNDEX function 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 any built-in string data type other than a BLOB, CLOB, or DBCLOB.

The data type of the result is CHAR(4). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID of the result is the default CCSID of the current server.

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. For more information, see “DIFFERENCE” on page 221.

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')
```

Returns the row:

```
000110 LUCCHESI
```

SPACE

►►—SPACE—(*—numeric-expression—*)——————►►

The SPACE function returns a character string that consists of the number of SBCS blanks that the argument specifies.

The argument must be an expression that results in an integer. The integer specifies the number of SBCS blanks for the result, and it must be between 0 and 32740. If *numeric-expression* is a constant, it must not be the constant 0.

The result of the function is a varying-length character string (VARCHAR) that contains SBCS data.

If *numeric-expression* is a constant, the length attribute of the result is the constant. Otherwise, the length attribute of the result is 4000. The actual length of the result is the value of *numeric-expression*. The actual length of the result must not be greater than the length attribute of the result.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The CCSID is the default CCSID for SBCS data of the job.

Example

- The following statement returns a character string that consists of 5 blanks.

```
SELECT SPACE(5)
FROM SYSIBM.SYSDUMMY1
```

SQRT

►►—SQRT—(*numeric-expression*)—◄◄

The SQRT function returns the square root of a number.

The argument must be an expression that returns a value of any built-in numeric data type. The value of *numeric-expression* must be greater than or equal to zero. The argument is converted to double-precision floating point for processing by the function.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

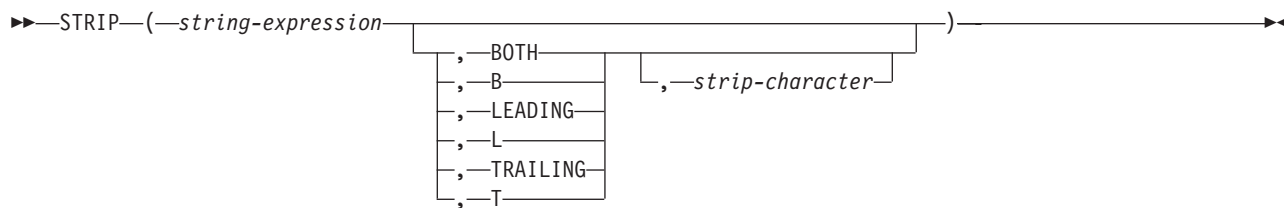
Example

- Assume the host variable SQUARE is a DECIMAL(2,1) host variable with a value of 9.0.

```
SELECT SQRT(:SQUARE)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 3.00.

STRIP

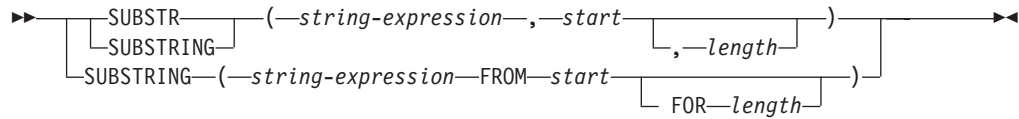


The STRIP function removes blanks or another specified character from the end and/or beginning of a string expression.

The STRIP function is identical to the TRIM scalar function. For more information, see “TRIM” on page 309.

SUBSTRING or SUBSTR

SUBSTRING or SUBSTR



The SUBSTR and SUBSTRING functions return a substring of a string.

string-expression

An expression that specifies the string from which the result is derived.

String-expression must be a character, graphic, or binary string. If *string-expression* is a character string, the result of the function is a character string. If it is a graphic string, the result of the function is a graphic string. If it is a binary string, the result of the function is a binary string.

A substring of *string-expression* is zero or more contiguous characters of *string-expression*. If *string-expression* is a graphic string, a character is a DBCS or UCS-2 character. If *string-expression* is a character string or binary string, a character is a byte. The SUBSTR function accepts mixed data strings. However, because SUBSTR operates on a strict byte-count basis, the result will not necessarily be a properly formed mixed data string.

start

An expression that specifies the position within *string-expression* of the first character (or byte) of the result. It must be a binary integer. *start* may be negative or zero. It may also be greater than the length attribute of *string-expression*. (The length attribute of a varying-length string is its maximum length.)

length

An expression that specifies the length of the result. If specified, *length* must be an integer that is greater than or equal to 0 and less than or equal to n , where n is the length attribute of *string-expression* - *start* + 1. It must not, however, be the integer constant 0.

If SUBSTR is specified and *length* is explicitly specified, *string-expression* is effectively padded on the right with the necessary number of blank characters so that the specified substring of *string-expression* always exists. Hexadecimal zeroes are used as the padding character when *string-expression* is a binary string.

If SUBSTRING is specified and *length* is explicitly specified, padding is not performed.

If *string-expression* is a fixed-length string, omission of *length* is an implicit specification of $\text{LENGTH}(\text{string-expression}) - \text{start} + 1$, which is the number of characters (or bytes) from the *start* character (or byte) to the last character (or byte) of *string-expression*. If *string-expression* is a varying-length string, omission of *length* is an implicit specification of zero or $\text{LENGTH}(\text{string-expression}) - \text{start} + 1$, whichever is greater. If the resulting length is zero, the result is the empty string.

The data type of the result depends on the data type of *string-expression* and whether the function is a SUBSTR or SUBSTRING:

SUBSTRING or SUBSTR

Data type of <i>string-expression</i>	Data Type of the Result for SUBSTRING	Data Type of the Result for SUBSTR
CHAR or VARCHAR	VARCHAR	CHAR, if: <ul style="list-style-type: none"> • <i>length</i> is explicitly specified by an integer constant. • <i>length</i> is not explicitly specified, but <i>string-expression</i> is a fixed-length string and <i>start</i> is an integer constant. VARCHAR, in all other cases
CLOB	CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC	GRAPHIC, if: <ul style="list-style-type: none"> • <i>length</i> is explicitly specified by an integer constant. • <i>length</i> is not explicitly specified, but <i>string-expression</i> is a fixed-length string and <i>start</i> is an integer constant. VARGRAPHIC, in all other cases.
DBCLOB	DBCLOB	DBCLOB
BLOB	BLOB	BLOB

If the SUBSTRING function is specified, the length attribute of the result is equal to the length attribute of *string-expression*.

If the SUBSTR function is specified and *string-expression* is not a LOB, the length attribute of the result depends on *length*, *start*, and the attributes of *string-expression*.

- If *length* is explicitly specified by an integer constant, the length attribute of the result is *length*.
- If *length* is not explicitly specified, but *string-expression* is a fixed-length string and *start* is an integer constant, the length attribute of the result is $\text{LENGTH}(\textit{string-expression}) - \textit{start} + 1$.

In all other cases, the length attribute of the result is the same as the length attribute of *string-expression*. (Remember that if the actual length of *string-expression* is less than the value for *start*, the actual length of the substring is zero.)

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.

The CCSID of the result is the same as that of *string-expression*.

Examples

- Assume the host variable NAME (VARCHAR(50)) has a value of 'KATIE AUSTIN' and the host variable SURNAME_POS (INTEGER) has a value of 7.

```
SELECT SUBSTR(:NAME, :SURNAME_POS)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'AUSTIN'.

- Likewise,

```
SELECT SUBSTR(:NAME, :SURNAME_POS, 1)
FROM SYSIBM.SYSDUMMY1
```

SUBSTRING or SUBSTR

Returns the value 'A'.

- 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'.

TAN

►►—TAN—(*numeric-expression*)—◄◄

The TAN function returns the tangent of the argument, where the argument is an angle expressed in radians. The TAN and ATAN functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable TANGENT is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT TAN(:TANGENT)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 14.10.

TANH

►►—TANH—(*numeric-expression*)—◄◄

The TANH function returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians. The TANH and ATANH functions are inverse operations.

The argument must be an expression that returns the value of any built-in numeric data type.

The data type of the result is double-precision floating point. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

- Assume the host variable HTANGENT is a DECIMAL(2,1) host variable with a value of 1.5.

```
SELECT TANH (:HTANGENT)
FROM SYSIBM.SYSDUMMY1
```

Returns the approximate value 0.90.

TIME

►►—TIME—(*expression*)—◄◄

The TIME function returns a time from a value.

The argument must be an expression that returns a value of one of the following built-in data types: a time, a timestamp, or a character string. If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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

The other rules depend on the data type of the argument:

- If the argument is a time:
The result is that time.
- If the argument is a timestamp:
The result is the time part of the timestamp.
- If the argument is a character string:
The result is the time or time part of the timestamp represented by the character string. When a string representation of a time is SBCS with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a time value.

When a string representation of a time is mixed data with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a time value.

Note

Syntax alternatives: The CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

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

TIMESTAMP

►►—TIMESTAMP—(—*expression*—, —*expression*—)—►►

The TIMESTAMP function returns a timestamp from its argument or arguments.

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

- If only one argument is specified:

The argument must be an expression that returns a value of one of the following built-in data types: a timestamp or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be one of the following:

- A valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.
- A character string with an actual length of 7 that represents a valid date in the form *yyyynnn*, where *yyyy* are digits denoting a year, and *nnn* are digits between 001 and 366 denoting a day of that year.
- A character string with an actual length of 14 that represents a valid date and time in the form *yyyxxddhhmmss*, where *yyyy* is year, *xx* is month, *dd* is day, *hh* is hour, *mm* is minute, and *ss* is seconds.

- If both arguments are specified:

The first argument must be an expression that returns a value of one of the following built-in data types: a date or a character string. The second argument must be an expression that returns a value of one of the following built-in data types: a time or a character string.

If the first *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date. If the second *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a time. For the valid formats of string representations of dates and times, see “String Representations of Datetime Values” on page 68.

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.

When a string representation of a timestamp is SBCS data with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a timestamp value.

When a string representation of a timestamp is mixed data with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a timestamp value.

Note

Syntax alternatives: If only one argument is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Example

- Assume the following date and time values:

```
SELECT TIMESTAMP( DATE('1988-12-25'), TIME('17.12.30') )  
FROM SYSIBM.SYSDUMMY1
```

Returns the value '1988-12-25-17.12.30.000000'.

TIMESTAMPDIFF

TIMESTAMPDIFF

►—TIMESTAMPDIFF—(—*numeric-expression*—,—*character-expression*—)————►

The TIMESTAMPDIFF function returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.

The first argument must be a built-in data type of either INTEGER or SMALLINT. Valid values of interval (the first argument) are:

1	Fractions of a second
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	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 an integer. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The 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

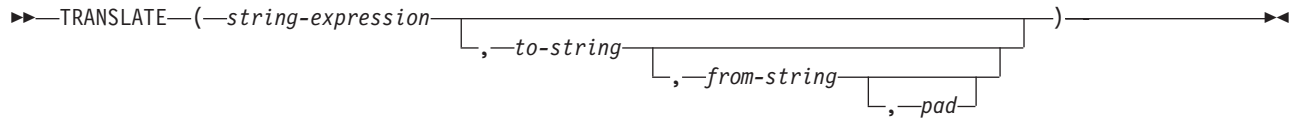
These assumptions are used when converting the information in the second argument, which is a timestamp duration, to the interval type specified in the first argument. The returned estimate may vary by a number of days. For example, if the number of days (interval 16) is requested for a difference in 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.

Example

- Estimate the age of employees in months.

```
SELECT
  TIMESTAMPDIFF(64,
    CAST(CURRENT_TIMESTAMP-CAST(BIRTHDATE AS TIMESTAMP) AS CHAR(22))
    AS AGE_IN_MONTHS
  FROM EMPLOYEE
```

TRANSLATE



The TRANSLATE function returns a value in which one or more characters in *string-expression* may have been converted into other characters.

string-expression

An expression that specifies the string to be converted *string-expression* must be a character string or a UCS-2 graphic string.

to-string

A string that specifies the characters to which certain characters in *string-expression* are to be converted. This string is sometimes called the *output translation table*. The string must be a character string constant. A character string argument must have an actual length that is not greater than 256.

If the length attribute of the *to-string* is less than the length attribute of the *from-string*, then the *to-string* is padded to the longer length using either the *pad* character if it is specified or a blank if a *pad* character is not specified. If the length attribute of the *to-string* is greater than the length attribute of the *from-string*, the extra characters in *to-string* are ignored without warning.

from-string

A string that specifies the characters that if found in *string-expression* are to be converted. This string is sometimes called the *input translation table*. When a character in *from-string* is found, the character in *string-expression* is converted to the character in *to-string* that is in the corresponding position of the character in *from-string*.

The string must be a character string constant. A character string argument must not have an actual length that is not greater than 256.

If there are duplicate characters in *from-string*, the first one scanning from the left is used and no warning is issued. The default value for *from-string* is a string starting with the character X'00' and ending with the character X'FF' (decimal 255).

pad

A string that specifies the character with which to pad *to-string* if its length is less than *from-string*. The string must be a character string constant with a length of 1. The default is an SBCS blank.

If the first argument is a UCS-2 graphic string, no other arguments may be specified.

If only the first argument is specified, the SBCS characters of the argument are converted to uppercase, based on the CCSID of the argument. Only SBCS characters are converted. The characters a-z are converted to A-Z, and characters with diacritical marks are converted to their uppercase equivalent, if any. If the first argument is UCS-2 graphic, the alphabetic UCS-2 characters are converted to uppercase. Refer to the UCS-2 level 1 mapping tables section of the Globalization topic in the iSeries Information Center for a description of the monocasing tables that are used for this conversion.

TRANSLATE

If more than one argument is specified, the result string is built character by character from *string-expression*, converting characters in *from-string* to the corresponding character in *to-string*. For each character in *string-expression*, the same character is searched for in *from-string*. If the character is found to be the *n*th character in *from-string*, the resulting string will contain the *n*th character from *to-string*. If *to-string* is less than *n* characters long, the resulting string will contain the pad character. If the character is not found in *from-string*, it is moved to the result string unconverted.

Conversion is done on a byte basis and, if used improperly, may result in an invalid mixed string. The SRTSEQ attribute does not apply to the TRANSLATE function.

The result of the function has the same data type, length attribute, actual length, and CCSID as the argument. If the first argument can be null, the result can be null. If the argument is null, the result is the null value.

Examples

- Monocase the string 'abcdef'.

```
SELECT TRANSLATE('abcdef')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'ABCDEF'.

- Monocase the mixed character string.

```
SELECT TRANSLATE('abs0Cs1def')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'AB^s₀C^s₁DEF'

- Given that the host variable SITE is a varying-length character string with a value of 'Pivabiska Lake Place'.

```
SELECT TRANSLATE(:SITE, '$', 'L')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Pivabiska \$ake Place'.

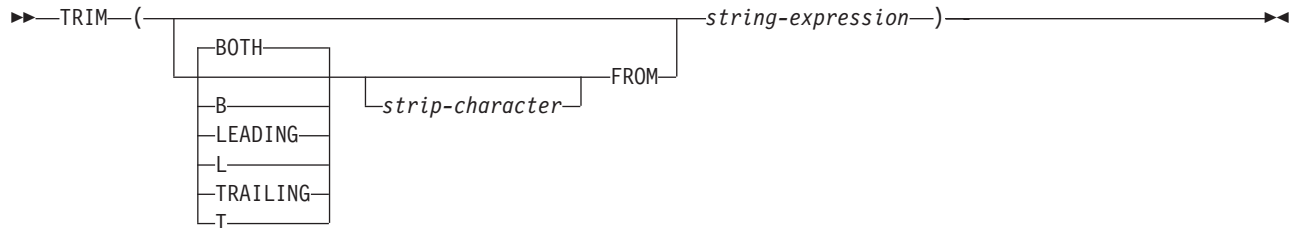
```
SELECT TRANSLATE(:SITE, '$$', 'L1')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'Pivabiska \$ake P\$ace'.

```
SELECT TRANSLATE(:SITE, 'pLA', 'Place', '.')
FROM SYSIBM.SYSDUMMY1
```

Returns the value 'pivAbiskA LAk. pLA..'.

TRIM



The TRIM function removes blanks or another specified character from the end or beginning of a string expression.

The *string-expression* must be a string expression.

The first argument, if specified, indicates whether characters are removed from the end or beginning of the string. If the first argument is not specified, then the characters are removed from both the end and the beginning of the string.

The second argument, if specified, is a single-character constant that indicates the binary, SBCS, or DBCS character that is to be removed. If *string-expression* is a binary string, the second argument must be a binary string constant. If *string-expression* is a DBCS graphic or DBCS-only string, the second argument must be a graphic constant consisting of a single DBCS character. If the second argument is not specified then:

- If *string-expression* is a binary string, then the default strip character is a hexadecimal zero (X'00').
- If *string-expression* is a DBCS graphic string, then the default strip character is a DBCS blank.
- If *string-expression* is a UCS-2 graphic string, then the default strip character is a UCS-2 blank.
- Otherwise, the default strip character is an SBCS blank.

The data type of the result depends on the data type of *string-expression*:

Data type of <i>expression</i>	Data type of the Result
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC
BLOB	BLOB
CLOB	CLOB
DBCLOB	DBCLOB

The length attribute of the result is the same as the length attribute of *string-expression*. The actual length of the result is the length of the expression minus the number of bytes removed. If all characters are removed, the result is an empty 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.

TRIM

The CCSID of the result is the same as that of the string.

The SRTSEQ attribute does not apply to the TRIM function.

Examples

- Assume the host variable HELLO of type CHAR(9) has a value of ' Hello'.

```
SELECT TRIM(:HELLO), TRIM( TRAILING FROM :HELLO)
FROM SYSIBM.SYSDUMMY1
```

Results in 'Hello' and ' Hello' respectively.

- Assume the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

```
SELECT TRIM( L '0' FROM :BALANCE )
FROM SYSIBM.SYSDUMMY1
```

Results in: '345.50'

- Assume the string to be stripped contains mixed data.

```
SELECT TRIM( BOTH 'S' 'I' FROM 'S AB C I' )
FROM SYSIBM.SYSDUMMY1
```

Results in: 'SAB CI'

TRUNCATE or TRUNC

► TRUNCATE (—*numeric-expression-1*—,—*numeric-expression-2*—) ►
 TRUNC

The TRUNCATE function returns *numeric-expression-1* truncated to some number of places to the right or left of the decimal point.

numeric-expression1

An expression that returns a value of any built-in numeric data type.

numeric-expression2

An expression that returns a small or large integer. The absolute value of integer specifies the number of places to the right of the decimal point for the result if *numeric-expression-2* is not negative, or to left of the decimal point if *numeric-expression-2* is negative.

If *numeric-expression-2* is not negative, *numeric-expression-1* is truncated to the *numeric-expression-2* number of places to the right of the decimal point.

If *numeric-expression-2* is negative, *numeric-expression-1* is truncated to the absolute value of (*numeric-expression-2*+1) number of places to the left of the decimal point.

If the absolute value of *numeric-expression-2* is larger than the number of digits to the left of the decimal point, the result is 0. For example, TRUNCATE(748.58,-4) = 0.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument.

If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

- Calculate the average monthly salary for the highest paid employee. Truncate the result to two places to the right of the decimal point.

```
SELECT TRUNCATE(MAX(SALARY/12, 2)
FROM EMPLOYEE
```

Because the highest paid employee in the sample employee table earns \$52750.00 per year, the example returns the value 4395.83.

- Calculate the number 873.726 truncated to 2, 1, 0, -1, -2, and -3 decimal places respectively.

```
SELECT TRUNCATE(873.726, 2),
       TRUNCATE(873.726, 1),
       TRUNCATE(873.726, 0),
       TRUNCATE(873.726, -1),
       TRUNCATE(873.726, -2),
       TRUNCATE(873.726, -3)
FROM SYSIBM.SYSDUMMY1
```

Returns the following values respectively:

```
0873.720 0873.700 0873.000 0870.000 0800.000 0000.000
```

- Calculate both positive and negative numbers.

TRUNCATE

```
SELECT TRUNCATE( 3.5, 0),  
       TRUNCATE( 3.1, 0),  
       TRUNCATE(-3.1, 0),  
       TRUNCATE(-3.5, 0)  
FROM SYSIBM.SYSDUMMY1
```

This example returns:

```
3.0  3.0  -3.0  -3.0
```

respectively.

UCASE

►► UCASE(*—string-expression—*) ◀◀

The UCASE function returns a string in which all the characters have been converted to uppercase characters, based on the CCSID of the argument.

The UCASE function is identical to the UPPER function. For more information, see “UPPER” on page 314.

VALUE

▶▶—VALUE—(—*expression*—, —*expression*—)—▶▶

The VALUE function returns the value of the first non-null expression.

The VALUE function is identical to the COALESCE scalar function. For more information, see “COALESCE” on page 200.

VARCHAR

VARCHAR

Character to Varchar

►► VARCHAR (*—character-expression* , *—length* , *—integer*)

length is optional and can be replaced by DEFAULT. *integer* is optional.

Graphic to Varchar

►► VARCHAR (*—graphic-expression* , *—length* , *—integer*)

length is optional and can be replaced by DEFAULT. *integer* is optional.

Integer to Varchar

►► VARCHAR (*—integer-expression*)

Decimal to Varchar

►► VARCHAR (*—decimal-expression* , *—decimal-character*)

decimal-character is optional.

Floating-point to Varchar

►► VARCHAR (*—floating-point-expression* , *—decimal-character*)

decimal-character is optional.

The VARCHAR function returns a character-string representation of:

- An integer number if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number if the first argument is a packed or zoned decimal number
- A double-precision floating-point number if the first argument is a DOUBLE or REAL
- A character string if the first argument is any type of character string
- A graphic string if the first argument is a UCS-2 graphic string

The result of the function is a varying-length 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.

Character to Varchar

character-expression

An expression that returns a value that is a built-in CHAR, VARCHAR, or CLOB data type.

length

Specifies the length attribute for the resulting varying length character string. The value must be between 1 and 32740 (32739 if nullable). If the first argument is mixed data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified:

- If the *character-expression* is an empty string constant, the length attribute of the result is 1.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer

Specifies the CCSID of the result. It must be a valid SBCS CCSID, mixed data CCSID, or 65535 (bit data). If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. If the third argument is 65535, then the result is bit data. If the third argument is a SBCS CCSID, then the first argument cannot be a DBCS-either or DBCS-only string.

If the third argument is not specified then:

- If the first argument is SBCS data, then the result is SBCS data. The CCSID of the result is the same as the CCSID of the first argument.
- If the first argument is mixed data (DBCS-open, DBCS-only, or DBCS-either), then the result is mixed data. The CCSID of the result is the same as the CCSID of the first argument.

Graphic to Varchar

graphic-expression

An expression that returns a value that is a GRAPHIC, VARGRAPHIC, and DBCLOB data type. It must not be DBCS-graphic data.

length

Specifies the length attribute for the resulting varying length character string. The value must be between 1 and 32740 (32739 if nullable). If the first argument contains DBCS data, the second argument cannot be less than 4.

If the second argument is not specified or DEFAULT is specified, the length attribute of the result is determined as follows (where *n* is the length attribute of the first argument):

- If the *graphic-expression* is the empty graphic string constant, the length attribute of the result is 1.
- If the result is SBCS data, the result length is *n*.
- If the result is mixed data, the result length is $(2.5*(n-1)) + 4$.

The actual length of the result is the minimum of the length attribute of the result and the actual length of *graphic-expression*. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning (SQLSTATE 01004) is returned unless the truncated characters were all blanks.

integer

Specifies the CCSID of the result. It must be a valid SBCS CCSID or mixed data CCSID. If the third argument is an SBCS CCSID, then the result is SBCS data. If the third argument is a mixed CCSID, then the result is mixed data. The third argument cannot be 65535.

VARCHAR

If the third argument is not specified, the CCSID of the result is the default CCSID at the current server. If the default CCSID is mixed data, then the result is mixed data. If the default CCSID is SBCS data, then the result is SBCS data.

Integer to Varchar

integer-expression

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

The result is a varying-length character string 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 argument is a small integer, the length attribute of the result is 6.
- If the argument is a large integer, the length attribute of the result is 11.
- If the argument is a big integer, the length attribute of the result is 20.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeroes are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit.

The CCSID of the result is the default SBCS CCSID at the current server.

Decimal to Varchar

decimal-expression

An expression that returns a value that is a packed or zoned decimal data type (either DECIMAL or NUMERIC). If a different precision and scale is desired, the DECIMAL scalar function can be used 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 must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see “Decimal Point” on page 102.

The result is a varying-length character string representation of the argument. The result includes a decimal character and up to p digits, where p is the precision of the *decimal-expression* with a preceding minus sign if the argument is negative. Leading zeros are not returned. Trailing zeros are returned.

The length attribute of the result is $2+p$ where p is the precision of the *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing characters are included. Leading zeros are not included. If the argument is negative, the result begins with a minus sign. Otherwise, the result begins with a digit.

The CCSID of the result is the default SBCS CCSID at the current server.

Floating-point to Varchar

floating-point expression

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

decimal-character

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal point. For more information, see “Decimal Point” on page 102.

The result is a varying-length character string representation of the argument in the form of a floating-point constant.

The length attribute of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If the argument is zero, the result is 0E0.

The CCSID of the result is the default SBCS CCSID at the current server.

Note

Syntax alternatives: If the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Example

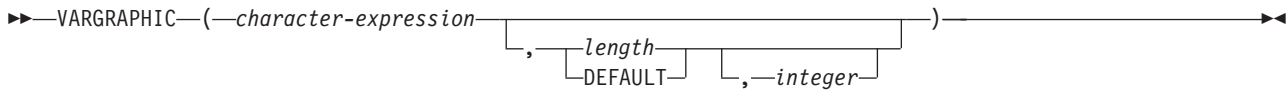
- Make EMPNO varying-length with a length of 10.

```
SELECT VARCHAR(EMPNO,10)
      INTO :VARHV
      FROM EMPLOYEE
```

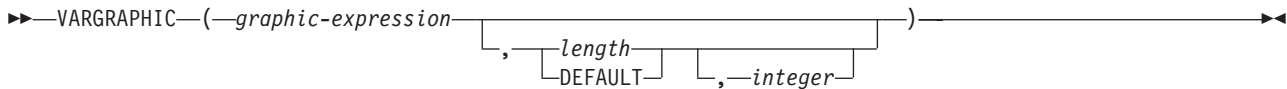
VARGRAPHIC

VARGRAPHIC

Character to Graphic



Graphic to Graphic



The VARGRAPHIC function returns a graphic string representation of a string expression.

The result of the function is a varying-length graphic string (VARGRAPHIC).

If the expression can be null, the result can be null. If the expression is null, the result is the null value. If the expression is an empty string or the EBCDIC string X'0E0F', the result is an empty string.

Character to Graphic

character-expression

Specifies a character string expression. It cannot be a CHAR or VARCHAR bit data.

length

Specifies the length attribute of the result and must be an integer constant between 1 and 16370 if the first argument is not nullable or between 1 and 16369 if the first argument is nullable.

If the second argument is not specified, or if DEFAULT is specified, the length attribute of the result is the same as the length attribute of the first argument, except if the expression is an empty string or the EBCDIC string X'0E0F', the length attribute of the result is 1.

The actual length of the result depends on the number of characters in the argument. Each character of the argument determines a character of the result. If the length attribute of the resulting varying-length string is less than the actual length of the first argument, truncation is performed and no warning is returned.

integer

Specifies the CCSID of the result. It must be a DBCS or UCS-2 CCSID. The CCSID cannot be 65535. If the CCSID represents UCS-2 graphic data, each character of the argument determines a character of the result. The nth character of the result is the UCS-2 equivalent of the nth character of the argument.

If *integer* is not specified then the CCSID of the result is determined by a mixed CCSID. Let M denote that mixed CCSID.

In the following rules, S denotes one of the following:

- If the string expression is a host variable containing data in a foreign encoding scheme, S is the result of the expression after converting the data to a CCSID in a native encoding scheme. (See “Character Conversion” on page 32 for more information.)
- If the string expression is data in a native encoding scheme, S is that string expression.

M is determined as follows:

- If the CCSID of S is a mixed CCSID, M is that CCSID.
- If the CCSID of S is an SBCS CCSID:
 - If the CCSID of S has an associated mixed CCSID, M is that CCSID.
 - Otherwise the operation is not allowed.

The following table summarizes the result CCSID based on M.

M	Result CCSID	Description	DBCS Substitution Character
930	300	Japanese EBCDIC	X'FEFE'
933	834	Korean EBCDIC	X'FEFE'
935	837	S-Chinese EBCDIC	X'FEFE'
937	835	T-Chinese EBCDIC	X'FEFE'
939	300	Japanese EBCDIC	X'FEFE'
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'

The equivalence of SBCS and DBCS characters depends on M. Regardless of the CCSID, every double-byte code point in the argument is considered a DBCS character, and every single-byte code point in the argument is considered an SBCS character with the exception of the EBCDIC mixed data shift codes X'0E' and X'0F'.

- If the nth character of the argument is a DBCS character, the nth character of the result is that DBCS character.
- If the nth character of the argument is an SBCS character that has an equivalent DBCS character, the nth character of the result is that equivalent DBCS character.
- If the nth character of the argument is an SBCS character that does not have an equivalent DBCS character, the nth character of the result is the DBCS substitution character.

Graphic to VarGraphic

graphic-expression

An expression that returns a value that is a graphic string.

length

Specifies the length attribute of the result and must be an integer constant between 1 and 16370 if the first argument is not nullable or between 1 and 16369 if the first argument is nullable.

If the second argument is not specified, or if DEFAULT is specified, the length attribute of the result is the same as the length attribute of the first argument, except if the expression is an empty string, the length attribute of the result is 1.

VARGRAPHIC

The actual length of the result depends on the number of characters in *graphic-expression*. If the length of *graphic-expression* is greater than the length specified, the result is truncated and no warning is returned.

integer

Specifies the CCSID of the result. It must be a DBCS or UCS-2 CCSID. The CCSID cannot be 65535.

If *integer* is not specified then the CCSID of the result is the CCSID of the first argument.

Note

Syntax alternatives: If the first argument is *graphic-expression* and the length attribute is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Example

- Using the EMPLOYEE table, set the host variable VAR_DESC (VARGRAPHIC(24)) to the VARGRAPHIC equivalent of the first name (FIRSTNME) for employee number (EMPNO) '000050'.

```
SELECT VARGRAPHIC(FIRSTNME)
      INTO :VAR_DESC
      FROM EMPLOYEE
      WHERE EMPNO = '000050'
```

WEEK

►►—WEEK—(—*expression*—)—————►►

The WEEK function returns an integer between 1 and 54 that represents the week of the year. The week starts with Sunday, and January 1 is always in the first week.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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

Example

- Using the PROJECT table, set the host variable WEEK (INTEGER) to the week that project ('PL2100') ended.

```
SELECT WEEK(PRENDATE)
  INTO :WEEK
  FROM PROJECT
  WHERE PROJNO = 'PL2100'
```

Results in WEEK being set to 38.

- Assume that table X has a DATE column called DATE_1 with various dates from the list below.

```
SELECT DATE_1, WEEK(DATE_1)
  FROM X
```

Results in the following list shows what is returned by the WEEK function for various dates.

1997-12-28	53
1997-12-31	53
1998-01-01	1
1999-01-01	1
1999-01-04	2
1999-12-31	53
2000-01-01	1
2000-01-03	2

WEEK_ISO

►► WEEK_ISO (—*expression*—) ◀◀

The WEEK_ISO function returns an integer between 1 and 53 that represents the week of the year. The week starts with Monday. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. Thus, it is possible to have up to 3 days at the beginning of the year appear as the last week of the previous year or to have up to 3 days at the end of a year appear as the first week of the next year.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, or a character string.

If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.

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 PROJECT table, set the host variable WEEK (INTEGER) to the week that project ('AD2100') ended.

```
SELECT WEEK_ISO(PRENDATE)
      INTO :WEEK
      FROM PROJECT
      WHERE PROJNO = 'AD3100'
```

Results in WEEK being set to 5.

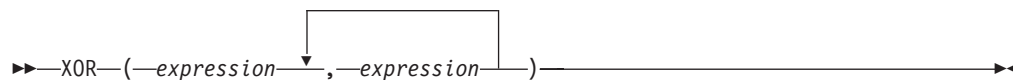
- Assume that table X has a DATE column called DATE_1 with various dates from the list below.

```
SELECT DATE_1, WEEK_ISO(DATE_1)
      FROM X
```

Results in the following:

1997-12-28	52
1997-12-31	1
1998-01-01	1
1999-01-01	53
1999-01-04	1
1999-12-31	52
2000-01-01	52
2000-01-03	1

XOR



The XOR function returns a string that is the logical XOR of the argument strings. This function takes the first argument string, does an XOR comparison with the next string, and then continues to do XOR comparisons for each successive argument using the previous result. If an argument is encountered that is shorter than the previous result, it is padded with blanks.

The arguments must be character strings but cannot be LOBs. The arguments cannot be mixed data character strings or graphic strings.

The arguments are converted, if necessary, to the attributes of the result. The attributes of the result are determined as follows:

- If all the arguments are fixed-length strings, the result is a fixed-length string of length n , where n is the length of the longest argument.
- If any argument is a varying-length string, the result is a varying-length string with length attribute n , where n is the length attribute of the argument with greatest length attribute. The actual length of the result is m , where m is the actual length of the longest argument.

If an argument can be null, the result can be null; if an argument is null, the result is the null value.

The CCSID of the result is 65535.

Example

- Assume the host variable L1 is a CHARACTER(2) host variable with a value of X'E1E1', host variable L2 is a CHARACTER(3) host variable with a value of X'F0F000', and host variable L3 is a CHARACTER(4) host variable with a value of X'0000000F'.

```
SELECT XOR(:L1,:L2,:L3)
FROM SYSIBM.SYSDUMMY1
```

Returns the value X'1111404F'. In this case, the shorter results are padded with blanks (X'40'), so the logical XOR result differs from the result in the following example.

```
SELECT XOR(:L3,:L2,:L1)
FROM SYSIBM.SYSDUMMY1
```

Returns the value X'1111400F'.

YEAR

►►—YEAR—(—*expression*—)—————►►

The YEAR function returns the year part of a value.

The argument must be an expression that returns a value of one of the following built-in data types: a date, a timestamp, a character string, or a numeric data type.

- If *expression* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date or timestamp. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 68.
- If *expression* is a number, it must be a date duration or timestamp duration. For the valid formats of datetime durations, see “Datetime Operands and Durations” on page 134.

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 or a timestamp or a valid character-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
```

ZONED

Numeric to Zoned Decimal

►► ZONED (*numeric-expression* [, *precision-integer* [, *scale-integer*]])

Character to Zoned Decimal

►► ZONED (*character-expression* [, *precision* [, *scale* [, *decimal-character*]]])

The ZONED function returns a zoned decimal representation of:

- A number
- A character string representation of an integer
- A character string representation of a decimal number
- A character string representation of a floating-point number

The result of the function is a zoned 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 Zoned Decimal

numeric-expression

An expression that returns a value of any built-in numeric data type.

precision

An integer constant with a value greater than or equal to 1 and less than or equal to 31.

The default for *precision* depends on the data type of the *numeric-expression*:

- 15 for floating point, decimal, numeric, or nonzero scale binary
- 19 for big integer
- 11 for large integer
- 5 for small integer

scale

An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

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 . 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 Zoned Decimal

ZONED

character-expression

An expression that must contain a character-string representation of a number. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer or decimal constant. The expression must not be a CLOB.

precision

An integer constant that is greater than or equal to 1 and less than or equal to 31. If not specified, the default is 15.

scale

An integer constant that is greater than or equal to 0 and less than or equal to *precision*. If not specified, the default is 0.

decimal-character

Specifies the single-byte character constant that was used to delimit the decimal digits in *character-expression* from the whole part of the number. The character must be a period or comma. If the second argument is not specified, the decimal point is the default decimal separator character. For more information, see “Decimal Point” on page 102.

The result is the same number that would result from `CAST(character-expression AS NUMERIC(p,s))`. Digits are truncated from the end if the number of digits to the right of the *decimal-character* is greater than the scale *s*. An error occurs if the number of significant digits to the left of the *decimal-character* (the whole part of the number) in *character-expression* is greater than *p-s*. The default decimal separator character is not valid in the substring if the *decimal-character* argument is specified.

Note

Syntax alternatives: When the precision is specified, the CAST specification should be used for maximal portability. For more information, see “CAST Specification” on page 143.

Examples

- Assume the host variable Z1 is a decimal host variable with a value of 1.123.

```
SELECT ZONED(:Z1,15,14)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 1.123000000000000.

- Assume the host variable Z1 is a decimal host variable with a value of 1123.

```
SELECT ZONED(:Z1,11,2)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 1123.00.

- Likewise,

```
SELECT ZONED(:Z1,4)
FROM SYSIBM.SYSDUMMY1
```

Returns the value 1123.

Chapter 4. Queries

A *query* specifies a result table or an intermediate result table.

A query is a component of certain SQL statements. There are three forms of a query:

- The *subselect*
- The *fullselect*
- The *select-statement*

Another form of select is described under “SELECT INTO” on page 727.

See also, “Authorization”.

Authorization

For any form of a query, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement,
 - The SELECT privilege on the table or view, and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table

The authorization ID of the statement has the SELECT privilege on a view when:

- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views on which this view is directly or indirectly dependent. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

If an *expression* includes a function, the authorization ID of the statement must include at least one of the following for each user-defined function:

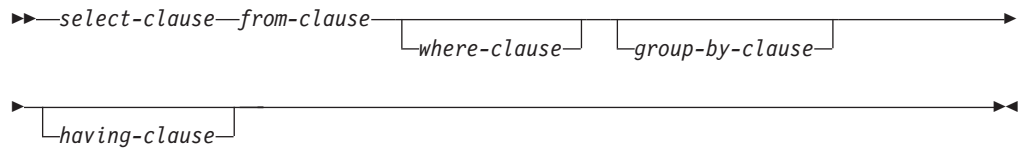
- The EXECUTE privilege on the function
- Administrative authority

The authorization ID of the statement has the EXECUTE privilege on a function when:

- It is the owner of the function,
- It has been granted the EXECUTE privilege on the function, or
- It has been granted the system authorities of *OBJOPR and *EXECUTE on the function.

subselect

subselect



The *subselect* is a component of the fullselect and the CREATE VIEW statement. It is also a component of certain predicates, which in turn, are components of a subselect.

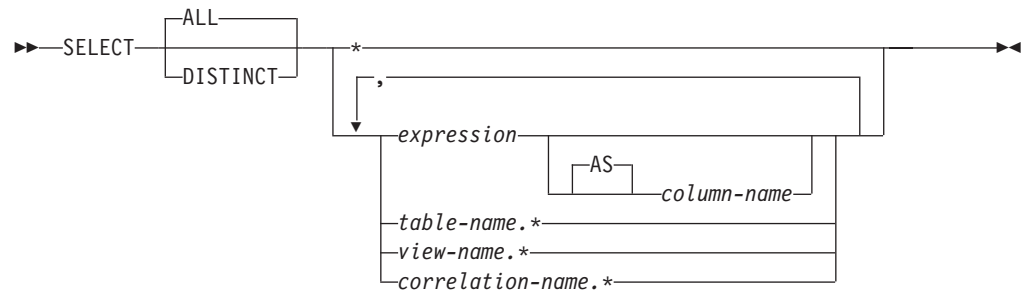
| A *scalar-subselect* is a subselect, enclosed in parentheses, that returns a single result
| row and a single result column. If the result of the subselect is no rows, then the
| null value is returned. An error is returned if there is more than one row in the
| result.

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 sequence of the (hypothetical) operations is:

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

Selects all rows of the final result table and does not eliminate duplicates. This is the default.

DISTINCT

Eliminates all but one of each set of duplicate rows of the final result table.

Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value in the second row. (For determining duplicate rows, two null values are considered equal.) Sort sequence is also used for determining distinct values.

DISTINCT is not allowed if the *select list* contains a LOB or DATALINK column.

Select List Notation

- * Represents a list of names that identify the columns of table R in the order the columns are produced by the **FROM** clause. The list of names is established when the statement containing the **SELECT** clause is prepared. Hence, * does not identify any columns that have been added to a table after the statement has been prepared.

expression

Specifies the values of a result column. Each *column-name* in the *expression* must unambiguously identify a column of R.

column-name or **AS** *column-name*

Names or renames the result column. The name must not be qualified and does not have to be unique.

*name.**

Represents a list of names that identify the columns of *name* in the order the columns are produced by the **FROM** clause. The *name* can 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.

select-clause

The list of names is established when the statement containing the SELECT clause is prepared. Hence, * does not identify any columns that have been added to a table after the statement has been prepared.

Normally, when SQL statements are implicitly rebound, the list of names is not re-established. Therefore, the number of columns returned by the statement does not change. However, there are four cases where the list of names is established again and the number of columns can change:

- When an SQL program or SQL package is saved and then restored on an iSeries system that is not the same release as the system from which it was saved.
- When SQL naming is specified for an SQL program or package and the owner of the program has changed since the SQL program or package was created.
- When an SQL statement is executed for the first time after the install of a more recent release of OS/400.
- When the SELECT * occurs in the subselect of an INSERT statement or in a subselect within a predicate, and a table or view referenced in the subselect has been deleted and recreated with additional columns.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established at prepare time), and cannot exceed 8000. The result of a subquery must be a single expression, unless the subquery is used in the EXISTS predicate.

Applying the Select List

The results of applying the select list to R depend on whether or not GROUP BY or HAVING is used:

If GROUP BY or HAVING is used:

- Each expression that contains a *column-name* in the select list must identify a grouping expression or be specified within a column function:
 - If the grouping expression is a column name, the select list may apply additional operators to the column name. For example, if the grouping expression is a column C1, the select list may contain C1+1.
 - If the grouping expression is not a column name, the select list may not apply additional operators to the expression. For example, if the grouping expression is C1+1, the select list may contain C1+1, but not (C1+1)/8.
- The RRN, PARTITION, NODENAME, and NODENUMBER functions cannot be specified in the select list.
- 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:

- The select list must not include any column functions, or it must be entirely a list of column functions.
- If the select list does not include column functions, 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, 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 allow null values if they are derived from:

- Any column function but COUNT and COUNT_BIG
- Any column that allows null values
- A scalar function or expression with an operand that allows null values
- A host variable that has an indicator variable
- A result of a UNION if at least one of the corresponding items in the select list is nullable
- An arithmetic expression
- A scalar subselect
- A user-defined scalar or table function

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 a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), then the result column name is the unqualified name of that column.
- All other result columns are unnamed.

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 or BIGINT (if the value of the constant is outside the range of INTEGER, but within the range of BIGINT).
a decimal or floating-point constant	The same as the data type of the constant, with the same precision and scale for decimal constants.
the name of any numeric host variable	The same as the data type of the variable, with the same precision and scale for decimal variables. If the data type of the variable is not identical to an SQL data type (for example, DISPLAY SIGN LEADING SEPARATE in COBOL), the result column is decimal.
an expression	The same as the data type of the result, with the same precision and scale for decimal results as described under "Expressions" on page 129.
any function	The data type of the result of the function. For a built-in function, see Chapter 3 to determine the data type of the result. For a user-defined function, the data type of the result is what was defined in the CREATE FUNCTION statement for the function.
the name of any string column	The same as the data type of the column, with the same length attribute.

select-clause

When the expression is:	The data type of the result column is:
the name of any string host variable	The same as the data type of the variable, with a length attribute equal to the length of the variable. 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 <i>n</i>	VARCHAR(<i>n</i>)
a graphic-string constant of length <i>n</i>	VARGRAPHIC(<i>n</i>)
the name of a datetime column, or an ILE RPG compiler or ILE COBOL compiler datetime host variable	The same as the data type of the column or host variable.
the name of a datalink column	A datalink, with the same length attribute.
the name of a row ID column or a row ID host variable	ROWID
the name of a distinct type column	the same as the distinct type of the column, with the same length, precision, and scale attributes, if any.

|
|
|
|

from-clause



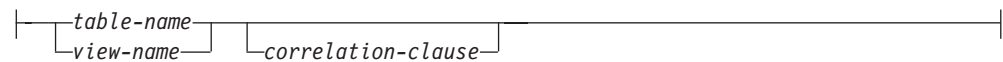
The FROM clause specifies an intermediate result table.

If only one *table-reference* is specified, the intermediate result table is simply the result of that *table-reference*. If more than one *table-reference* is specified in the FROM clause, 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*.

table-reference



single-table:



nested-table-expression:

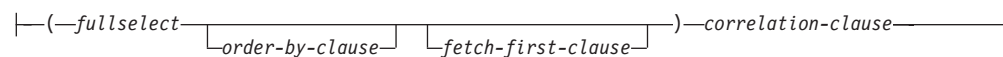
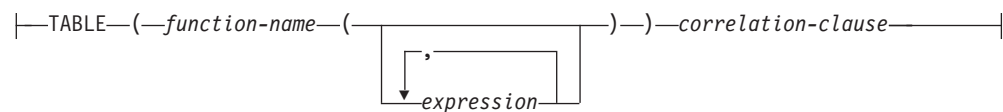
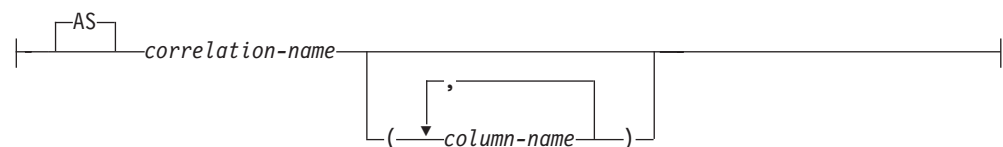


table-function:



correlation-clause:



from-clause

A *table-reference* specifies an intermediate result table.

- If a single table or view is identified, the intermediate result table is simply that table or view.
- A fullselect in parentheses in a WITH clause is called a *nested table expression*.⁴⁰ If a nested table expression is specified, the result table is the result of that nested table expression. The columns of the result do not need unique names, but a column with a non-unique name cannot be explicitly referenced.
- If a *function-name* is specified, the intermediate result table is the set of rows returned by the table function.
- If a *joined-table* is specified, the intermediate result table is the result of one or more join operations. For more information, see “joined-table” on page 338.

The list of names in the FROM clause must conform to these rules:

- Each *table-name* and *view-name* must name an existing table or view at the current server or the table-name of a *common-table expression* (see “common-table-expression” on page 350) defined preceding the subselect containing the table-reference.
- The exposed names must be unique. An exposed name is a correlation-name, a table-name that is not followed by a correlation-name, or a view-name that is not followed by a correlation-name.
- Each *function-name*, together with the types of its arguments, must resolve to a table function that exists at the current server. An algorithm called function resolution, which is described on “Function resolution” on page 124, uses the function name and the arguments to determine the exact function to use. Unless given column names in the *correlation-clause*, the column names for a table function are those specified on the RETURNS clause of the CREATE FUNCTION statement. This is analogous to the column names of a table, which are defined in the CREATE TABLE.

Each *correlation-name* is defined as a designator of the intermediate result table specified by the immediately preceding table-reference. A correlation name must be specified for a *nested table expression* and a *table function*.

The exposed names of all table references should be unique. An exposed name is:

- A *correlation-name*
- A *table-name* or *view-name* that is not followed by a *correlation-name*

Any qualified reference to a column for a table, view, nested table expression, or table function 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*, *nested-table-expression* or *table-function*. If a column list is specified, there must be a name in the column list for each column in the table or view and for each result column in the *nested-table-expression* or *table-function*. For more information, see “Correlation Names” on page 108.

In general, *nested-table-expressions* and *table-functions* can be specified in any *from-clause*. Columns from the nested table expressions and table functions can be referenced in the select list and in the rest of the subselect using the correlation

40. A *nested table expression* is also called a *derived table*.

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.

Correlated References in table-references: Correlated references can be used in *nested table expressions*. The basic rule that applies is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. For more information see “Column Name Qualifiers to Avoid Ambiguity” on page 110

A table function can contain one or more correlated references to other tables in the same FROM clause if the referenced tables precede the reference in the left-to-right order of the tables in the FROM clause. Otherwise, only references to higher levels in the hierarchy of subqueries is allowed.

Example 1: The following example is valid:

```
SELECT D.DEPTNO, D.DEPTNAME, EMPINFO.AVGSAL, EMPINFO.EMPCOUNT
FROM DEPARTMENT D,
     (SELECT AVG(E.SALARY) AS AVGSAL, COUNT (*) AS EMPCOUNT
      FROM EMPLOYEE E
      WHERE E.WORKDEPT =
            (SELECT X.DEPTNO
             FROM DEPARTMENT X
             WHERE X.DEPTNO = E.WORKDEPT ) ) AS EMPINFO
```

The following example is not valid because the reference to D.DEPTNO in the WHERE clause of the *nested-table-expression* attempts to reference a table that is outside the hierarchy of subqueries:

```
SELECT D.DEPTNO, D.DEPTNAME,
       EMPINFO.AVGSAL, EMPINFO.EMPCOUNT
FROM DEPARTMENT D,
     (SELECT AVG(E.SALARY) AS AVGSAL, COUNT (*) AS EMPCOUNT
      FROM EMPLOYEE E
      WHERE E.WORKDEPT = D.DEPTNO ) AS EMPINFO
```

Example 2: The following example of a table function is valid:

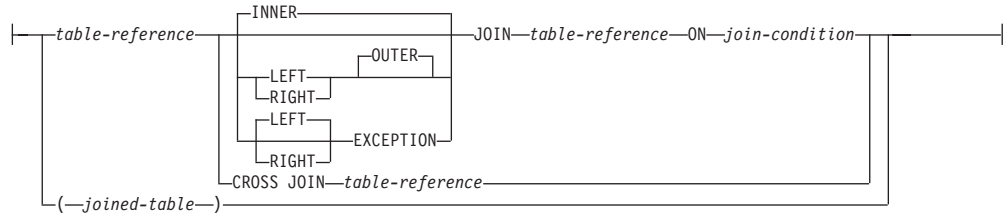
```
SELECT t.c1, z.c5
FROM t, TABLE(tf3 (t.c2 ) ) AS z WHERE t.c3 = z.c4
```

The following example is not valid because the reference to t.c2 is for a table that is to the right of the table function in the FROM clause:

```
SELECT t.c1, z.c5
FROM TABLE(tf6 (t.c2 ) ) AS z, t
WHERE t.c3 = z.c4
```

from-clause

joined-table



A *joined-table* specifies an intermediate result table that is the result of either an inner, outer, cross, or exception join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, LEFT EXCEPTION, RIGHT EXCEPTION or CROSS to its operands.

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
LEFT 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)
LEFT JOIN (TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1)
ON TB1.C1=TB3.C1
```

An inner join combines each row of the left table with every row of the right table keeping only the rows where the join-condition is true. Thus, the result table may be missing rows of from either or both of the joined tables. Outer joins include the rows produced by the inner join as well as the missing rows, depending on the type of outer join. Exception joins include only the missing rows, depending on the type of exception join as follows:

- *Left outer.* Includes the rows from the left table that were missing from the inner join.
- *Right outer.* Includes the rows from the right table that were missing from the inner join.
- *Left exception.* Includes only the rows from the left table that were missing from the inner join.
- *Right exception.* Includes only the rows from the right table that were missing from the inner join.

A joined table can be used in any context in which any form of the SELECT statement is used. A view or a cursor is read-only if its SELECT statement includes a joined table.

Join Condition: The *join-condition* is a *search-condition* that must conform to these rules:

- It cannot contain a quantified subquery, IN predicate with a subselect, or EXISTS subquery. It can contain basic predicate subqueries and scalar-subselects.

- Each column name must unambiguously identify a column in one of the tables in the *from-clause*.
- Column functions cannot be used in the *expression*.

For any type of join, column references in an expression of the join-condition are resolved using the rules for resolution of column name qualifiers specified in “Column Names” on page 108 before any rules about which tables the columns must belong to are applied.

Join Operations: A *join-condition* specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of 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. In the case of OUTER joins, 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.

INNER JOIN or JOIN

The result of T1 INNER JOIN T2 consists of their paired rows.

Using the INNER JOIN syntax with a *join-condition* will produce the same result as specifying the join by listing two tables in the FROM clause separated by commas and using the *where-clause* to provide the condition.

LEFT JOIN or LEFT OUTER JOIN

The result of T1 LEFT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.

RIGHT JOIN or RIGHT OUTER JOIN

The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.

LEFT EXCEPTION JOIN and EXCEPTION JOIN

The result of T1 LEFT EXCEPTION JOIN T2 consists only of each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.

RIGHT EXCEPTION JOIN

The result of T1 RIGHT EXCEPTION JOIN T2 consists only of each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.

CROSS JOIN

The result of T1 CROSS JOIN T2 consists of each row of T1 paired with each row of T2. CROSS JOIN is also known as cartesian product.

where-clause

►—WHERE—*search-condition*—◄

The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the statement.

The *search-condition* must conform to the following rules:

- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table, view, common table expression, or nested table expression 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 executed for each row of R if it includes a correlated reference to a column of R. A subquery with no correlated reference is typically executed just once.

If a sort sequence other than *HEX is in effect when the statement that contains the WHERE clause is executed and if the search-condition contains predicates that have SBCS, mixed, or UCS-2 data, then the comparison for those predicates is done using weighted values. The weighted values are derived by applying the sort sequence to the operands of the predicate.

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.

A *grouping-expression* is an *expression* used in defining the grouping of R. Each column name included in a *grouping-expression* must unambiguously identify a column of R. LOB and DataLink columns cannot be used in a *grouping-expression*. The length attribute of each *grouping-expression* must not be more than 2000, or 1999 if the expression is nullable. A *grouping-expression* cannot include a function that is non-deterministic, or the RRN, PARTITION, NODENAME, or NODENUMBER functions.

The result of the GROUP BY clause is a set of groups of rows. In each group of more than one row, all values of each *grouping-expression* are equal, and all rows with the same set of values of the *grouping-expressions* are in the same group. For grouping, all null values for a *grouping-expression* are considered equal.

Because every row of a group contains the same value of any *grouping-expression*, *grouping-expressions* can be used in a search condition in a HAVING clause, in the SELECT clause, or in a *sort-key-expression* of an ORDER BY clause (see “order-by-clause” on page 351 for details). In each case, the reference specifies only one value for each group. The *grouping-expression* specified in these clauses must exactly match the *grouping-expression* in the GROUP BY clause, except that blanks are not significant. For example, a *grouping-expression* of SALARY*.10

will match the expression in a *having-clause* of
HAVING SALARY*.10

but will not match

HAVING .10 *SALARY
 or
HAVING (SALARY*.10)+100

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.

The number of *grouping-expressions* must not exceed 120 and the sum of their length attributes must not exceed 2000–*n* bytes, where *n* is the number of *grouping-expressions* specified that allow nulls.

If a sort sequence other than *HEX is in effect when the statement that contains the GROUP BY clause is executed, the rows are placed into groups using the weighted

group-by-clause

values. The weighted values are derived by applying the sort sequence to the SBCS data *grouping-expressions*, to the SBCS data of mixed data *grouping-expressions*, and to UCS-2 data *grouping-expressions*.

having-clause

►—HAVING—*search-condition*—◄

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the *search-condition* is true. R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered a single group with no grouping expressions.

Each expression that contains a *column-name* in the search condition must do one of the following:

- Unambiguously identify a grouping expression 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, view, common table expression, or nested table expression identified in an outer subselect.

The RRN, PARTITION, NODENAME, and NODENUMBER functions cannot be specified in the HAVING clause unless it is within a column function. See "Functions" in Chapter 3 for restrictions that apply to the use of column functions.

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 examples 6 and 7 under "Examples of a subselect" on page 344.

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, any column name in the select list must appear within a column function.

If a sort sequence other than *HEX is in effect when the statement that contains the HAVING clause is executed and if the search-condition contains predicates that have SBCS, mixed, or UCS-2 data, the comparison for those predicates is done using weighted values. The weighted values are derived by applying the sort sequence to the operands in the predicate.

Examples of a subselect

Example 1

Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

Example 2

Join the EMPPROJACT and EMPLOYEE tables, select all the columns from the EMPPROJACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

```
SELECT EMPPROJACT.*, LASTNAME
FROM EMPPROJACT, EMPLOYEE
WHERE EMPPROJACT.EMPNO = EMPLOYEE.EMPNO
```

Example 3

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
```

This subselect could also be written as follows:

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
FROM EMPLOYEE INNER JOIN DEPARTMENT
ON WORKDEPT = DEPTNO
WHERE YEAR(BIRTHDATE) < 1930
```

Example 4

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 5

Select all the rows of EMPPROJACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

```
SELECT * FROM EMPPROJACT
WHERE EMPNO IN (SELECT EMPNO
FROM EMPLOYEE
WHERE WORKDEPT = 'E11')
```

Example 6

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 7

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)
```

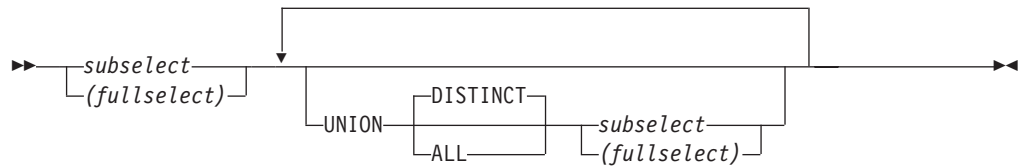
In contrast to example 6, the subquery in the HAVING clause would need to be executed for each group.

Example 8

Join the EMPLOYEE and EMPPROJACT tables, select all of the employees and their project numbers. Return even those employees that do not have a project number currently assigned.

```
SELECT EMPLOYEE.EMPNO, PROJNO
FROM EMPLOYEE LEFT OUTER JOIN EMPPROJACT
ON EMPLOYEE.EMPNO = EMPPROJACT.EMPNO
```

Any employee in the EMPLOYEE table that does not have a project number in the EMPPROJACT table will return one row in the result table containing the EMPNO value and the null value in the PROJNO column.

fullselect


The *fullselect* is a component of the *select-statement* and the CREATE VIEW statement.

A fullselect used that is enclosed in parenthesis is called a *subquery*. For example, a *subquery* can be used in a search condition.

A *fullselect* specifies a result table. If UNION is not used, the result of the fullselect is the result of the specified subselect.

UNION DISTINCT or UNION ALL

Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

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 the result table has the result column name. If the *n*th column of R1 and the *n*th column of R2 do not have the same names, then the result column is unnamed.

Two rows are duplicates if each value in the first is equal to the corresponding value of the second. (For determining duplicates, two null values are considered equal.)

If a sort sequence other than *HEX is in effect when the statement that contains the UNION keyword is executed and if the result tables contain columns that have SBCS, UCS-2, or mixed data, the comparison for those columns is done using weighted values. The weighted values are derived by applying the sort sequence to each value.

Both UNION and UNION ALL are associative operations. However, when UNION and UNION ALL are used in the same statement, the result depends on the order in which the operations are performed. Operations within parentheses are performed first. When the order is not specified by parentheses, operations are performed in left-to-right order.

Rules for Columns

R1 and R2 must have the same number of columns, and the data type of the *n*th column of R1 must be compatible with the data type of the *n*th column of R2. Character-string values are not compatible with datetime values.

The *n*th column of the result of UNION and UNION ALL is derived from the *n*th columns of R1 and R2. The attributes of the result columns are determined using the rules for result columns. For more information see “Rules for Result Data Types” on page 93.

If UNION is specified, no column can be a LOB or DATALINK column.

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 EMPPROJECT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```
SELECT EMPNO FROM EMPLOYEE
WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO FROM EMPPROJECT
WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 3

Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

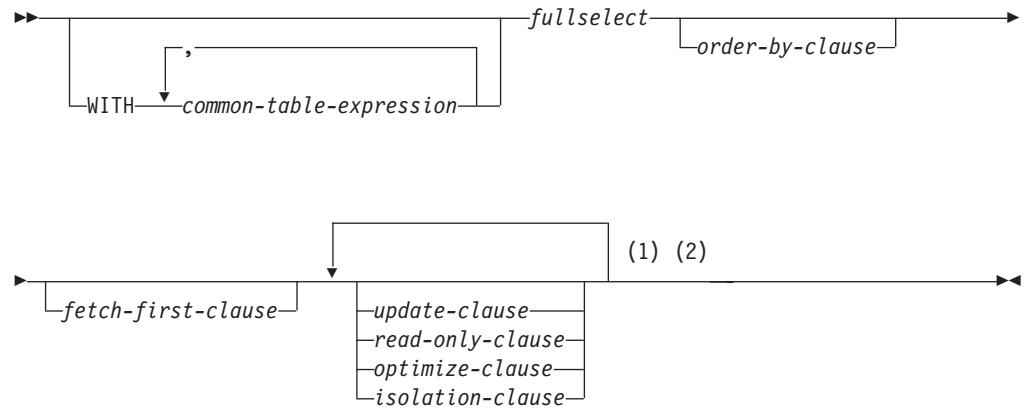
```
SELECT EMPNO FROM EMPLOYEE
WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO FROM EMPPROJECT
WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 4

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 EMPPROJECT table with 'empproject'. 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, 'empproject' FROM EMPPROJECT
WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```


select-statement

**Notes:**

- 1 The *update-clause* and *read-only-clause* cannot both be specified in the same *select-statement*.
- 2 Each clause may be specified only once.

The *select-statement* is the form of a query that can be directly specified in a *DECLARE CURSOR* statement, prepared and then referenced in a *DECLARE CURSOR* statement, or directly specified in an SQLJ assignment clause. It can also be issued interactively, using the interactive facility (STRSQL command), causing a result table to be displayed at your work station. In any case, the table specified by a *select-statement* is the result of the *fullselect*.

order-by-clause

**sort-key:**

The ORDER BY clause specifies an ordering of the rows of the result table. If a single sort specification (one *sort-key* with associated ascending or descending ordering specification) is identified, the rows are ordered by the values of that 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.

If a sort sequence other than *HEX is in effect when the statement that contains the ORDER BY clause is executed and if the ORDER BY clause involves sort specifications that have SBCS, UCS-2, or mixed data, the comparison for those sort specifications is done using weighted values. The weighted values are derived by applying the sort sequence to the values of the sort specifications.

A named column in the select list may be identified by a *sort-key* that is a *integer* or a *column-name*. An unnamed column in the select list may be identified by a *integer* or, in some cases by a *sort-key-expression* that matches the expression in the select list (see details of *sort-key-expression*). “Names of result columns” on page 333 defines when result columns are unnamed. If the fullselect includes a UNION operator, see “fullselect” on page 346 for the rules on named columns in a fullselect.

Ordering is performed in accordance with the comparison rules described in Chapter 2. The null value is higher than all other values. If your ordering specification does not determine a complete ordering, rows with duplicate values of the last identified *sort-key* have an arbitrary order. If the ORDER BY clause is not specified, the rows of the result table have an arbitrary order.

The number of *sort-keys* must not exceed 10000-*n* and the sum of their length attributes must not exceed 10000-*n* bytes (where *n* is the number of *sort-keys* specified that allow nulls).

column-name

Must unambiguously identify a column of the result table. The column must not be a LOB or DATALINK column. The rules for unambiguous column references are the same as in the other clauses of the fullselect. See “Column Name Qualifiers to Avoid Ambiguity” on page 110 for more information.

If the fullselect includes a UNION or UNION ALL, the column name cannot be qualified.

The *column-name* may also identify a column name of a table, view, or *nested-table-expression* identified in the FROM clause if the query is a subselect.

order-by-clause

| An error occurs if the subselect includes a column function in the select list
| and the *column-name* is not a *grouping-expression*.

integer

Must be greater than 0 and not greater than the number of columns in the result table. The integer *n* identifies the *n*th column of the result table. The identified column must not be a LOB or DATALINK column.

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 contain RRN, PARTITION, NODENAME, or
| NODENUMBER if the *fullselect* includes a UNION or UNION ALL. The result
| of the *sort-key-expression* must not be a LOB or DATALINK.

If the subselect is grouped, the *sort-key-expression* can be an expression in the select list of the subselect or can include a *grouping-expression* from the GROUP BY clause of the subselect.

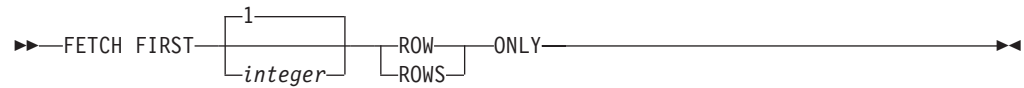
ASC

Uses the values of the column in ascending order. This is the default.

DESC

Uses the values of the column in descending order.

fetch-first-clause



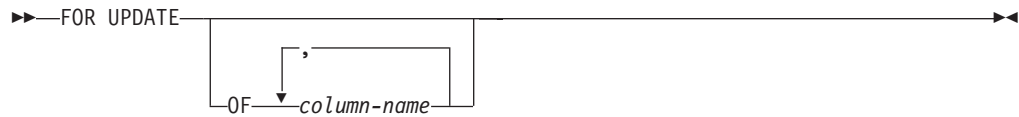
The *fetch-first-clause* sets a maximum number of rows that can be retrieved. It lets the database manager know that only *integer* rows should be made available to be retrieved, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond *integer* rows is handled the same way as normal end of data (SQLSTATE 02000). 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 *order-by-clause* and *fetch-first-clause* are specified, the FETCH FIRST operation is always performed on the ordered data. Specification of the *fetch-first-clause* in a *select-statement* makes the result table *read-only*. A *read-only* result table must not be referred to in an UPDATE or DELETE statement. The *fetch-first-clause* cannot appear in a statement containing an UPDATE clause.

update-clause

update-clause



The 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. The clause must not be specified if the result table of the fullselect is read-only.

If the UPDATE clause is specified without column names, all updateable columns of the table or view identified in the first FROM clause of the fullselect are included.

The FOR UPDATE OF clause must not be specified if the result table of the fullselect is read-only (for more information see “DECLARE CURSOR” on page 576), if the FOR READ ONLY clause is used, or if the SCROLL keyword is specified without the DYNAMIC keyword on the DECLARE CURSOR statement.

Positioned UPDATE statements identifying the cursor associated with a *select-statement* can update all updateable columns, if:

- The select-statement does not contain one of the following:
 - An UPDATE clause
 - A FOR READ ONLY clause
 - An ORDER BY clause
- The DECLARE CURSOR statement does not contain a SCROLL keyword without the DYNAMIC keyword.

read-only-clause

►►—FOR READ ONLY—◄◄

The FOR READ ONLY or FOR FETCH ONLY clause indicates that the result table is read-only and therefore the cursor cannot be used for Positioned UPDATE and DELETE statements.

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.

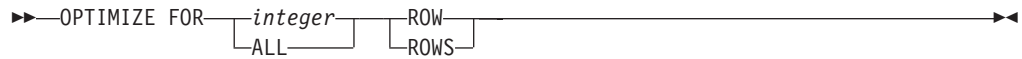
For result tables in which updates and deletes are allowed, specifying FOR READ ONLY can possibly improve the performance of FETCH operations by allowing the database manager to do blocking and avoid exclusive locks. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors that have not specified SCROLL without the DYNAMIC keyword as if the UPDATE clause was specified.

A read-only result table must not be referred to in an UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY.

Syntax Alternatives: FOR FETCH ONLY can be specified in place of FOR READ ONLY.

optimize-clause

optimize-clause



The *optimize-clause* tells the database manager to assume that the program does not intend to retrieve more than *integer* rows from the result table. Without this clause, or with the keyword `ALL`, the database manager assumes that all rows of the result table are to be retrieved. Optimizing for *integer* rows can improve performance. The database manager will optimize the query based on the specified number of rows.

The clause does not change the result table or the order in which the rows are fetched. Any number of rows can be fetched, but performance can possibly degrade after *integer* fetches.

The value of *integer* must be a positive integer (not zero).

isolation-clause



The *isolation-clause* specifies an isolation level at which the select statement is executed.

- RR - Repeatable Read
- RS - Read Stability
- CS - Cursor Stability

The *KEEP LOCKS* clause specifies that any read locks acquired will be held for a longer duration. Normally, read locks are released when the next row is read. If the isolation clause is associated with a cursor, the locks will be held until the cursor is closed or until a *COMMIT* or *ROLLBACK* statement is executed.

Otherwise, the locks will be held until the completion of the SQL statement. The *KEEP LOCKS* clause is only allowed on an SQL *SELECT*, *SELECT INTO*, or *DECLARE CURSOR* statement. It is not allowed on updateable cursors.

- UR - Uncommitted Read
- NC - No Commit

If *isolation-clause* is not specified, the default isolation is used with the exception of a default isolation level of uncommitted read. See “Isolation Level” on page 21 for a description of how the default is determined.

Keyword Synonyms: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword *NONE* can be used as a synonym for *NC*.
- The keyword *CHG* can be used as a synonym for *UR*.
- The keyword *ALL* can be used as a synonym for *RS*.

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 AVGSAL
```

Example 4

Declare a cursor named UP_CUR, to be used in a C program, that updates 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. The declaration specifies that the access path for the query be optimized for the retrieval of a maximum of 2 rows. Even so, the program can retrieve more than 2 rows from the result table. However, when more than 2 rows are retrieved, performance could possibly degrade.

```
EXEC SQL DECLARE UP_CUR CURSOR FOR
SELECT PROJNO, PRSTDATE, PRENDATE
FROM PROJECT
FOR UPDATE OF PRSTDATE, PRENDATE
OPTIMIZE FOR 2 ROWS ;
```

Example 5

Select items from a table with an isolation level of Repeatable Read (RS).

```
SELECT NAME, SALARY
FROM PAYROLL
WHERE DEPT = 704
WITH RS
```

Example 6

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 7

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. 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
```

isolation-clause

Chapter 5. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements listed in the following table.

Table 33. SQL Schema Statements

SQL Statement	Description	Page
ALTER TABLE	Alters the description of a table	370
COMMENT	Replaces or adds a comment to the description of an alias, column, function, index, package, parameter, procedure, table, type or view	404
CREATE ALIAS	Creates an alias	425
CREATE DISTINCT TYPE	Creates a distinct type	428
CREATE FUNCTION	Creates a user-defined function	435
CREATE FUNCTION (External Scalar)	Creates an external scalar function	439
CREATE FUNCTION (External Table)	Creates an external table function	455
CREATE FUNCTION (Sourced)	Creates a user-defined function based on another existing scalar or column function	469
CREATE FUNCTION (SQL Scalar)	Creates an SQL scalar function	478
CREATE FUNCTION (SQL Table)	Creates an SQL table function	487
CREATE INDEX	Creates an index on a table	496
CREATE PROCEDURE	Creates a procedure.	500
CREATE PROCEDURE (External)	Creates an external procedure.	501
CREATE PROCEDURE (SQL)	Creates an SQL procedure.	512
CREATE SCHEMA	Creates a schema and a set of objects in that schema	521
CREATE TABLE	Creates a table	525
CREATE TRIGGER	Creates a trigger	556
CREATE VIEW	Creates a view of one or more tables or views	569
DROP	Drops an alias, function, index, package, procedure, schema, table, trigger, type, or view	629
GRANT (Distinct Type Privileges)	Grants privileges on a distinct type	652
GRANT (Function or Procedure Privileges)	Grants privileges on a function or procedure	655
GRANT (Package Privileges)	Grants privileges on a package	662
GRANT (Table Privileges)	Grants privileges on a table or view	664

Statements

Table 33. SQL Schema Statements (continued)

SQL Statement	Description	Page
LABEL	Replaces or adds a label on the description of an alias, column, package, table, or view	681
RENAME	Renames a table, view, or index.	703
REVOKE (Distinct Type Privileges)	Revokes the privilege to use a distinct type	706
REVOKE (Function or Procedure Privileges)	Revokes privileges on a function or procedure	708
REVOKE (Package Privileges)	Revokes the privilege to execute statements in a package	715
REVOKE (Table Privileges)	Revokes privileges on a table or view	717

Table 34. SQL Data Change Statements

SQL Statement	Description	Page
DELETE	Deletes one or more rows from a table	612
INSERT	Inserts one or more rows into a table	673
UPDATE	Updates the values of one or more columns in one or more rows of a table	762

Table 35. SQL Data Statements

SQL Statement	Description	Page
	All SQL Data Change statements	Table 34
CLOSE	Closes a cursor	402
DECLARE CURSOR	Defines an SQL cursor	576
FETCH	Positions a cursor on a row of the result table; can also assign values from one or more rows of the result table to host variables	645
FREE LOCATOR	Removes the association between a LOB locator variable and its value	651
HOLD LOCATOR	Allows a LOB locator variable to retain its association with a value beyond a unit of work	669
LOCK TABLE	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table	684
OPEN	Opens a cursor	686
SELECT	Executes a query	726
SELECT INTO	Assigns values to host variables	727
SET transition-variable	Assigns values to a transition variable	758
SET variable	Assigns values to a host variable	760
VALUES	Provides a method to invoke a user-defined function from a trigger.	771
VALUES INTO	Specifies a result table of no more than one row and assigns the values to host variables.	773

Table 36. SQL Transaction Statements

SQL Statement	Description	Page
COMMIT	Ends a unit of work and commits the database changes made by that unit of work	413
RELEASE SAVEPOINT	Releases a savepoint within a unit of work	702
ROLLBACK	Ends a unit of work and backs out the database changes made by that unit of work	720
SAVEPOINT	Sets a savepoint within a unit of work	724
SET TRANSACTION	Changes the isolation level for the current unit of work	755

Table 37. SQL Connection Statements

SQL Statement	Description	Page
CONNECT (Type 1)	Connects to a server and establishes the rules for remote unit of work	416
CONNECT (Type 2)	Connects to a server and establishes the rules for application-directed distributed unit of work	421
DISCONNECT	Immediately ends one or more connections	627
RELEASE (Connection)	Places one or more connections in the release-pending state	700
SET CONNECTION	Establishes the server of the process by identifying one of its existing connections	730

Table 38. SQL Dynamic Statements

SQL Statement	Description	Page
DESCRIBE	Describes the result columns of a prepared statement	618
DESCRIBE TABLE	Obtains information about a table or view	623
EXECUTE	Executes a prepared SQL statement	640
EXECUTE IMMEDIATE	Prepares and executes an SQL statement	643
PREPARE	Prepares an SQL statement for execution	691

Table 39. SQL Session Statements

SQL Statement	Description	Page
DECLARE GLOBAL TEMPORARY TABLE	Defines a declared global temporary table	583
SET PATH	Assigns a value to the CURRENT PATH special register	747
SET SCHEMA	Assigns a value to the CURRENT SCHEMA special register	753

Table 40. SQL Embedded Host Language Statements

SQL Statement	Description	Page
BEGIN DECLARE SECTION	Marks the beginning of an SQL declare section	394

Statements

Table 40. SQL Embedded Host Language Statements (continued)

SQL Statement	Description	Page
DECLARE PROCEDURE	Defines an external procedure	598
DECLARE STATEMENT	Declares the names used to identify prepared SQL statements	607
DECLARE VARIABLE	Declares a subtype or CCSID other than the default for a host variable	609
END DECLARE SECTION	Marks the end of an SQL declare section	639
INCLUDE	Inserts declarations into a source program	671
SET RESULT SET	Identifies the result sets in a procedure	750
SET OPTION	Establishes the options for processing SQL statements	733
WHENEVER	Defines actions to be taken on the basis of SQL return codes	776

Table 41. SQL Control Statements

SQL Statement	Description	Page
assignment-statement	Assigns a value to an output parameter or to a local variable	784
CALL	Calls a procedure	396
CASE	Selects an execution path based on multiple conditions	788
compound-statement	Groups other statements together in an SQL routine	790
FOR	Executes a statement for each row of a table	797
GET DIAGNOSTICS	Obtains information about the previous SQL statement that was executed	799
GOTO	Branches to a user-defined label within an SQL routine or trigger	802
IF	Provides conditional execution based on the truth value of a condition	804
ITERATE	Causes the flow of control to return to the beginning of a labelled loop	806
LEAVE	Continues execution by leaving a block or loop	807
LOOP	Repeats the execution of a statement	808
REPEAT	Repeats the execution of a statement	809
RESIGNAL	Resignals an error or warning condition	811
RETURN	Returns from a routine	814
SIGNAL	Signals an error or warning condition	816
WHILE	Repeats the execution of a statement while a specified condition is true	819

See also:

- “How SQL Statements Are Invoked” on page 365
- “SQL Return Codes” on page 367

- “SQL Comments” on page 368

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 any of the following ways:

- Embedded in an application program
- Dynamically prepared and executed
- Issued interactively

Note: Statements embedded in REXX or processed using RUNSQLSTM are prepared and executed dynamically.

Depending on the statement, you can use some or all of these methods. The *Invocation* section in the description of each statement tells you which methods can be used.

A *nonexecutable statement* can only be embedded in an application program.

Embedding a Statement in an Application Program

SQL statements can be included in a source program that will be submitted to the precompiler by using the CRTSQLCBL, CRTSQLCBLLI, CRTSQLCI, CRTSQLFTN, CRTSQLCPPI, CRTSQLPLI, CRTSQLRPG, or CRTSQLRPGI commands. 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 a keyword (or keywords) to indicate that the statement is an SQL statement:

- In C, COBOL FORTRAN, PL/I, and RPG, each embedded statement must be preceded by the keywords EXEC and SQL.
- In Java, each embedded statement must be preceded by the keywords #sql.
- In REXX, each embedded statement must be preceded by the keyword EXECSQL.

Executable statements

An executable statement embedded in an application program is executed every time a statement of the host language would be executed if specified in the same place. This means that 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.

Statements

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.

Objects referenced in SQL statements need not exist when the statements are prepared.

Nonexecutable statements

An embedded nonexecutable statement is processed only by the precompiler. The precompiler reports any errors encountered in such a statement. The statement is *never* executed, and acts as a no-operation if placed among executable statements of the application program. 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 work station). The statement can be prepared for execution using the (embedded) statement `PREPARE` and executed by the (embedded) statement `EXECUTE`. Alternatively, the (embedded) statement `EXECUTE IMMEDIATE` can be used to prepare and execute a statement in one step. In Java, the statement can be prepared for execution by means of the `Statement`, `PreparedStatement`, and `CallableStatement` classes, and executed by means of their respective `execute()` methods.

A statement that is dynamically prepared must not contain references to host variables. It can contain parameter markers instead. See “`PREPARE`” on page 691 for rules concerning the parameter markers. When the prepared statement is executed, the parameter markers are effectively replaced by the current values of the host variables specified in the `EXECUTE` statement. See “`EXECUTE`” on page 640 for rules concerning this replacement. After a statement is prepared, it can be executed several times with different values of host variables. Parameter markers are not allowed in `EXECUTE IMMEDIATE`.

In C, COBOL FORTRAN, PL/I, REXX, and RPG, 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 for embedded statements. See the topic “SQL Return Codes” on page 367 for more information. In Java, the successful or unsuccessful execution of the statement is handled by Java Exceptions.

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 or multiple rows at a time by using the multiple-row `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 work station). 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 or multiple rows at a time by using the multiple-row FETCH statement.

Used in this way, the *select-statement* must not contain references to host variables. It can instead contain parameter markers. See "PREPARE" on page 691 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 686 for rules concerning this replacement.

Interactive Invocation

A capability for entering SQL statements from a work station is part of the architecture of the database manager. The DB2 UDB for iSeries licensed program provides the Start Structured Query Language (STRSQL) command, the Start Query Manager (STRQM) command, and the SQL Script support of iSeries Navigator for this facility. Other products are also available. A statement entered in this way is said to be issued interactively. A statement that cannot be dynamically prepared cannot be issued interactively, with the exception of connection management statements (CONNECT, DISCONNECT, RELEASE, and SET CONNECTION).

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

Each host language provides a mechanism for handling SQL return codes:

- In C, COBOL, FORTRAN, and PL/I, an application program containing executable SQL statements must provide at least one of the following:
 - A structure named SQLCA.
 - A stand-alone CHAR(5) (CHAR(6) in C) variable named SQLSTATE (SQLSTA or SQLSTATE in FORTRAN).
 - A stand-alone integer variable named SQLCODE (SQLCOD in FORTRAN).

A stand-alone SQLSTATE or SQLCODE must not be declared in a host structure. Both a stand-alone SQLSTATE and SQLCODE may be provided.

An SQLCA can be obtained by using the INCLUDE SQLCA statement. If an SQLCA is provided, neither a stand-alone SQLSTATE or SQLCODE can be provided. The SQLCA includes a character-string variable named SQLSTATE (SQLSTT in FORTRAN) and an integer variable named SQLCODE (SQLCOD in FORTRAN) .

A stand-alone SQLSTATE should be used to conform with the SQL 1999 Core standard.

- In Java, for error conditions, the getSQLState method can be used to get the SQLSTATE and the getErrorCode method can be used to get the SQLCODE.
- In REXX and RPG, an SQLCA is provided automatically.

Statements

SQLSTATE

Regardless of whether the application program provides an SQLCA or a stand-alone variable, 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 scheme is the same for all database managers and is based on the proposed ISO/ANSI standard. A complete list of SQLSTATE classes and SQLSTATEs associated with each SQLCODE is supplied in the SQL Messages and Codes book in the iSeries Information Center.

SQLCODE

The SQLCODE is also set by the database manager after each SQL statement is executed 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.

A complete listing of DB2 UDB for iSeries SQLCODEs and their corresponding SQLSTATEs is provided in the SQL Messages and Codes book in the iSeries Information Center.

SQL Comments

Static SQL statements can include host language or SQL comments. Dynamic SQL statements can include SQL comments. There are two types of SQL comments:

simple comments

Simple comments are introduced by two consecutive hyphens.

bracketed comments

Bracketed comments are introduced by /* and end with */.

These rules apply to the use of simple comments:

- The two hyphens must be on the same line and must not be separated by a space.
- Simple comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Simple comments cannot be continued to the next line.
- In COBOL, the hyphens must be preceded by a space.

These rules apply to the use of bracketed comments:

- The /* must be on the same line and not separated by a space.
- The */ must be on the same line and not separated by a space.
- Bracketed comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').

- Bracketed comments can be continued to the next line.
- Bracketed comments can be nested within other bracketed comments.

Example 1: This example shows how to include simple comments in a statement:

```
CREATE VIEW PRJ_MAXPER      -- PROJECTS WITH MOST SUPPORT PERSONNEL
AS SELECT PROJNO, PROJNAME -- NUMBER AND NAME OF PROJECT
FROM PROJECT
WHERE DEPTNO = 'E21'      -- SYSTEMS SUPPORT DEPT CODE
AND PRSTAFF > 1
```

Example 2: This example shows how to include bracketed comments in a statement:

```
CREATE VIEW PRJ_MAXPER      /* PROJECTS WITH MOST SUPPORT
                             PERSONNEL */
AS SELECT PROJNO, PROJNAME /* NUMBER AND NAME OF PROJECT */
FROM PROJECT
WHERE DEPTNO = 'E21'      /* SYSTEMS SUPPORT DEPT CODE */
AND PRSTAFF > 1
```

ALTER TABLE

The ALTER TABLE statement alters the definition of a table.

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 of the statement must include at least one of the following:

- For the table identified in the statement,
 - The ALTER privilege on the table, and
 - The system authority *EXECUTE on the library containing the table
- Administrative authority

The authorization ID of the statement has the ALTER privilege on the table when one of the following is true:

- It is the owner of the table.
- It was granted the ALTER privilege to the table.
- It was granted the system authorities of either *OBJALTER or *OBJMGT to the table.

To define a foreign key, the privileges held by the authorization ID of the statement must include at least one of the following on the parent table:

- The REFERENCES privilege or object management authority for the table
- The REFERENCES privilege on each column of the specified parent key
- Ownership of the table
- Administrative authority

The authorization ID of the statement has the REFERENCES privilege on a table when one of the following is true:

- It is the owner of the table.
- It was granted the REFERENCES privilege to the table.
- It was granted the system authorities of either *OBJREF or *OBJMGT to the table.

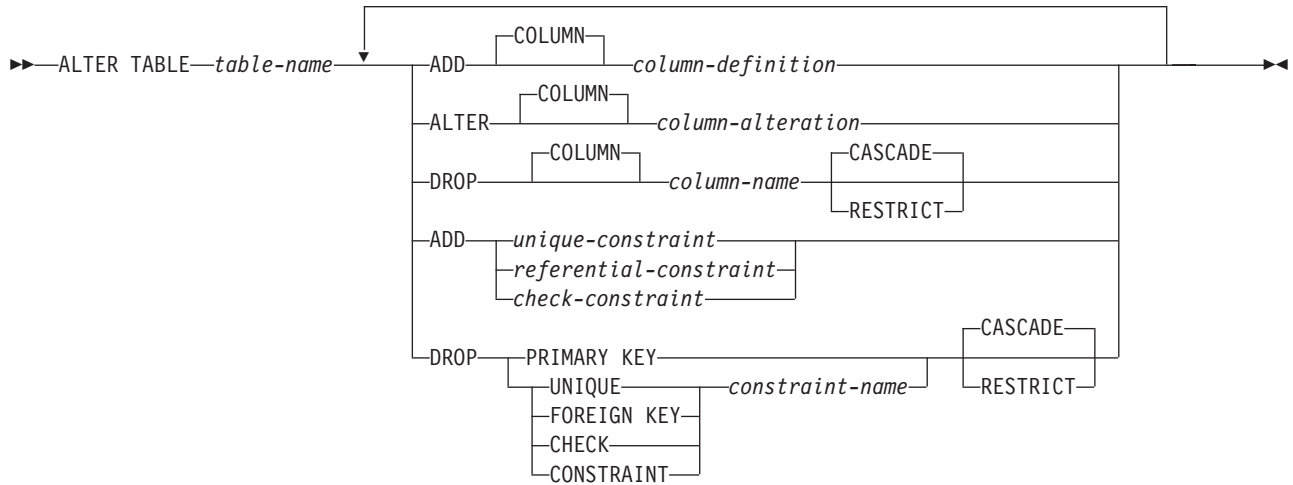
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

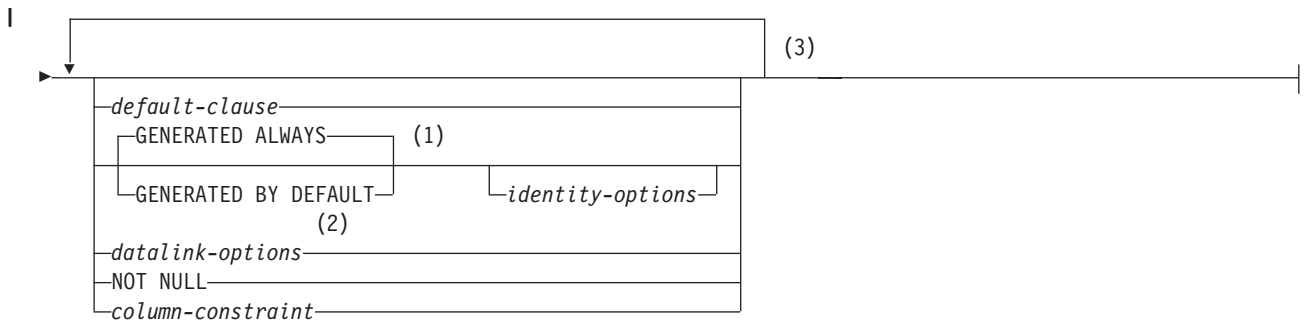
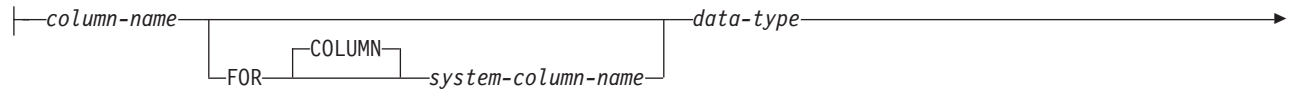
The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

Syntax



column-definition:



Notes:

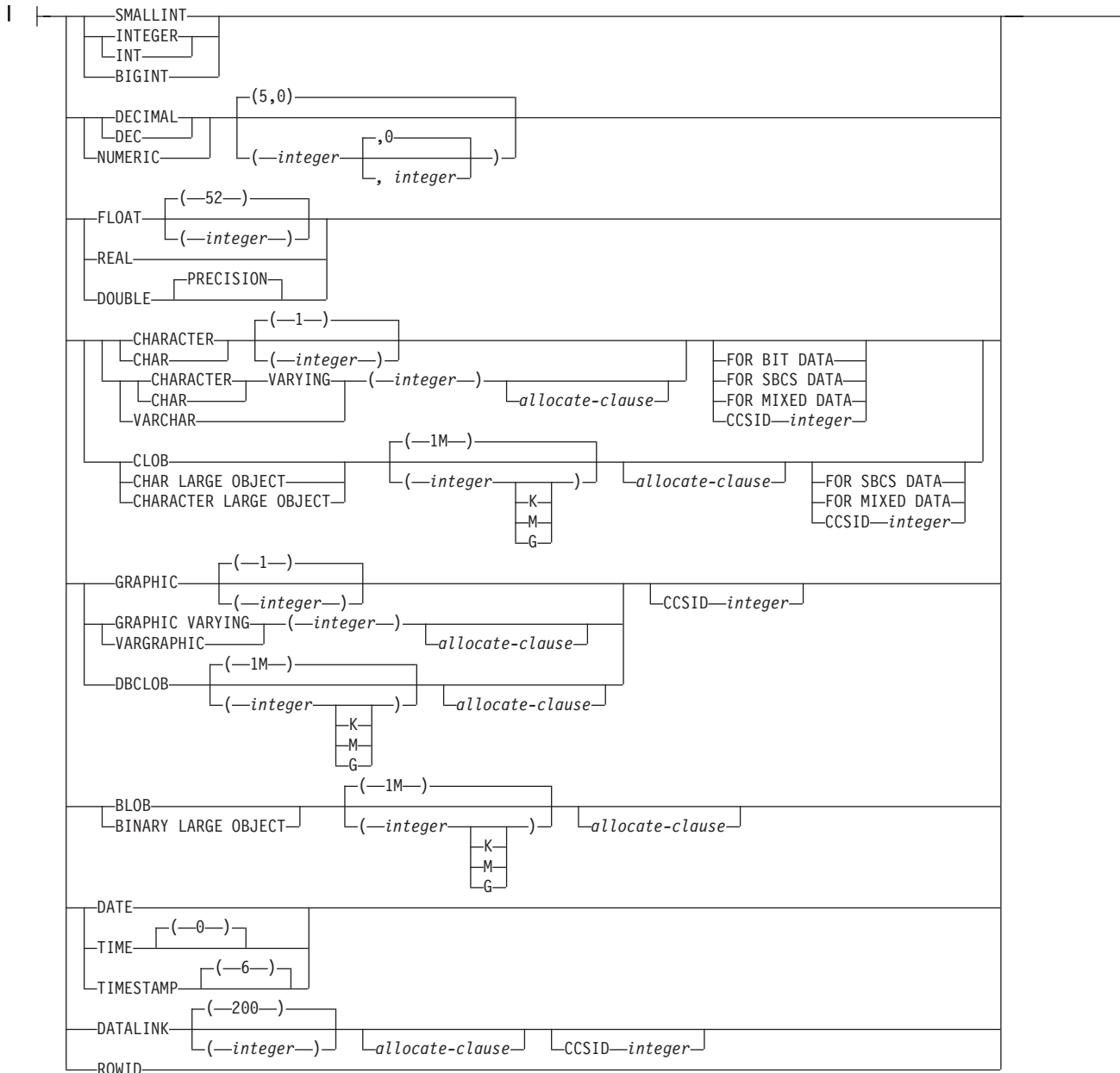
- 1 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), or the column is an identity column.
- 2 The datalink-options can only be specified for DATALINKs and distinct-types sourced on DATALINKs.
- 3 The same clause must not be specified more than once.

ALTER TABLE

data-type:

| *built-in-type* |
 | *distinct-type-name* |

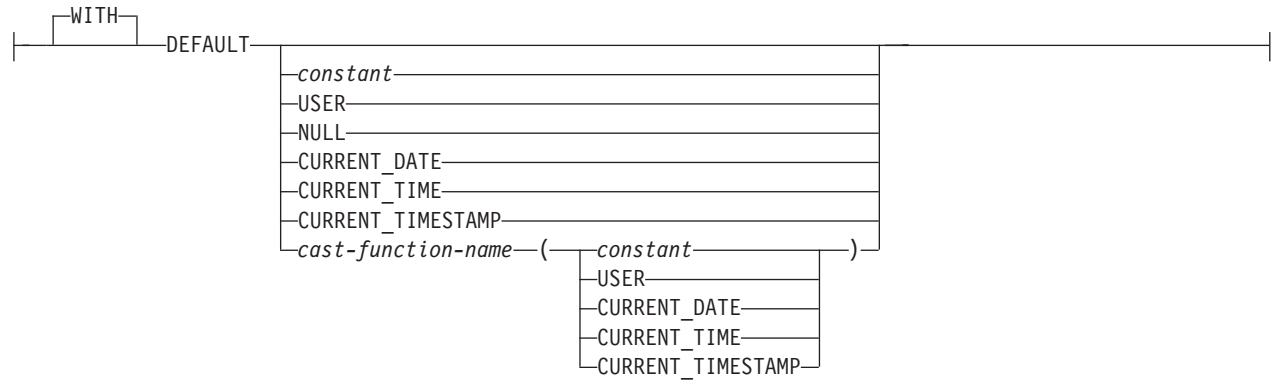
built-in-type:



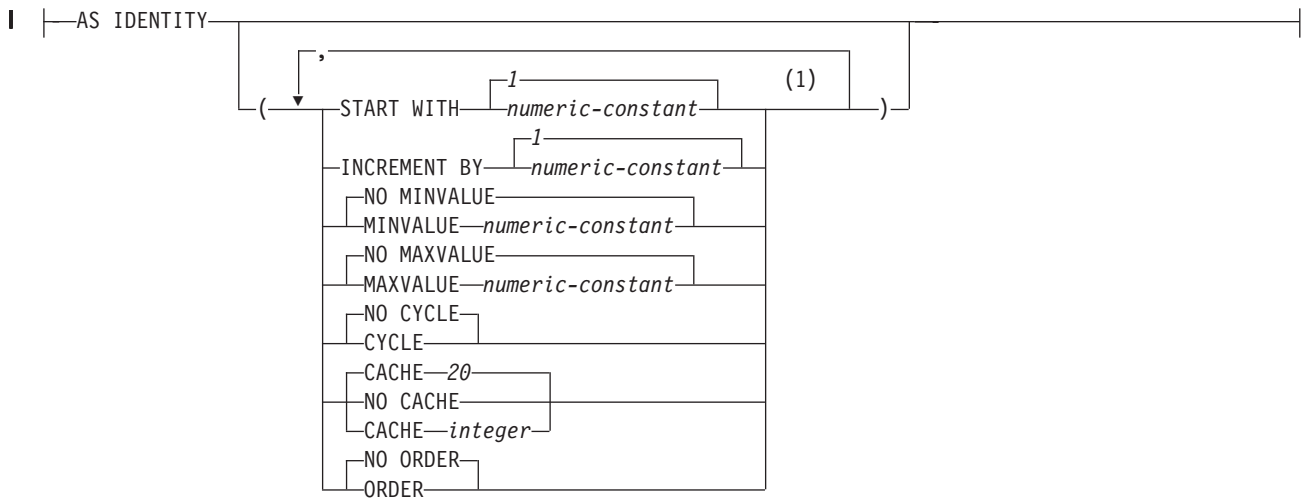
allocate-clause:

| ALLOCATE $(integer)$ |

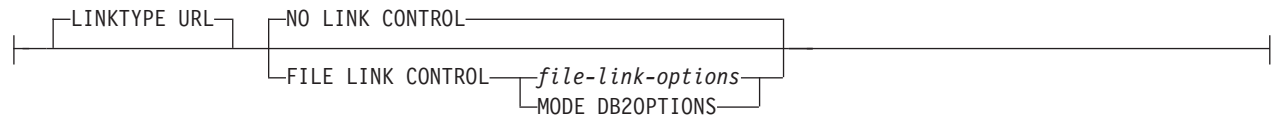
default-clause:



identity-options:



datalink-options:

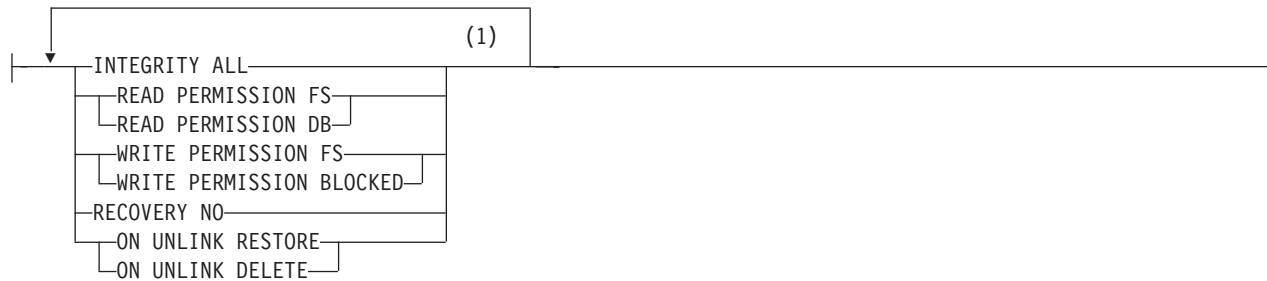


Notes:

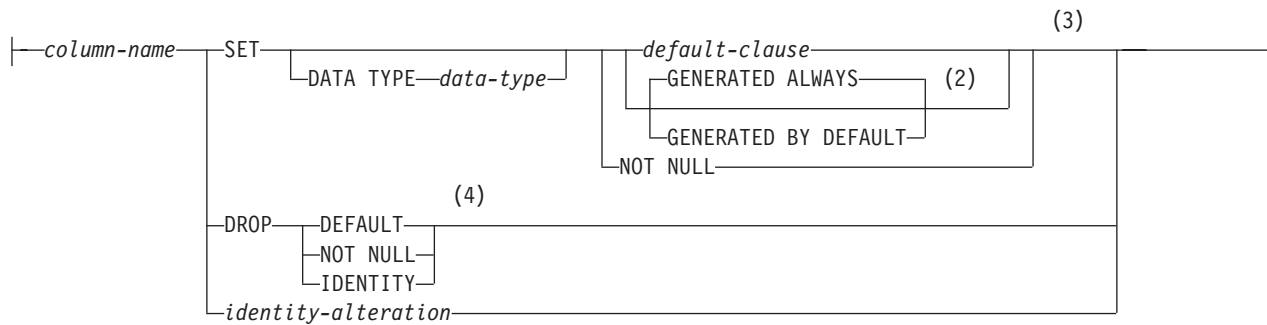
- 1 Each clause may be specified only once.

ALTER TABLE

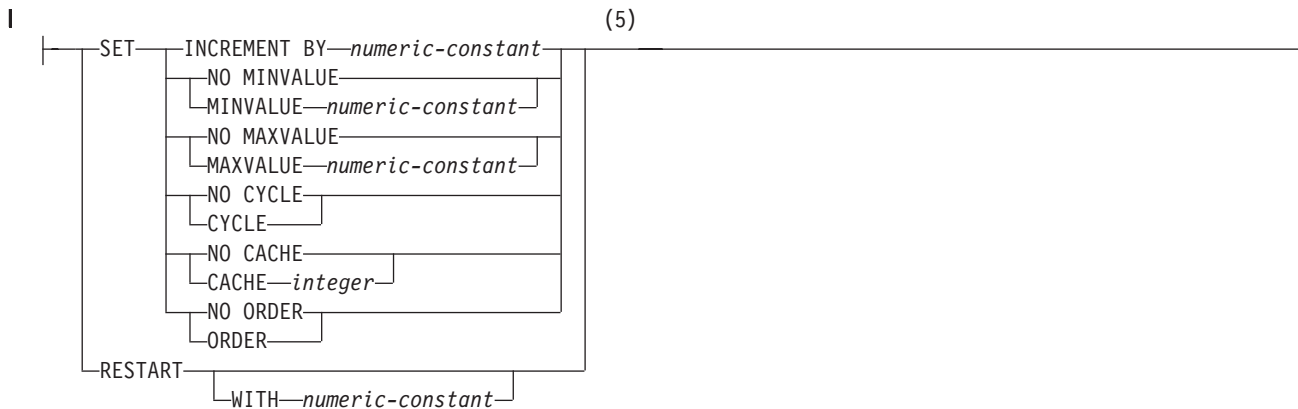
file-link-options:



column-alteration:

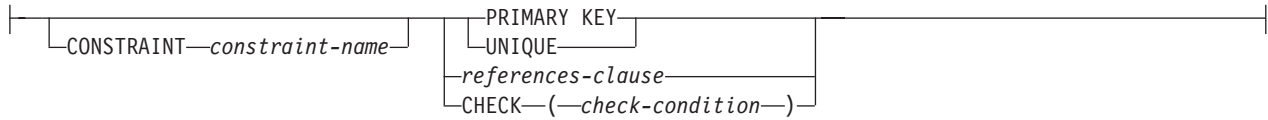
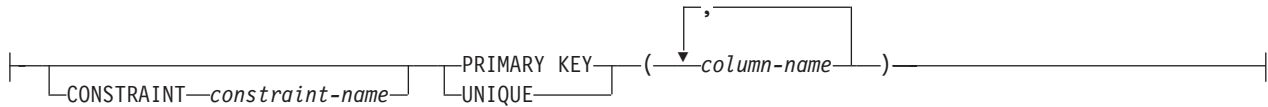
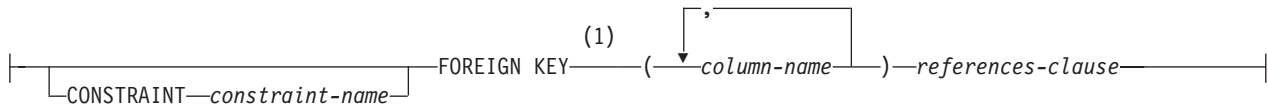
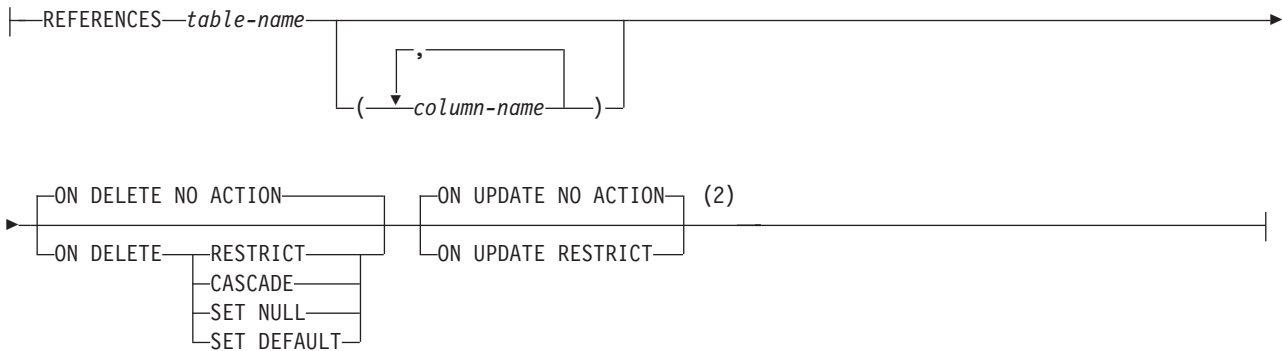
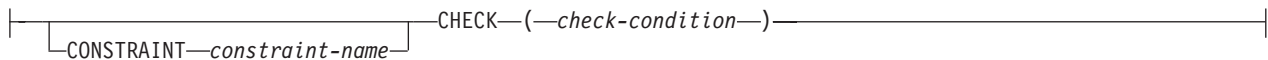


identity-alteration:



Notes:

- 1 All five *file-link-options* must be specified, but they can be specified in any order.
- 2 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), or the column is an identity column.
- 3 Each clause may be specified only once.
- 4 Each clause may be specified only once.
- 5 Each clause may be specified only once.

column-constraint:**unique-constraint:****referential-constraint:****references-clause:****check-constraint:****Notes:**

- 1 For compatibility with other products, constraint-name (without the CONSTRAINT keyword) may be specified following FOREIGN KEY.
- 2 The ON DELETE and ON UPDATE clauses may be specified in either order.

ALTER TABLE

Description

table-name

Identifies the table you want to be altered. The *table-name* must identify a table that exists at the current server. It must not be a view, a catalog table, or a global temporary table.

ADD *column-definition*

Adds a column to the table. If the table has rows, every value of the column is set to its default value, unless the column is a ROWID column or an identity column (a column that is defined AS IDENTITY). The database manager generates default values for ROWID columns and identity columns. If the table previously had n columns, the ordinality of the new column is $n+1$. The value of $n+1$ must not exceed 8000.

A table can have only one ROWID column. You cannot add an identity column to a table that already has an identity column.

A DataLink column with FILE LINK CONTROL cannot be added to a table that is a dependent in a referential constraint with a delete rule of CASCADE.

Adding a new column must not make the sum of the row buffer byte counts of the columns be greater than 32766 or, if a VARCHAR or VARGRAPHIC column is specified, 32740. Additionally, if a LOB is specified, the sum of the byte counts of the columns must not be greater than 3 758 096 383 at the time of insert or update. For information on the byte counts of columns according to data type, see “Notes” on page 550.

column-name

Names the column to be added to the table. Do not use the same name for more than one column of the table or for a system-column-name of the table. Do not qualify *column-name*.

FOR COLUMN *system-column-name*

Provides an OS/400 name for the column. Do not use the same name for more than one column-name or system-column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see “Rules for Column Name Generation” on page 553.

data-type

Specifies the data type of the column. The data type can be a built-in data type or a distinct type.

built-in-type

Specifies a built-in data type. See “CREATE TABLE” on page 525 for a description of built-in types.

distinct-type-name

Specifies the data type of a column is a distinct type. The length, precision and scale of the column are respectively the length, precision, and scale of the source type of the 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. If the column is to be used in the definition of the foreign key of a referential constraint, the data type of the corresponding column of the parent key must have the same distinct type.

DEFAULT

Specifies a default value for the column. This clause cannot be specified more than once in the same *column-definition*. DEFAULT cannot be specified for a ROWID column or an identity column (a column that is defined AS IDENTITY). The database manager generates default values for ROWID columns and identity columns. If a value is not specified following the DEFAULT keyword, then:

- if the column is nullable, the default value is the null value.
- if the column is not nullable, the default depends on the data type of the column:

Data type	Default value
Numeric	0
Fixed-length string	Blanks
Varying-length string	A string length of 0
Date	For existing rows, a date corresponding to 1 January 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 1 January 0001 and a time corresponding to 0 hours, 0 minutes, 0 seconds, and 0 microseconds. For added rows, the current timestamp.
Datalink	A value corresponding to DLVALUE('','URL','').
Distinct type	The default value of the corresponding source type of the distinct type.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

constant

Specifies the constant as the default 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 “Assignments and Comparisons” on page 80. A floating-point constant must not be used for a SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC column. A decimal constant must not contain more digits to the right of the decimal point than the specified scale of the column.

USER

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the column must be CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the USER special register. For existing rows, the value is that of the USER special register at the time the ALTER TABLE statement is processed.

ALTER TABLE

NULL

Specifies null as the default for the column. If NOT NULL is specified, DEFAULT NULL must not be specified within the same *column-definition*.

CURRENT_DATE

Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the column must be DATE or a distinct type based on a DATE.

CURRENT_TIME

Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the column must be TIME or a distinct type based on a TIME.

CURRENT_TIMESTAMP

Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the column must be TIMESTAMP or a distinct type based on a TIMESTAMP.

cast-function-name

This form of a default value can only be used with columns defined as a distinct type, BLOB, CLOB, DBCLOB, DATE, TIME, or TIMESTAMP data types. The following table describes the allowed uses of these *cast-functions*.

Data Type	Cast Function Name
Distinct type N based on a BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
Distinct type N based on a DATE, TIME, or TIMESTAMP	N (the user-defined cast function that was generated when N was created) ** or DATE, TIME, or TIMESTAMP *
Distinct type N based on other data types	N (the user-defined cast function that was generated when N was created) **
BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *
Notes:	
* The name of the function must match the name of the data type (or the source type of the distinct type) with an implicit or explicit schema name of QSYS2.	
** 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.	

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. For BLOB, CLOB, DBCLOB, DATE, TIME, and TIMESTAMP functions, the constant must be a string constant.

USER

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the source type of the distinct type of the column must be CHAR or VARCHAR with a length attribute that is greater than or equal to the

length attribute of USER. For existing rows, the value is that of the USER special register at the time the ALTER TABLE statement is processed.

CURRENT_DATE

Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the source type of the distinct type of the column must be DATE.

CURRENT_TIME

Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the source type of the distinct type of the column must be TIME.

CURRENT_TIMESTAMP

Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the source type of the distinct type of the column must be TIMESTAMP.

If the value specified is not valid, an error is returned.

GENERATED

Specifies that the database manager generates values for the column. GENERATED must be specified if the column is to be considered an identity column (defined with the AS IDENTITY clause). It may also be specified if the data type of the column is a ROWID (or a distinct type that is based on a ROWID). Otherwise, it must not be specified.

ALWAYS

Specifies that the database manager will always generate a value for the column when a row is inserted into the table. ALWAYS is the recommended value.

BY DEFAULT

Specifies that the database manager will generate a value for the column when a row is inserted only if a value is not specified for the column. If a value is specified, the database manager uses that value.

For a ROWID column, the database manager uses a specified value, but it must be a valid unique row ID value that was previously generated by DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries.

For an identity column, the database manager inserts a specified value but does not verify that it is a unique value for the column unless the identity column has a unique constraint or a unique index that solely specifies the identity column.

AS IDENTITY

Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC with a scale of zero, or a distinct type based on one of these types).

An identity column is implicitly NOT NULL.

START WITH *numeric-constant*

Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

ALTER TABLE

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending sequence and the MAXVALUE for a descending sequence. This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value that is not 0, does not exceed the value of a large integer constant, and could be assigned to the column without any non-zero digits existing to the right of the decimal point. The default is 1.

If the value is positive, the sequence of values for the identity column ascends. If the value is negative, the sequence of values descends.

MAXVALUE *numeric-constant*

Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be greater than the minimum value.

If a value is not explicitly specified when the identity column is defined, this is the maximum value of the data type (and precision, if DECIMAL) for an ascending sequence; or the START WITH value, or -1 if START WITH was not specified, for a descending sequence.

MINVALUE *numeric-constant*

Specifies the numeric constant that is the minimum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be less than the maximum value.

If a value is not explicitly specified when the identity column is defined, this is the START WITH value, or 1 if START WITH was not specified, for an ascending sequence; or the minimum value of the data type (and precision, if DECIMAL) for a descending sequence.

CACHE or NO CACHE

Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table.

CACHE *integer*

Specifies the number of values of the identity column sequence that the database manager preallocates and keeps in memory. The minimum value that can be specified is 2, and the maximum is the largest value that can be represented as an integer. The default is 20.

During a system failure, all cached identity column values that are yet to be assigned are lost, and thus, will never be used. Therefore, the value specified for CACHE also represents the maximum number of values for the identity column that could be lost during a system failure.

NO CACHE

Specifies that values for the identity column are not preallocated.

CYCLE or NO CYCLE

Specifies whether this identity column should continue to generate values after reaching either the maximum or minimum value of the sequence.

CYCLE

Specifies that values continue to be generated for this column after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches the maximum value of the sequence, it generates its minimum value. After a descending sequence reaches its minimum value of the sequence, it generates its maximum value. The maximum and minimum values for the column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by the database manager for an identity column. If a unique constraint or unique index exists on the identity column, and a non-unique value is generated for it, an error occurs.

NO CYCLE

Specifies that values will not be generated for the identity column once the maximum or minimum value for the sequence has been reached. This is the default.

ORDER or NO ORDER

Specifies whether the identity values must be generated in order of request.

ORDER

Specifies that the values are generated in order of request.

NO ORDER

Specifies that the values do not need to be generated in order of request. This is the default.

datalink-options

Specifies the options associated with a DATALINK column. See "CREATE TABLE" on page 525 for a description of *datalink-options*.

NOT NULL

Prevents the column from containing null values. Omission of NOT NULL implies that the column can contain null values. If NOT NULL is specified in the column definition, then DEFAULT must also be specified.

column-constraint

The *column-constraint* of a *column-definition* provides a shorthand method of defining a constraint composed of a single column. Thus, if a *column-constraint* is specified in the definition of column C, the effect is the same as if that constraint were specified as a *unique-constraint*, *referential-constraint* or *check-constraint* in which C is the only identified column.

CONSTRAINT *constraint-name*

Names the constraint. A constraint-name must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

PRIMARY KEY

Provides a shorthand method of defining a primary key composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

ALTER TABLE

This clause must not be specified in more than one *column-definition* and must not be specified at all if the UNIQUE clause is specified in the column definition. The column must not be a LOB or DataLink column.

When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in the column that makes up the primary key.

UNIQUE

Provides a shorthand method of defining a unique key composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE (C) clause is specified as a separate clause.

This clause cannot be specified more than once in a column definition and must not be specified if PRIMARY KEY is specified in the *column-definition*. The column must not be a LOB or DataLink column.

references-clause

The *references-clause* of a column-definition provides a shorthand method of defining a foreign key composed of a single column. Thus, if a references-clause is specified in the definition of column C, the effect is the same as if that references-clause were specified as part of a FOREIGN KEY clause in which C is the only identified column.

CHECK(*check-condition*)

Provides a shorthand method of defining a check constraint whose *check-condition* only references a single column. Thus, if CHECK is specified in the column definition of column C, no columns other than C can be referenced in the *check-condition* of the check constraint. The effect is the same as if the check constraint were specified as a separate clause.

ROWID or DATALINK with FILE LINK CONTROL columns cannot be referenced in a CHECK constraint. For additional restrictions see, "ADD check-constraint" on page 387.

ALTER *column-alteration*

Alters the definition of a column. Only the attributes specified will be altered. Others will remain unchanged.

column-name

Identifies the column to be altered. The name must not be qualified. The name must identify an existing column of the specified table. The name must not identify a column that is being added or dropped in the same ALTER TABLE statement.

SET DATA TYPE *data-type*

Specifies the new data type of the column to be altered. The new data type must be compatible with the existing data type of the column. For more information about the compatibility of data types see "Assignments and Comparisons" on page 80. However, changing a datetime data type to a character-string data type is not allowed.

The specified length, precision, and scale may be larger, smaller, or the same as the existing length, precision, and scale. However, if the new length, precision, or scale is smaller, truncation or numeric conversion errors may occur.

If the specified column has a default value and a new default value is not specified, the existing default value must represent a value that could be

assigned to the column in accordance with the rules for assignment as described in “Assignments and Comparisons” on page 80.

If the column is specified in a unique, primary, or foreign key, the new sum of the lengths of the columns of the keys must not exceed 2000-n, where n is the number of columns specified that allow nulls.

Changing the attributes will cause any existing values in the column to be converted to the new column attributes according to the rules for assignment to a column, except that string values will be truncated.

SET *default-clause*

Specifies the new default value of the column to be altered. The specified default value must represent a value that could be assigned to the column in accordance with the rules for assignment as described in “Assignments and Comparisons” on page 80.

SET NOT NULL

Specifies that the column cannot contain null values. All values for this column in existing rows of the table must be not null. If the specified column has a default value and a new default value is not specified, the existing default value must not be NULL. SET NOT NULL is not allowed if the column is identified in the foreign key of a referential constraint with a DELETE rule of SET NULL and no other nullable columns exist in the foreign key.

SET GENERATED ALWAYS or GENERATED BY DEFAULT

Specifies that the database manager generates values for the column. GENERATED must be specified if the column is to be considered an identity column (defined with the AS IDENTITY clause) or the data type of the column is a ROWID (or a distinct type that is based on a ROWID). Otherwise, it must not be specified.

DROP DEFAULT

Drops the current default for the column. The specified column must have a default value and must not have NOT NULL as the null attribute. The new default value is the null value.

DROP NOT NULL

Drops the NOT NULL attribute of the column, allowing the column to have the null value. If a default value is not specified or does not already exist, the new default value is the null value. DROP NOT NULL is not allowed if the column is specified in the primary key of the table.

DROP IDENTITY

Drops the identity attributes of the column, making the column a simple numeric data type column. DROP IDENTITY is not allowed if the column is not an identity column.

identity-alteration

Alters the identity attributes of the column. The column must be an identity column. For a description of the attributes, see 379.

RESTART

Specifies the next value for an identity column. If WITH numeric-constant is not specified the sequence is restarted at the value specified implicitly or explicitly as the starting value when the identity column was originally created. The column must be an identity column.

WITH *numeric-constant*

Specifies that *numeric-constant* will be used as the next value for the column. *numeric-constant* must be an exact numeric constant. It can be

ALTER TABLE

any positive or negative value that could be assigned to this column as long as there are no nonzero digits to the right of the decimal point.

DROP COLUMN

Drops the identified column from the table.

column-name

Identifies the column to be dropped. The column name must not be qualified. The name must identify a column of the specified table. The name must not identify a column that was already added or altered in this ALTER TABLE statement. The name must not identify the only column of a table.

CASCADE

Specifies that any views, indexes, triggers, or constraints that are dependent on the column being dropped are also dropped.⁴¹

RESTRICT

Specifies that the column cannot be dropped if any views, indexes, triggers, or constraints are dependent on the column.⁴¹

If all the columns referenced in a constraint are dropped in the same ALTER TABLE statement, RESTRICT does not prevent the drop.

ADD unique-constraint

CONSTRAINT *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that already exists at the current server. The *constraint-name* must be unique within a schema.

If not specified, a unique constraint name is generated by the database manager.

UNIQUE (*column-name,...*)

Defines a unique key composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB or DATALINK column. The number of identified columns must not exceed 120, and the sum of their lengths must not exceed 2000-n, where n is the number of columns specified that allow nulls.

The set of identified columns cannot be the same as the set of columns specified in another UNIQUE constraint or PRIMARY KEY on the table. For example, UNIQUE (A,B) is not allowed if UNIQUE (B,A) or PRIMARY KEY (A,B) already exists on the table. Any existing nonnull values in the set of columns must be unique. Multiple null values are allowed.

If a unique index already exists on the identified columns, that index is designated as a unique constraint index. Otherwise, a unique index is created to support the uniqueness of the unique key. The unique index is created as part of the system physical file, not as a separate system logical file.

PRIMARY KEY (*column-name,...*)

Defines a primary key composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB

41. A trigger is dependent on the column if it is referenced in the UPDATE OF column list or anywhere in the triggered action.

or DATALINK column. The number of identified columns must not exceed 120, and the sum of their lengths must not exceed 2000. The table must not already have a primary key.

The identified columns cannot be the same as the columns specified in another UNIQUE constraint on the table. For example, PRIMARY KEY (A,B) is not allowed if UNIQUE (B,A) already exists on the table. Any existing values in the set of columns must be unique.

When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in any of the columns that make up the primary key.

If a unique index already exists on the identified columns, that index is designated as a primary index. Otherwise, a primary index is created to support the uniqueness of the primary key. The unique index is created as part of the system physical file, not a separate system logical file.

ADD referential-constraint

CONSTRAINT *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that already exists at the current server.

If not specified, a unique constraint name is generated by the database manager.

FOREIGN KEY

Defines a referential constraint.

Let T1 denote the table being altered.

(*column-name*,...)

The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T1. The same column must not be identified more than once. The column must not be a LOB or DATALINK column. The number of the identified columns must not exceed 120, and the sum of their lengths must not exceed 2000-n, where n is the number of columns specified that allows nulls.

REFERENCES *table-name*

The *table-name* specified in a REFERENCES clause must identify a base table that exists at the current server, but it must not identify a catalog table or a global temporary table. This table is referred to as the parent table in the constraint relationship.

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 an existing referential constraint on the table. Duplicate referential constraints are allowed, but not recommended.

Let T2 denote the identified parent table.

(*column-name*,...)

The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than once. The column must not be a LOB or DATALINK column. The number of

ALTER TABLE

identified columns must not exceed 120, and the sum of their lengths must not exceed 2000-n, where n is the number of columns specified that allow nulls.

The list of column names must be identical to the list of column names in the primary key of T2 or a UNIQUE constraint that exists on T2. The names may be specified in any order. For example, if (A,B) is specified, a unique constraint defined as UNIQUE (B,A) would satisfy the requirement. If a column name list is not specified then T2 must have a primary key. Omission of the column name list is an implicit specification of the columns of that primary key.

The specified foreign key must have the same number of columns as the parent key of T2. The description of the *n*th column of the foreign key and the *n*th column of the parent key must have identical data types and lengths.

Unless the table is empty, the values of the foreign key must be validated before the table can be used. Values of the foreign key are validated during the execution of the ALTER TABLE statement. Therefore, every nonnull value of the foreign key must match some value of the parent key of T2.

The referential constraint specified by the FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

ON DELETE

Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are five possible actions:

- NO ACTION (default)
- RESTRICT
- CASCADE
- SET NULL
- SET DEFAULT

SET NULL must not be specified unless some column of the foreign key allows null values. SET NULL and SET DEFAULT must not be specified if T1 has an update trigger.

CASCADE must not be specified if T1 has a delete trigger.

CASCADE must not be specified if T1 contains a DataLink column with FILE LINK CONTROL.

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.
- If SET DEFAULT is specified, each column of the foreign key of each dependent of *p* in T1 is set to its default value.

ON UPDATE

Specifies what action is to take place on the dependent tables when a row of the parent table is updated.

The update rule applies when a row of T2 is the object of an UPDATE or propagated update 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 updated.

ADD check-constraint**CONSTRAINT** *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that already exists at the current server. The *constraint-name* must be unique within a schema.

If not specified, a unique constraint name is generated by the database manager.

CHECK(*check-condition*)

Defines a check constraint. The *check-condition* must be true or unknown for every row of the table.

The *check-condition* is a *search-condition*, except:

- It can only refer to columns of the table and the column names must not be qualified.
- It cannot reference ROWID or DATALINK with FILE LINK CONTROL columns.
- It must not contain any of the following:
 - Subqueries
 - Column functions
 - Host variables
 - Parameter markers
 - Complex expressions that contain LOBs (such as concatenation)
 - CURRENT TIMEZONE, CURRENT SCHEMA, CURRENT SERVER, CURRENT PATH, and USER special registers
 - CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP special registers
 - User-defined functions other than functions that were implicitly generated with the creation of a distinct type
 - NOW, CURDATE, and CURTIME scalar functions
 - NODENAME scalar function
 - ATAN2, DIFFERENCE, RAND, RADIANS, and SOUNDEX scalar functions
 - DLVALUE, DLURLPATH, DLURLPATHONLY, DLURLSERVER, or DLURLSCHEME scalar functions
 - DLURLCOMPLETE scalar function (for DataLinks with an attribute of FILE LINK CONTROL and READ PERMISSION DB)

For more information about search-condition, see “Search Conditions” on page 161.

ALTER TABLE

DROP

PRIMARY KEY

Drops the definition of the primary key and all referential constraints in which the primary key is a parent key. The table must have a primary key.

FOREIGN KEY *constraint-name*

Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint in which the table is a dependent.

UNIQUE *constraint-name*

Drops the unique constraint *constraint-name* and all referential constraints dependent on this unique constraint. The *constraint-name* must identify a unique constraint on the table. DROP UNIQUE will not drop a PRIMARY KEY unique constraint.

CHECK *constraint-name*

Drops the check constraint *constraint-name*. The *constraint-name* must identify a check constraint on the table.

CONSTRAINT *constraint-name*

Drops the constraint *constraint-name*. The *constraint-name* must identify a unique, referential, or check constraint on the table. If the constraint is a PRIMARY KEY or UNIQUE constraint, all referential constraints in which the primary key or unique key is a parent are also dropped.

CASCADE

Specifies for unique constraints that any referential constraints that are dependent on the constraint being dropped are also dropped.

RESTRICT

Specifies for unique constraints that the constraint cannot be dropped if any referential constraints are dependent on the constraint.

Notes

Column References: A column can only be referenced *once* in an ADD, ALTER, or DROP COLUMN clause in a single ALTER TABLE statement. However, that same column can be referenced multiple times for adding or dropping constraints in the same ALTER TABLE statement.

Order of Operations: The order of operations within an ALTER TABLE statement is:

- drop constraints
- drop columns for which the RESTRICT option was specified
- alter all other column definitions
 - drop columns for which the CASCADE option was specified
 - add alter column attributes
 - add columns
- add constraints

Within each of these stages, the order in which the user specifies the clauses is the order in which they are performed, with one exception. If any columns are being dropped, that operation is logically done before any column definitions are added or altered.

QTEMP Considerations: Any views or logical files in another job's QTEMP that are dependent on the table being altered will be dropped as a result of an ALTER TABLE statement.

Authority Checking: Authority checking is performed only on the table being altered. Other objects may be accessed by the ALTER TABLE statement, but no authority to those objects is required. For example, no authority is required on views that exist on the table being altered, nor on dependent tables that reference the table being altered through a referential constraint.

Backup Recommendation: It is strongly recommended that a current backup of the table and dependent views and logical files exist prior to altering a table.

Performance Considerations: The following performance considerations apply to an ALTER TABLE statement when adding, altering, or dropping columns from a table:

- The data in the table may be copied.⁴²

Adding and dropping columns require the data to be copied.

Altering a column usually requires the data to be copied. The data does not need to be copied, however, if the alter only includes the following changes:

- The length attribute of a VARCHAR column is increasing and the current length attribute is greater than 20.
- The length attribute of a VARGRAPHIC column is increasing and the current length attribute is greater than 10.
- The allocated length of a VARCHAR column is changing and the current and new allocated lengths are both less than or equal to 20.
- The allocated length of a VARGRAPHIC column is changing and the current and new allocated lengths are both less than or equal to 10.
- The CCSID of a column is changing but no conversion is necessary between the old and new CCSID. For example, if one CCSID is 65535, no data conversion is necessary.
- The default value is changing, and the length of the default value is not greater than the current allocated length.
- DROP DEFAULT is specified.
- DROP NOT NULL is specified, but at least one nullable column will still exist in the table after the alter table is complete.

- Indexes may need to be rebuilt.⁴³

An index does not need to be rebuilt when columns are added to a table or when columns are dropped or altered and those columns are not referenced in the index key.

Altering a column that is used in the key of an index or constraint usually requires the index to be rebuilt. The index does not need to be rebuilt, however, in the following cases:

- The length attribute of a VARCHAR or VARGRAPHIC key is increasing.
- The CCSID of a column is changing but no conversion is necessary between the old and new CCSID. For example, if one CCSID is 65535.

42. In cases where enough storage does not exist to make a complete copy, a special copy that only requires approximately 16-32 megabytes of free storage is performed.

43. Any indexes that need to be rebuilt are rebuilt asynchronously by database server jobs.

ALTER TABLE

Cascaded Effects

Adding a column has no cascaded effects to SQL views or most logical files.⁴⁴ For example, adding a column to a table does not cause the column to be added to any dependent views, even if those views were created with a SELECT * clause.

Dropping or altering a column may cause several cascaded effects. Table 42 lists the cascaded effects of dropping a column.

Table 42. Cascaded effects of dropping a column

Operation	RESTRICT Effect	CASCADE Effect
Drop of a column referenced by a view	The drop of the column is not allowed.	The view and all views dependent on that view are dropped.
Drop of a column referenced by a non-view logical file	The drop is allowed, and the column is dropped from the logical file if: <ul style="list-style-type: none">• The logical file shares a format with the file being altered, and• The dropped column is not used as a key field or in select/omit specifications, and• That format is not used again in the logical file with another based-on file. Otherwise, the drop of the column is not allowed.	The drop is allowed, and the column is dropped from the logical file if: <ul style="list-style-type: none">• The logical file shares a format with the file being altered, and• The dropped column is not used as a key field or in select or omit specifications, and• That format is not used again in the logical file with another based-on file. Otherwise, the logical file is dropped.
Drop of a column referenced in the key of an index	The drop of the index is not allowed.	The index is dropped.
Drop of a column referenced in a unique constraint	If all the columns referenced in the unique constraint are dropped in the same ALTER COLUMN statement and the unique constraint is not referenced by a referential constraint, the columns and the constraint are dropped. (Hence, the index used to satisfy the constraint is also dropped.) For example, if column A is dropped, and a unique constraint of UNIQUE (A) or PRIMARY KEY (A) exists and no referential constraints reference the unique constraint, the operation is allowed. Otherwise, the drop of the column is not allowed.	The unique constraint is dropped as are any referential constraints that refer to that unique constraint. (Hence, any indexes used by those constraints are also dropped).

44. A column will also be added to a logical file that shares its physical file's format when a column is added to that physical file (unless that format is used again in the logical file with another based-on file).

Table 42. Cascaded effects of dropping a column (continued)

Operation	RESTRICT Effect	CASCADE Effect
Drop of a column referenced in a referential constraint	<p>If all the columns referenced in the referential constraint are dropped at the same time, the columns and the constraint are dropped. (Hence, the index used by the foreign key is also dropped). For example, if column B is dropped and a referential constraint of FOREIGN KEY (A) exists, the operation is allowed.</p> <p>Otherwise, the drop of the column is not allowed.</p>	The referential constraint is dropped. (Hence, the index used by the foreign key is also dropped).

Table 43 lists the cascaded effects of altering a column. (Alter of a column in the following chart means altering a data type, precision, scale, length, or nullability characteristic.)

Table 43. Cascaded effects of altering a column

Operation	Effect
Alter of a column referenced by a view	<p>The alter is allowed.</p> <p>The views that are dependent on the table will be recreated. The new column attributes will be used when recreating the views.</p>
Alter of a column referenced by a non-view logical file	<p>The alter is allowed.</p> <p>The non-view logical files that are dependent on the table will be recreated. If the logical file shares a format with the file being altered, and that format is not used again in the logical file with another based-on file, the new column attributes will be used when recreating the logical file.</p> <p>Otherwise, the new column attributes will not be used when recreating the logical file. Instead, the current logical file attributes are used.</p>
Alter of a column referenced in the key of an index.	The alter is allowed. (Hence, the index will usually be rebuilt.)
Alter of a column referenced in a unique constraint	<p>The alter is allowed. (Hence, the index will usually be rebuilt.)</p> <p>If the unique constraint is referenced by a referential constraint, the attributes of the foreign keys no longer match the attributes of the unique constraint. The constraint will be placed in a defined and check-pending state.</p>
Alter of a column referenced in a referential constraint	<p>The alter is allowed.</p> <ul style="list-style-type: none"> • If the referential constraint is in the defined but check-pending state, the alter is allowed and an attempt is made to put the constraint in the enabled state. (Hence, the index used to satisfy the unique constraint will usually to be rebuilt.) • If the referential constraint is in the enabled state, the constraint is placed in the defined and check-pending state.

ALTER TABLE

Examples

Example 1: Add a new column named RATING, which is one character long, to the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT
ADD RATING CHAR
```

Example 2: Add a new column named PICTURE_THUMBNAIL to the EMPLOYEE table. Create PICTURE_THUMBNAIL 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 converted.

```
ALTER TABLE EMPLOYEE
ADD PICTURE_THUMBNAIL VARCHAR(1000) FOR BIT DATA
```

Example 3: Assume a new table EQUIPMENT has been created with the following columns:

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. 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 EQUIPMENT
FOREIGN KEY DEPTQUIP (EQUIP_OWNER)
REFERENCES DEPARTMENT
ON DELETE SET NULL
```

Change the default value for the EQUIP_OWNER column to 'ABC'.

```
ALTER TABLE EQUIPMENT
ALTER COLUMN EQUIP_OWNER
SET DEFAULT 'ABC'
```

Drop the LOCATION column. Also drop any views, indexes, or constraints that are built on that column.

```
ALTER TABLE EQUIPMENT
DROP COLUMN LOCATION CASCADE
```

Alter the table so that a new column called SUPPLIER is added, the existing column called LOCATION is dropped, a unique constraint over the new column SUPPLIER is added, and a primary key is built over the existing column EQUIP_NO.

```
ALTER TABLE EQUIPMENT
ADD COLUMN SUPPLIER INT
DROP COLUMN LOCATION
ADD UNIQUE SUPPLIER
ADD PRIMARY KEY EQUIP_NO
```

Notice that the column EQUIP_DESC is a variable length column. If an allocated length of 25 was specified, the following ALTER TABLE statement would not change that allocated length.

ALTER TABLE

```
ALTER TABLE EQUIPMENT
  ALTER COLUMN EQUIP_DESC
  SET DATA TYPE VARCHAR(60)
```

Example 4: Alter the EMPLOYEE table. 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 EMPLOYEE table. Drop the constraint REVENUE which was previously defined.

```
ALTER TABLE EMPLOYEE
  DROP CONSTRAINT REVENUE
```

Example 6: Alter the EMPLOYEE table. Alter the column PHONENO to accept up to 20 characters for a phone number.

```
ALTER TABLE EMPLOYEE
  ALTER COLUMN PHONENO SET DATA TYPE VARCHAR (20)
```

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement marks the beginning of an SQL declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java, RPG, or REXX.

Authorization

None required.

Syntax

►►—BEGIN DECLARE SECTION—◄◄

Description

The BEGIN DECLARE SECTION statement is used to indicate the beginning of an SQL declare section. It can be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. It cannot be coded in the middle of a host structure declaration. An SQL declare section ends with an END DECLARE SECTION statement, described in “END DECLARE SECTION” on page 639.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and cannot be nested.

SQL statements should not be included within a declare section, with the exception of the DECLARE VARIABLE and INCLUDE statements.

If SQL declare sections are specified in the program, only the variables declared within the SQL declare sections can be used as host variables. If SQL declare sections are not specified in the program, all variables in the program are eligible for use as host variables.

SQL declare sections should be specified for host languages, other than RPG and REXX, so that the source program conforms to the IBM SQL standard of SQL. SQL declare sections are required for all host variables in C++. The SQL declare section should appear before the first reference to the variable. Host variables are declared without the use of these statements in Java and RPG, and they are not declared at all in REXX.

Variables declared outside an SQL declare section should not have the same name as variables declared within an SQL declare section.

More than one SQL declare section can be specified in the program.

Syntax alternatives: If an ADD constraint is the first clause of the ALTER TABLE statement, the ADD keyword is optional, but strongly recommended. Otherwise, it is required.

Examples

Example 1: Define the host variables hv_smint (SMALLINT), hv_vchar24 (VARCHAR(24)), and hv_double (DOUBLE) in a C program.

```
EXEC SQL BEGIN DECLARE SECTION;
    static short                hv_smint;
    static struct {
        short hv_vchar24_len;
        char  hv_vchar24_value[24];
    }
    static double              hv_double;
EXEC SQL END DECLARE SECTION;
```

Example 2: Define the host variables HV-SMINT (smallint), HV-VCHAR24 (varchar(24)), and HV-DEC72 (dec(7,2)) in a COBOL program.

```
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 HV-SMINT                PIC S9(4)          BINARY.
01 HV-VCHAR24.
   49 HV-VCHAR24-LENGTH PIC S9(4)          BINARY.
   49 HV-VCHAR24-VALUE  PIC X(24).
01 HV-DEC72                PIC S9(5)V9(2)    PACKED-DECIMAL.
    EXEC SQL END DECLARE SECTION END-EXEC.
```

CALL

CALL

The CALL statement calls a procedure.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

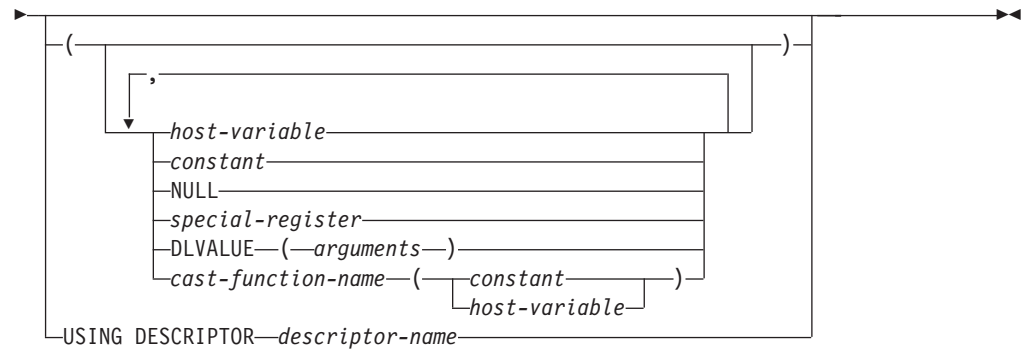
- If the procedure is a REXX external procedure:
 - The system authorities *OBJOPR, *READ, and *EXECUTE on the source file associated with the procedure,
 - The system authority *EXECUTE on the library containing the source file, and
 - The system authority *USE to the CL command,
- If the procedure is a Java external procedure:
 - Read authority (*R) to the integrated file system file that contains the Java class.
 - Read and execute authority (*RX) to all directories that must be accessed in order to find the integrated file system file.
- If the procedure is an external procedure, but not a REXX or Java external procedure:
 - The system authority *EXECUTE on the program associated with the procedure, and
 - The system authority *EXECUTE on the library containing the program associated with the procedure
- If the procedure is an SQL procedure:
 - The EXECUTE privilege on the procedure, and
 - The system authority *EXECUTE on the library containing the SQL procedure
- Administrative authority

The authorization ID of the statement has the EXECUTE privilege on a procedure when:

- It is the owner of the procedure,
- It has been granted the EXECUTE privilege on the procedure, or
- It has been granted the system authorities of *OBJOPR and *EXECUTE on the procedure.

Syntax

►► CALL *procedure-name* *host-variable* →



Description

procedure-name **or** *host-variable*

Identifies the procedure to call by the specified *procedure-name* or the procedure name contained in the *host-variable*. The identified procedure must exist at the current server.

If a *host-variable* is specified:

- It must be a character-string variable or UCS-2 graphic-string.
- It must not include an indicator variable.
- The procedure name that is contained within the host variable must be left-justified and must be padded on the right with blanks if its length is less than that of the host variable.
- The name of the procedure must be in uppercase unless it is a delimited name.

If the procedure name is unqualified, it is implicitly qualified based on the path and number of parameters. For more information see “Qualification of Unqualified Object Names” on page 53.

If the procedure-name identifies a procedure that was defined by a DECLARE PROCEDURE statement, and the current server is a DB2 UDB for iSeries server, then:

- The DECLARE PROCEDURE statement determines the name of the external program, language, and calling convention.
- The attributes of the parameters of the procedure are defined by the DECLARE PROCEDURE statement.

Otherwise:

- The current server determines the name of the external program, language, and calling convention.
- If the current server is DB2 UDB for iSeries:
 - The external program name is assumed to be the same as the external procedure name.
 - If the program attribute information associated with the program identifies a recognizable language, then that language is used. Otherwise, the language is assumed to be C.
 - The calling convention is assumed to be GENERAL.
- The application requester assumes all parameters that are host variables or parameter markers are INOUT. All parameters that are not host variables are assumed to be IN.

CALL

- The actual attributes of the parameters are determined by the current server. If the current server is a DB2 UDB for iSeries, the attributes of the parameters will be the same as the attributes of the arguments specified on the CALL statement.⁴⁵

host-variable or *constant* or **NULL** or *special-register*

Identifies a list of values to be passed as parameters to the procedure. The *n*th value corresponds to the *n*th parameter in the procedure.

Each parameter defined (using a CREATE PROCEDURE or DECLARE PROCEDURE statement) as OUT or INOUT must be specified as a host variable.

The number of arguments specified must be the same as the number of parameters of a procedure defined at the current server with the specified *procedure-name*.

The application requester assumes all parameters that are host variables are INOUT parameters except for Java, where it is assumed all parameters that are host variables are IN unless the mode is explicitly specified in the host variable reference. All parameters that are not host variables are assumed to be input parameters. The actual attributes of the parameters are determined by the current server.

For an explanation of *constant* and *host-variable*, see “Constants” on page 99 and “References to Host Variables” on page 114. For a description of *special-register*, see “Special Registers” on page 105. NULL specifies the null value.

DLVALUE(*arguments*)

Specifies the value for the parameter is the value resulting from a DLVALUE scalar function. A DLVALUE scalar function can only be specified for a DataLink parameter. The DLVALUE function requires a link value on insert (scheme, server, and path/file). The first argument of DLVALUE must be a constant, host variable, or a typed parameter marker (CAST(? AS data-type)). The second and third arguments of DLVALUE must be constants or host-variables.

cast-function-name

This form of an argument can only be used with parameters defined as a distinct type, BLOB, CLOB, DBCLOB, DATE, TIME or TIMESTAMP data types. The following table describes the allowed uses of these *cast-functions*.

Parameter Type	Cast Function Name
Distinct type N based on a BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
Distinct type N based on a DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *
BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *
Notes:	
* The name of the function must match the name of the data type (or the source type of the distinct type) with an implicit or explicit schema name of QSYS2.	

45. Note that in the case of decimal constants, leading zeroes are significant when determining the attributes of the argument. Normally, leading zeroes are not significant.

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. For BLOB, CLOB, DBCLOB, DATE, TIME, and TIMESTAMP functions, the constant must be a string constant.

host-variable

Specifies a host variable as the argument. The host variable 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.

USING DESCRIPTOR *descriptor-name*

Identifies an SQLDA that must contain a valid description of host variables.

Before the CALL statement is processed, you must set the following fields in the SQLDA. (The rules for REXX are different. For more information, see the SQL Programming with Host Languages book.)

- 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 $16 + \text{SQLN} \times (80)$, where 80 is the length of an SQLVAR occurrence. If LOBs or distinct types are specified, there must be two SQLVAR entries for each parameter marker and SQLN must be set to two times the number of parameter markers.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameters in the CALL statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement. (For a description of an SQLDA, see Appendix D, "SQL Descriptor Area (SQLDA)" on page 847.)

Note that RPG/400 does not provide the function for setting pointers. Because the SQLDA uses pointers to locate the appropriate host variables, you have to set these pointers outside your RPG/400 application.

Notes

Parameter assignments: When the CALL statement is executed, the value of each of its parameters is assigned (using storage assignment) to the corresponding parameter of the procedure. Control is passed to the procedure according to the calling conventions of the host language. When execution of the procedure is complete, the value of each parameter of the procedure is assigned (using storage assignment) to the corresponding parameter of the CALL statement defined as OUT or INOUT. For details on the assignment rules, see "Assignments and Comparisons" on page 80.

Cursors and prepared statements in procedures: All cursors opened in the called procedure that are not result set cursors are closed and all statements prepared in the called procedure are destroyed when the procedure ends.

CALL

Result Sets from procedures: Result sets are only returned from a procedure when the procedure is called from a client using the iSeries Access Open Database Connectivity (ODBC) driver, a client using the iSeries Access Optimized SQL API, from the SQL Call Level Interface, or from JDBC. There are three ways to return result sets from a procedure:

- If a SET RESULT SETS statement is executed in the procedure, the SET RESULT SETS statement identifies the result sets. The result sets are returned in the order specified on the SET RESULT SETS statement.
- If a SET RESULT SETS statement is not executed in the procedure,
 - If no cursors have specified a WITH RETURN clause, each cursor that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.
 - If any cursors have specified a WITH RETURN clause, each cursor that is defined with the WITH RETURN clause that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.

When a result set is returned using an open cursor, the rows are returned starting with the current cursor position.

Nesting CALL Statements: A program that is executing as a procedure, a user-defined function, or a trigger can issue a CALL statement. When a procedure, user-defined function, or trigger calls a procedure, user-defined function, or trigger, the call is considered to be nested. There is no limit on how many levels procedures and functions can be nested, but triggers can only be nested up to 300 levels deep.

If a procedure returns any query result sets, the result sets are returned to the caller of the procedure. If the SQL CALL statement is nested, the result sets are visible only to the program that is at the previous nesting level. For example, if a client program calls procedure PROCA, which in turn calls procedure PROCB. Only PROCA can access any result sets that PROCB returns; the client program has no access to the query result sets.

When an SQL or an external procedure is called, an attribute is set for SQL data-access that was defined when the procedure was created. The possible values for the attribute are:

NONE
CONTAINS
READS
MODIFIES

If a second procedure is invoked within the execution of the current procedure, an error is issued if:

- The invoked procedure possibly contains SQL and the invoking procedure does not allow SQL
- The invoked procedure reads SQL data and the invoking procedure does not allow reading SQL data
- The invoked procedure modifies SQL data and the invoking procedure does not allow modifying SQL data

REXX procedures: If the external procedure to be called is a REXX procedure, then the procedure must be declared using the CREATE PROCEDURE or DECLARE PROCEDURE statement.

Host variables cannot be used in the CALL statement within a REXX procedure. Instead, the CALL must be the object of a PREPARE and EXECUTE using parameter markers.

Examples

Example 1: Call procedure PGM1 and pass two parameters.

```
CALL PGM1 (:hv1, :hv2)
```

Example 2: In C, invoke a procedure called SALARY_PROCEED 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 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)
LANGUAGE JAVA PARAMETER STYLE JAVA
EXTERNAL NAME 'parts!onhand';
```

A Java application calls this procedure on the connection context 'ctx' using the following code fragment:

```
...
int      variable1;
BigDecimal variable2;
Integer  variable3;
...
#sql [ctx] {CALL PARTS_ON_HAND(:IN variable1, :OUT variable2, :OUT variable3)};
...
```

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

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 only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required. See “DECLARE CURSOR” on page 576 for the authorization required to use a cursor.

Syntax

►►—CLOSE—*cursor-name*—►►

Description

cursor-name

Identifies the cursor to be closed. The *cursor-name* must identify a declared cursor as explained in the DECLARE CURSOR statement. When the CLOSE statement is executed, the cursor must be in the open state.

Notes

Implicit cursor close: All cursors in a program are in the closed state when:

- The program is called.
 - If CLOSQLCSR(*ENDPGM) is specified, all cursors are in the closed state each time the program is called.
 - If CLOSQLCSR(*ENDSQL) is specified, all cursors are in the closed state only the first time the program is called as long as one SQL program remains on the call stack.
 - If CLOSQLCSR(*ENDJOB) is specified, all cursors are in the closed state only the first time the program is called in the job.
 - If CLOSQLCSR(*ENDMOD) is specified, all cursors are in the closed state each time the module is initiated.
 - If CLOSQLCSR(*ENDACTGRP) is specified, all cursors are in the closed state the first time the module in the program is initiated within the activation group.
- A program starts a new unit of work by executing a COMMIT or ROLLBACK statement without a HOLD option. Cursors declared with the HOLD option are not closed by a COMMIT statement.

Note: The DB2 UDB for iSeries database manager will open files in order to implement queries. The closing of the files can be separate from the SQL CLOSE statement. For more information, see the SQL Programming Concepts book.

Close cursors for performance: Explicitly closing cursors as soon as possible can improve performance.

Procedure considerations: Special rules apply to cursors within procedures that have not been closed before returning to the calling program. For more information, see “CALL” on page 396.

Example

In a COBOL program, use the cursor C1 to fetch the values from the first four columns of the EMPPROJECT table a row at a time and put them in the following host variables:

- EMP (CHAR(6))
- PRJ (CHAR(6))
- ACT (SMALLINT)
- TIM (DECIMAL(5,2))

Finally, close the cursor.

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  77 EMP          PIC X(6).
  77 PRJ          PIC X(6).
  77 ACT          PIC S9(4) BINARY.
  77 TIM          PIC S9(3)V9(2) PACKED-DECIMAL.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.

EXEC SQL DECLARE C1 CURSOR FOR
      SELECT EMPNO, PROJNO, ACTNO, EMPTIME
      FROM EMPPROJECT          END-EXEC.

EXEC SQL OPEN C1 END-EXEC.

EXEC SQL FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM END-EXEC.

IF SQLSTATE = '02000'
  PERFORM DATA-NOT-FOUND
ELSE
  PERFORM GET-REST-OF-ACTIVITY UNTIL SQLSTATE IS NOT EQUAL TO '00000'.

EXEC SQL CLOSE C1 END-EXEC.

GET-REST-OF-ACTIVITY
EXEC SQL FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM END-EXEC.
.
.
.

```

COMMENT

The COMMENT statement adds or replaces comments in the catalog descriptions of various database objects.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

To comment on a table, view, alias, index, column, distinct type, or package, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the table, view, alias, index, distinct type, or package in the statement,
 - The ALTER privilege on the table, view, alias, index, distinct type, or package, and
 - The system authority *EXECUTE on the library that contains the table, view, alias, index, distinct type, or package
- Administrative authority

The authorization ID of the statement has the ALTER privilege on the table, view, alias, index, distinct type, or package when:

- It is the owner of the table, view, alias, index, distinct type, or package
- It has been granted the ALTER privilege to the table, view, alias, distinct type, or package, or
- It has been granted the system authorities of either *OBJALTER or *OBJMGT to the table, view, alias, index, distinct type, or package

To comment on a trigger, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the subject table of the trigger in the statement:
 - The ALTER privilege on the subject table, and
 - The system authority *EXECUTE on the library that contains the subject table
- Administrative authority

To comment on a function, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSFUNCS catalog view:
 - The UPDATE privilege on the view
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

To comment on a procedure, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPROCS catalog view:
 - The UPDATE privilege on the view, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

To comment on a parameter, the privileges held by the authorization ID of the statement must include at least one of the following:

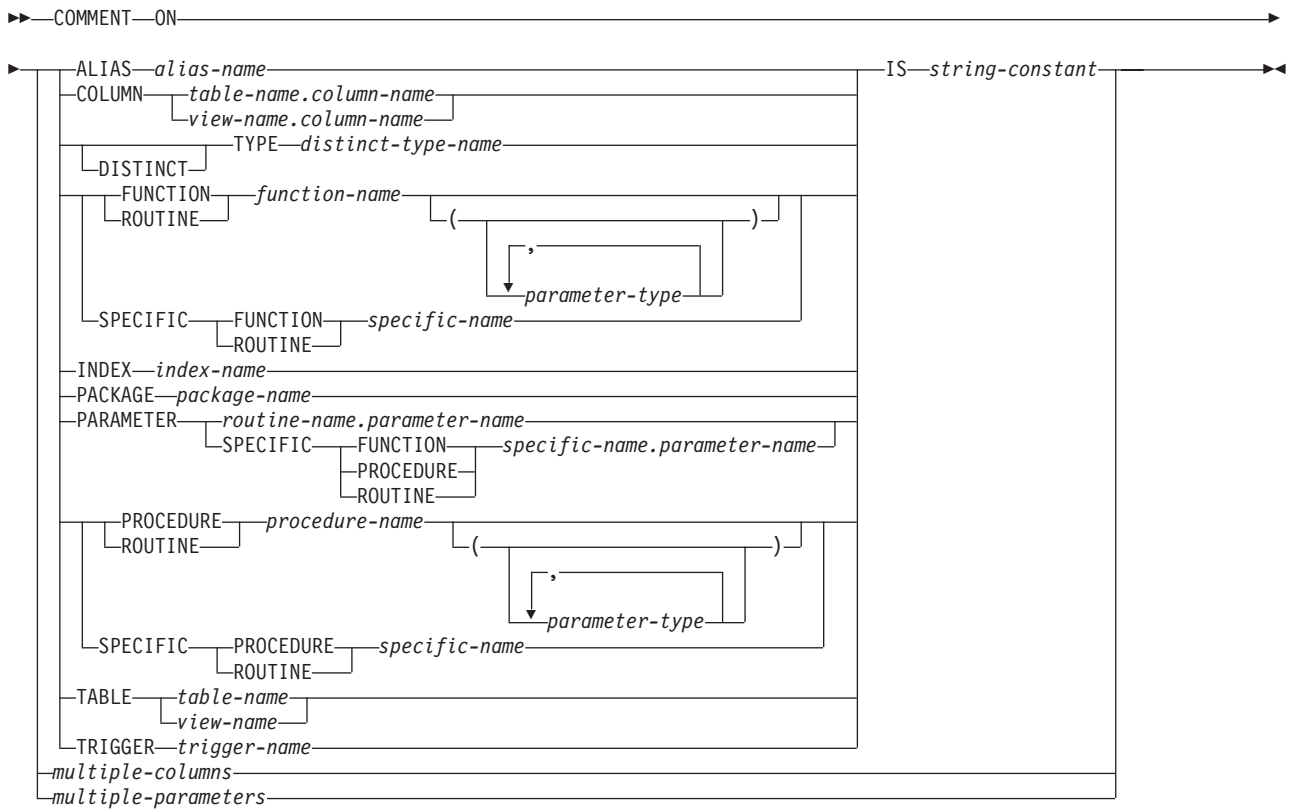
- For the SYSPARMS catalog table:
 - The UPDATE privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

The authorization ID of the statement has the UPDATE privilege on a table when any of these are true:

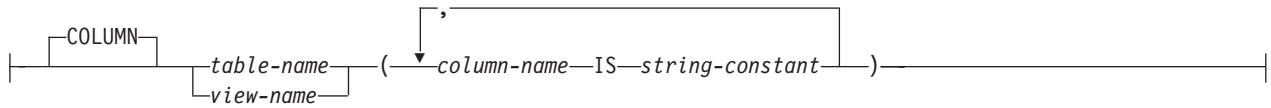
- It is the owner of the table
- It has been granted the UPDATE privilege on the table
- It has been granted the system authorities of *OBJOPR and *UPD on the table.

COMMENT

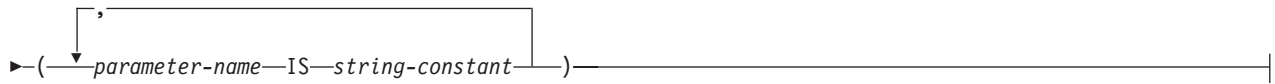
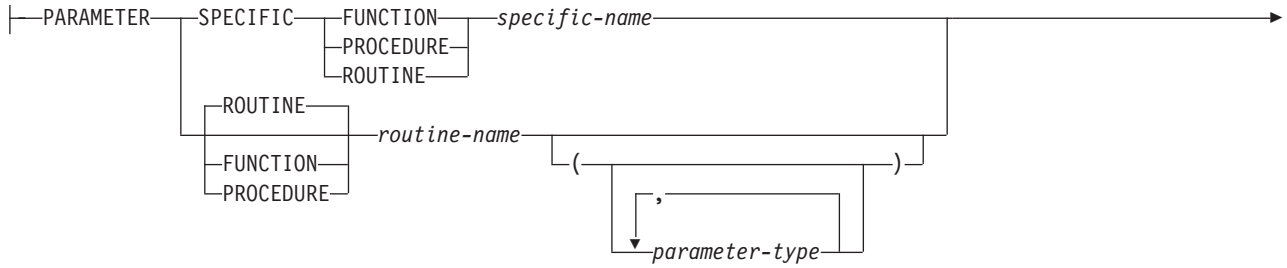
Syntax



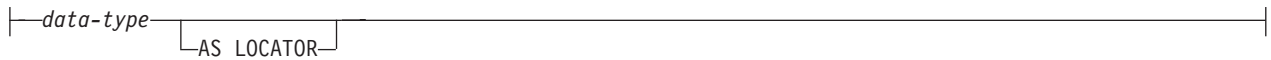
multiple-columns:



multiple-parameters:



parameter-type:

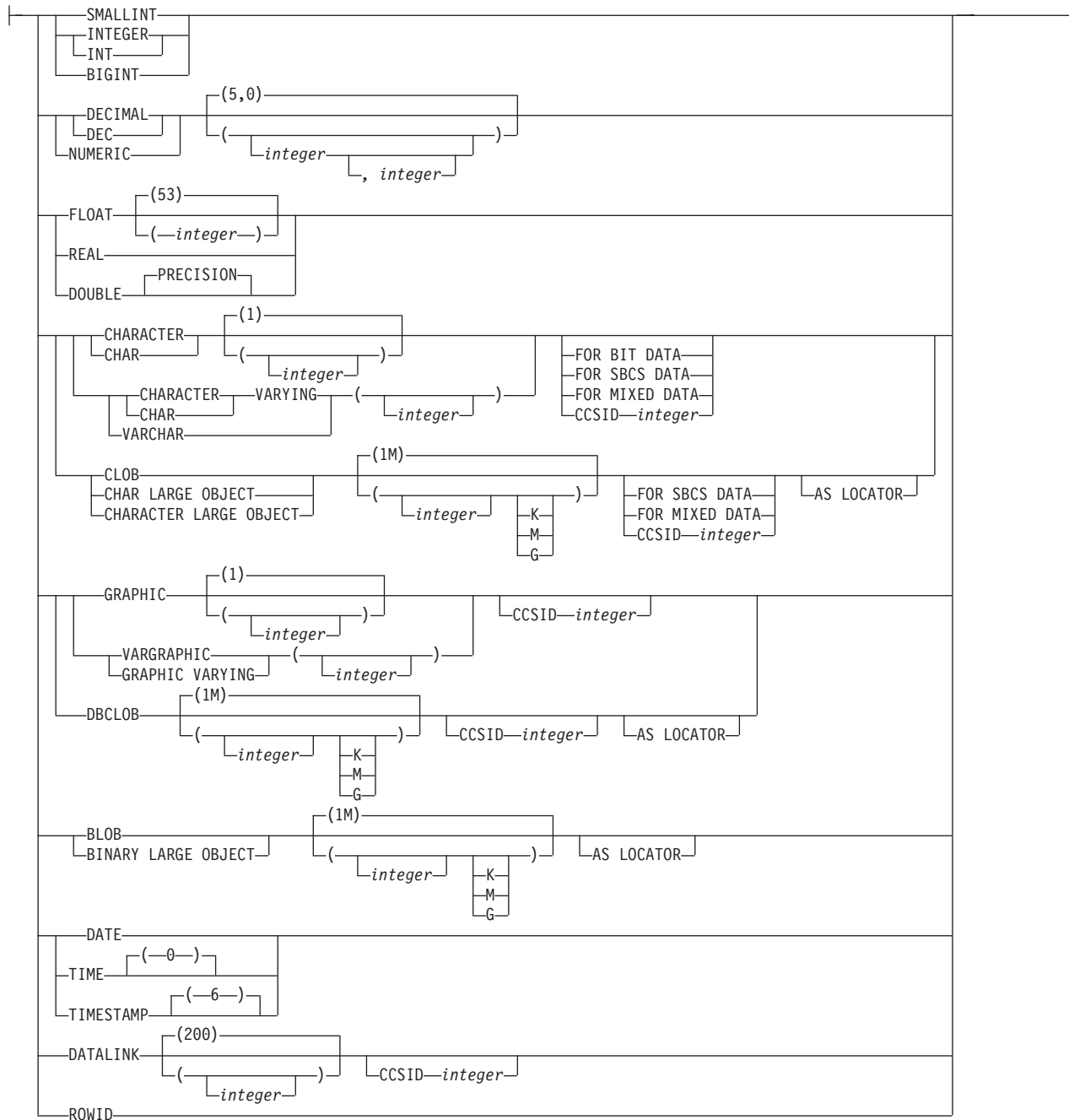


data-type:



COMMENT

built-in-type:



Description

ALIAS *alias-name*

Identifies the alias to which the comment applies. The *alias-name* must identify an alias that exists at the current server.

COLUMN

Specifies that a comment will be added to or replaced for a column.

table-name.column-name or *view-name.column-name*

Identifies the column to which the comment applies. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a global temporary table. The *column-name* must identify a column of that table or view.

DISTINCT TYPE *distinct-type-name*

Identifies the distinct type to which the comment applies. The *distinct-type-name* must identify a distinct type that exists at the current server.

FUNCTION or SPECIFIC FUNCTION

Identifies the function on which the comment applies. The function must exist at the current server and it must be a user-defined function. The function can be identified by its name, function signature, or specific name.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which to comment. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type, ...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type

COMMENT

are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX *index-name*

Identifies the index to which the comment applies. The *index-name* must identify an index that exists at the current server.

PACKAGE *package-name*

Identifies the package to which the comment applies. The *package-name* must identify a package that exists at the current server.

PARAMETER

Specifies that a comment will be added to or replaced for a parameter.

routine-name.parameter-name

Identifies the parameter to which the comment applies. The parameter could be for a procedure or a function. The *routine-name* must identify a procedure or function that exists at the current server, and the *parameter-name* must identify a parameter of that procedure or function.

specific-name.parameter-name

Identifies the parameter to which the comment applies. The parameter could be for a procedure or a function. The *specific-name* must identify a procedure or function that exists at the current server, and the *parameter-name* must identify a parameter of that procedure or function.

PROCEDURE or SPECIFIC PROCEDURE

Identifies the procedure on the comment applies. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*

Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*

Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. 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 commented on. Synonyms for data types are considered a match.

If *procedure-name* () is specified, the procedure identified must have zero parameters.

procedure-name

Identifies the name of the procedure.

(*parameter-type*, ...)

Identifies the parameters of the procedure.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional.

Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR

Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC PROCEDURE *specific-name*

Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

TABLE *table-name* or *view-name*

Identifies the table or view to which the comment applies. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a global temporary table.

TRIGGER *trigger-name*

Identifies the trigger to which the comment applies. The *trigger-name* must identify a trigger that exists at the current server.

IS

Introduces the comment that to be added or replaced.

COMMENT

string-constant

Can be any character-string constant of up to 2000 characters.

multiple-columns

To comment on more than one column in a table or view, specify the table or view name and then, in parenthesis, a list of the form:

```
column-name IS string-constant,  
column-name IS string-constant, ...
```

The column name must not be qualified, each name must identify a column of the specified table or view, and that table or view must exist at the current server.

multiple-parameters

To comment on more than one parameter in a procedure or function, specify the procedure name, function name, or specific name, and then, in parenthesis, a list of the form:

```
parameter-name IS string-constant,  
parameter-name IS string-constant, ...
```

The parameter name must not be qualified, each name must identify a parameter of the specified procedure or function, and that procedure or function must exist at the current server.

Notes

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword PROGRAM can be used as a synonym for PACKAGE.
- The keyword DATA can be used as a synonym for DISTINCT.

Examples

Example 1: Insert a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE  
IS 'Reflects first quarter 2000 reorganization'
```

Example 2: Insert a comment for the EMP_VIEW1 view.

```
COMMENT ON TABLE EMP_VIEW1  
IS 'View of the EMPLOYEE table without salary information'
```

Example 3: Insert a comment for the EDLEVEL column of the EMPLOYEE table.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL  
IS 'Highest grade level passed in school'
```

Example 4: Enter comments on two columns in the DEPARTMENT table.

```
COMMENT ON DEPARTMENT  
(MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',  
ADMNDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT')
```

Example 5: Insert a comment for the PAYROLL package.

```
COMMENT ON PACKAGE PAYROLL  
IS 'This package is used for distributed payroll processing.'
```


COMMIT

The COMMIT statement ends 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 interactively. It is an executable statement that can be dynamically prepared.

COMMIT is not allowed in a trigger if the trigger program and the triggering program run under the same commitment definition. COMMIT is not allowed in a procedure if the procedure is called on a remote server.

Authorization

None required.

Syntax



Description

The COMMIT statement ends the unit of work in which it is executed and starts a new unit of work. It commits all changes made by SQL schema statements (except DROP SCHEMA) and SQL data change statements during the unit of work. For information on SQL schema statements and SQL data change statements see Table 33 on page 361 and Table 34 on page 362.

Connections in the release-pending state are ended.

WORK

COMMIT WORK has the same effect as COMMIT.

HOLD

Specifies a hold on resources. If specified, currently open cursors are not closed and all resources acquired during the unit of work are held. Locks on specific rows and objects implicitly acquired during the unit of work are released.

All implicitly acquired locks are released; except for object level locks required for the cursors that are not closed.

All locators that are not held are released. For more information on held locators, see "HOLD LOCATOR" on page 669.

Notes

Recommended coding practices: An explicit COMMIT or ROLLBACK statement should be coded at the end of an application process. Either an implicit commit or rollback operation will be performed at the end of an application process depending on the application environment. Thus, a portable application should explicitly execute a COMMIT or ROLLBACK before execution ends in those environments where explicit COMMIT or ROLLBACK is permitted.

COMMIT

An implicit COMMIT or ROLLBACK may be performed under the following circumstances.

- For the default activation group:
 - An implicit COMMIT is not performed when applications that run in the default activation group end. Interactive SQL, Query Manager, and non-ILE programs are examples of programs that run in the default activation group.
 - In order to commit work, you must issue a COMMIT.
- For non-default activation groups when the scope of the commitment definition is to the activation group:
 - If the activation group ends normally, the commitment definition is implicitly committed.
 - If the activation group ends abnormally, the commitment definition is implicitly rolled back.
- Regardless of the type of activation group, if the scope of the commitment definition is the job, an implicit commit is never performed.

Effect of commit: Commit without HOLD causes the following to occur:

- Connections in the release-pending state are ended.
For existing connections:
 - all open cursors that were declared with the WITH HOLD clause are preserved and their current position is maintained, although a FETCH statement is required before a Positioned UPDATE or Positioned DELETE statement can be executed.
 - all open cursors that were declared without the WITH HOLD clause are closed.
- 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.
- All locks acquired by the LOCK TABLE statement are released. All implicitly acquired locks are released, except for those required for the cursors that were not closed.

Row lock limit: A unit of work can include the processing of up to 4 million rows, including rows retrieved during a SELECT or FETCH statement⁴⁶, and rows inserted, deleted, or updated as part of INSERT, DELETE, and UPDATE statements.⁴⁷

Unaffected statements: The commit and rollback operations do not affect the DROP SCHEMA statement, and this statement is not, therefore, allowed in an application program that also specifies COMMIT(*CHG), COMMIT(*CS), COMMIT(*ALL), or COMMIT(*RR).

Commitment definition use: The commitment definition used by SQL is determined as follows:

46. This limit also includes:

- Any rows accessed or changed through files opened under commitment control through high-level language file processing
- Any rows deleted, updated, or inserted as a result of a trigger or CASCADE, SET NULL, or SET DEFAULT referential integrity delete rule.

47. Unless you specified COMMIT(*CHG) or COMMIT(*CS), in which case these rows are not included in this total.

- If the activation group of the program calling SQL is already using an activation group level commitment definition, then SQL uses that commitment definition.
- If the activation group of the program calling SQL is using the job level commitment definition, then SQL uses the job level commitment definition.
- If the activation group of the program calling SQL is not currently using a commitment definition but the job commitment definition is started, then SQL uses the job commitment definition.
- If the activation group of the program calling SQL is not currently using a commitment definition and the job commitment definition is not started, then SQL implicitly starts a commitment definition. SQL uses the Start Commitment Control (STRCMTCTL) command with:
 - A CMTSCOPE(*ACTGRP) parameter
 - A LCKLVL parameter based on the COMMIT option specified on either the CRTSQLxxx, STRSQL, or RUNSQLSTM commands. In REXX, the LCKLVL parameter is based on the commit option in the SET OPTION statement.

Example

In a C program, transfer a certain amount of commission (COMM) from one employee (EMPNO) to another in the EMPLOYEE table. Subtract the amount from one row and add it to the other. Use the COMMIT statement to ensure that no permanent changes are made to the database until both operations are completed successfully.

```
void main ()
{
    EXEC SQL BEGIN DECLARE SECTION;
    decimal(5,2) AMOUNT;
    char FROM_EMPNO[7];
    char TO_EMPNO[7];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL WHENEVER SQLERROR GOTO SQLERR;
    ...
    EXEC SQL UPDATE EMPLOYEE
        SET COMM = COMM - :AMOUNT
        WHERE EMPNO = :FROM_EMPNO;
    EXEC SQL UPDATE EMPLOYEE
        SET COMM = COMM + :AMOUNT
        WHERE EMPNO = :TO_EMPNO;
    FINISHED:
    EXEC SQL COMMIT WORK;
    return;

    SQLERR:
    ...
    EXEC SQL WHENEVER SQLERROR CONTINUE; /* continue if error on rollback */
    EXEC SQL ROLLBACK WORK;
    return;
}
```

CONNECT (Type 1)

The CONNECT (TYPE 1) statement connects an activation group within an application process to the identified server using the rules for remote unit of work. This server is then the current server for the activation group. This type of CONNECT statement is used if RDBCNNMTH(*RUW) was specified on the CRTSQLxxx command. Differences between the two types of statements are described in “CONNECT (Type 1) and CONNECT (Type 2) Differences” on page 845. Refer to “Application-Directed Distributed Unit of Work” on page 29 for more information about connection states.

Invocation

This statement can only be embedded within an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

CONNECT is not allowed in a trigger, a function, or a procedure if the procedure is called on a remote server.

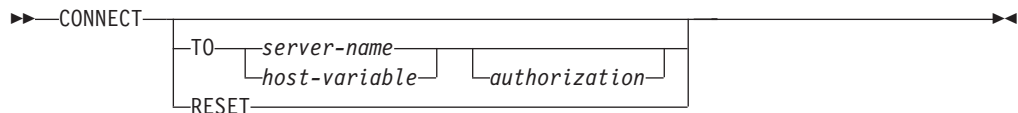
Authorization

The privileges held by the authorization ID of the statement must include communications-level security. (See the section about security in the Distributed Database Programming book.)

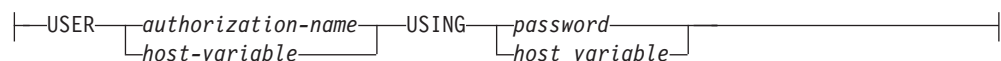
If the server is DB2 UDB for iSeries, the user profile of the person issuing the statement must also be a valid user profile on the server system, UNLESS:

- User is specified. In this case, the USER clause must specify a valid user profile on the server system.
- TCP/IP is used with a server authorization entry for the server. In this case, the server authorization entry must specify a valid user profile on the server system.

Syntax



authorization:



Description

TO *server-name* or *host-variable*

Identifies the server by the specified server name or the server name contained in the host variable. If a host variable is specified:

- It must be a character-string variable.
- It must not be followed by an indicator variable.

CONNECT (Type 1)

- The server name must be left-justified within the host variable and must conform to the rules for forming an ordinary identifier.
- If the length of the server name is less than the length of the host variable, it must be padded on the right with blanks.

When the CONNECT statement is executed, the specified server name or the server name contained in the host variable must identify a server described in the local directory and the activation group must be in the connectable state.

If the *server-name* is a local relational database and an authorization-name is specified, it must be the authorization-name of the job. If the specified authorization-name is different than the authorization-name of the job, an error occurs and the application is left in the unconnected state.

USER *authorization-name* or *host-variable*

Identifies the authorization-name by the specified *authorization-name* or a *host-variable* which contains the authorization name that will be used to start the remote job.

If a *host-variable* is specified,

- It must be a character string variable.
- It must not be followed by an indicator variable.
- The authorization name must be left-justified within the host variable and must conform to the rules of forming an authorization name.
- If the length of the authorization name is less than the length of the host variable, it must be padded on the right with blanks.
- The value of the server name must not contain lowercase characters.

USING *password* or *host-variable*

Identifies the password by the specified *password* or a *host-variable*, which contains the password for the authorization-name that will be used to start the remote job.

If password is specified as a literal, it must be a character string. The maximum length is 128 characters. It must be left justified.

If a *host-variable* is specified,

- It must be a character-string variable.
- It must not be followed by an indicator variable.
- The password must be left-justified within the host variable.
- If the length of the password is less than that of the host variable, it must be padded on the right with blanks.

RESET

CONNECT RESET is equivalent to CONNECT TO x where x is the local server name.

CONNECT with no operand

This form of the CONNECT statement returns information about the current server and has no effect on connection states, open cursors, prepared statements, or locks. The information is returned in the SQLCA as described above.

Notes

Successful connect: If the CONNECT statement is successful:

CONNECT (Type 1)

- All open cursors are closed, all prepared statements are destroyed, and all locks are released from the current connection.
- The activation group is disconnected from all current and dormant connections, if any, and connected to the identified server.
- The name of the server is placed in the CURRENT SERVER special register.
- Information about the server is placed in the SQLERRP and SQLERRD(4) fields of the SQLCA. If the server is an IBM relational database product, the information in the SQLERRP field has the form *pppvrrm*, where:
 - *ppp* identifies the product as follows:
 - ARI for DB2 for VM and VSE
 - DSN for DB2 UDB for OS/390 and z/OS
 - QSQ for DB2 UDB for iSeries
 - SQL for all other DB2 UDB products
 - *vv* is a two-digit version identifier such as '07'
 - *rr* is a two-digit release identifier such as '01'
 - *m* is a one-digit modification level such as '0'

For example, if the server is Version 7 of DB2 UDB for OS/390 and z/OS, the value of SQLERRP is 'DSN07010'.

The SQLERRD(4) field of the SQLCA contains values indicating whether the server allows commitable updates to be performed. For a CONNECT (Type 1) statement SQLERRD(4) will always contain the value 1. The value 1 indicates that commitable updates can be performed, and the connection:

- Uses an unprotected conversation,⁴⁸ or
 - Is a connection to an application requester driver program using the *RUW connection method, or
 - Is a local connection using the *RUW connection method.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. Refer to Appendix B, "SQL Communication Area" on page 825

Unsuccessful connect: If the CONNECT 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 identify the product. For example, if the application requester is DB2 UDB UWO for NT the first three characters are 'SQL'.

If the CONNECT statement is unsuccessful because the activation group is not in the connectable state, the connection state of the activation group is unchanged.

If the CONNECT statement is unsuccessful for any other reason:

- The activation group remains in a connectable, but unconnected state
- All open cursors are closed, all prepared statements are destroyed, and all locks are released from all current and dormant connections.

An application in a connectable but unconnected state can only execute the CONNECT or SET CONNECTION statements.

Implicit connect:

48. To reduce the possibility of confusion between network connections and SQL connections, in this book the term 'conversation' will be used to apply to network connections over TCP/IP as well as over APPC, even though it formally applies only to APPC connections.

CONNECT (Type 1)

- When running in the default activation group, the SQL program implicitly connects to a remote relational database when:
 - The activation group is in a connectable state.
 - The first SQL statement in the first SQL program on the program stack is executed.
- When running in a non-default activation group, the SQL program implicitly connects to a remote relational database when the first SQL statement in the first SQL program for that activation group is executed.

Note: It is a good practice for the first SQL statement executed by an activation group to be the CONNECT statement.

When APPC is used for connecting to an RDB, implicit connect always sends the *authorization-name* of the application requester job and does not send passwords. If the *authorization-name* of the server job is different, or if a password must be sent, an explicit connect statement must be used.

When TCP/IP is used for connecting to an RDB, an implicit connect is not bound by the above restrictions. Use of the ADDSVRAUTE and other -SVRAUTE commands allows one to specify, for a given user under which the implicit (or explicit) CONNECT is done, the remote authorization-name and password to be used in connecting to a given RDB.

In order for the password to be stored with the ADDSVRAUTE or CHGSVRAUTE command, the QRETSVRSEC system value must be set to '1' rather than the default of '0'. When using these commands for DRDA connection, it is very important to realize that the value of the RDB name entered into the SERVER parameter must be in UPPER CASE. For more information, see Example 2 under Type 2 CONNECT.

For more information about implicit connect, refer to the SQL Programming Concepts book. Once a connection to a relational database for a user profile is established, the password, if specified, may not be validated again on subsequent connections to the same relational database with the same user profile. Revalidation of the password depends on if the conversation is still active. See the Distributed Database Programming book for more details.

Connection states: For a description of connection states, see "Remote Unit of Work Connection Management" on page 27. Consecutive CONNECT statements can be executed successfully because CONNECT does not remove the activation group from the connectable state.

A CONNECT to either a current or dormant connection in the application group is executed as follows:

- If the connection identified by the server-name was established using a CONNECT (Type 1) statement, then no action is taken. Cursors are not closed, prepared statements are not destroyed, and locks are not released.
- If the connection identified by the server-name was established using a CONNECT (Type 2) statement, then the CONNECT statement is executed like any other CONNECT statement.

CONNECT cannot execute successfully when it is preceded by any SQL statement other than CONNECT, COMMIT, DISCONNECT, SET CONNECTION, RELEASE, or ROLLBACK. To avoid an error, execute a commit or rollback operation before a CONNECT statement is executed.

CONNECT (Type 1)

If any previous current or dormant connections were established using protected conversations, then the CONNECT (Type 1) statement will fail. Either, a CONNECT (Type 2) statement must be used, or the connections using protected conversations must be ended by releasing the connections and successfully committing.

For more information about connecting to a remote relational database and the local directory, see the SQL Programming Concepts book and the Distributed Database Programming book.

Examples

Example 1: In a C program, connect to the application server TOROLAB.

```
EXEC SQL CONNECT TO TOROLAB;
```

Example 2: In a C program, connect to an application server whose name is stored in the host variable APP_SERVER (VARCHAR(18)). Following a successful connection, copy the 3 character product identifier of the application server to the host variable PRODUCT (CHAR(3)).

```
void main ()
{
    char product[4] = " ";
    EXEC SQL BEGIN DECLARE SECTION;
    char APP_SERVER[19];
    char username[11];
    char userpass[129];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL INCLUDE SQLCA;
    strcpy(APP_SERVER,"TOROLAB");
    strcpy(username,"JOE");
    strcpy(userpass,"XYZ1");
    EXEC SQL CONNECT TO :APP_SERVER
            USER :username USING :userpass;
    if (strcmp(SQLSTATE, "00000", 5) )
        { strcpy(product,sqlca.sqlerrp,3); }
    ...
    return;
}
```


CONNECT (Type 2)

The CONNECT (Type 2) statement connects an activation group within an application process to the identified server using the rules for application directed distributed unit of work. This server is then the current server for the activation group. This type of CONNECT statement is used if RDBCNNMTH(*DUW) was specified on the CRTSQLxxx command. Differences between the two types of statements are described in “CONNECT (Type 1) and CONNECT (Type 2) Differences” on page 845. Refer to “Application-Directed Distributed Unit of Work” on page 29 for more information about connection states.

Invocation

This statement can only be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

CONNECT is not allowed in a trigger, a function, or a procedure if the procedure is called on a remote server.

Authorization

The privileges held by the authorization ID of the statement must include communications-level security. (See the section about security in the Distributed Database Programming book.)

If the server is DB2 UDB for iSeries, the profile ID of the person issuing the statement must also be a valid user profile on the server system, UNLESS:

- USER is specified. If USER is specified, the USER clause must specify a valid user profile on the server system.
- TCP/IP is used with a server authorization entry for the server. If this is the case, the server authorization entry must specify a valid user profile on the server system.

Syntax

```

CONNECT
  TO server-name | host-variable | authorization
  RESET
  
```

authorization:

```

USER authorization-name | host-variable USING password | host variable
  
```

Description

TO *server-name* or *host-variable*

Identifies the server by the specified server name or the server name contained in the host variable. If a host variable is specified:

- It must be a character-string variable.
- It must not be followed by an indicator variable

CONNECT (Type 2)

- The server name must be left-justified within the host variable and must conform to the rules for forming an ordinary identifier
- If the length of the server name is less than the length of the host variable, it must be padded on the right with blanks.
- The value of the server name must not contain lowercase characters.

When the CONNECT statement is executed, the specified server name or the server name contained in the host variable must identify a server described in the local directory.

Let S denote the specified server name or the server name contained in the host variable. S must not identify an existing connection of the application process.

USER *authorization-name* or *host-variable*

Identifies the authorization-name by the specified *authorization-name* or a *host-variable*, which contains the authorization name that will be used to start the remote job.

If a *host-variable* is specified,

- It must be a character string variable.
- It must not be followed by an indicator variable. The authorization name must be left-justified within the host variable and must conform to the rules of forming an authorization name.
- If the length of the authorization name is less than the length of the host variable, it must be padded on the right with blanks.

USING *password* or *host-variable*

Identifies the password by the specified *password* or a *host-variable*, which contains the password for the authorization-name that will be used to start the remote job.

If password is specified as a literal, it must be a character string. The maximum length is 128 characters. It must be left justified.

If a *host-variable* is specified,

- It must be a character-string variable.
- It must not be followed by an indicator variable.
- The password must be left-justified within the host variable.
- If the length of the password is less than that of the host variable, it must be padded on the right with blanks.

RESET

CONNECT RESET is equivalent to CONNECT TO *x* where *x* is the local server name.

CONNECT with no operand

This form of the CONNECT statement returns information about the current server and has no effect on connection states, open cursors, prepared statements, or locks. The information is returned in the fields of the SQLCA as described above.

In addition, the SQLERRD(3) field of the SQLCA will indicate the status of connection for this unit of work. It will have one of the following values:

- 1 - commitable updates can be performed on the connection for this unit of work.

- 2 - No committable updates can be performed on the connection for this unit of work.

Notes

Successful connect: If the CONNECT statement is successful:

- A connection to server S is created and placed in the current and held states. The previous connection, if any, is placed in the dormant state.
- S is placed in the CURRENT SERVER special register.
- Information about server S is placed in the SQLERRP and SQLERRD(4) fields of the SQLCA. If the server is an IBM relational database product, the information in the SQLERRP field has the form *pppvrrm*, where:
 - *ppp* identifies the product as follows:
ARI for DB2 for VM and VSE
DSN for DB2 UDB for OS/390 and z/OS
QSQ for DB2 UDB for iSeries
SQL for all other DB2 UDB products
 - *vv* is a two-digit version identifier such as '07'
 - *rr* is a two-digit release identifier such as '01'
 - *m* is a one-digit modification level such as '0'

For example, if the server is Version 7 of DB2 UDB for OS/390 and z/OS, the value of SQLERRP is 'DSN07010'.

The SQLERRD(4) field of the SQLCA contains values indicating whether server S allows committable updates to be performed. Following is a list of values and their meanings for the SQLERRD(4) field of the SQLCA on the CONNECT:

- 1 - committable updates can be performed. Conversation is unprotected. ⁴⁸
- 2 - No committable updates can be performed. Conversation is unprotected.
- 3 - It is unknown if committable updates can be performed. Conversation is protected.
- 4 - It is unknown if committable updates can be performed. Conversation is unprotected.
- 5 - It is unknown if committable updates can be performed. The connection is either a local connection or a connection to an application requester driver program.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. Refer to Appendix B, "SQL Communication Area" on page 825.

Unsuccessful connect: If the CONNECT statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Implicit connect: Implicit connect will always send the *authorization-name* of the application requester job and will not send passwords. If the *authorization-name* of the server job is different or if a password must be sent, an explicit connect statement must be used.

When TCP/IP is used for connecting to an RDB, an implicit connect is not bound by the above restrictions. Use of the ADDSVRAUTE and other -SVRAUTE commands allows one to specify, for a given user under which the implicit (or explicit) CONNECT is done, the remote authorization-name and password to be used in connecting to a given RDB.

CONNECT (Type 2)

In order for the password to be stored with the ADDSVRAUTE or CHGSVRAUTE command, the QRETSVRSEC system value must be set to '1' rather than the default of '0'. When using these commands for DRDA connection, it is very important to realize that the value of the RDB name entered into the SERVER parameter must be in UPPER CASE. For more information, see Example 2 under Type 2 CONNECT.

For more information about implicit connect, refer to the SQL Programming Concepts book. Once a connection to a relational database for a user profile is established, the password, if specified, may not be validated again on subsequent connections to the same relational database with the same user profile. Revalidation of the password depends on if the conversation is still active. See the Distributed Database Programming book for more details.

Examples

Example 1: Execute SQL statements at TOROLAB and SVLLAB. The first CONNECT statement creates the TOROLAB connection and the second CONNECT statement places it in the dormant state.

```
EXEC SQL CONNECT TO TOROLAB;  
  
    (execute statements referencing objects at TOROLAB)  
  
EXEC SQL CONNECT TO SVLLAB;  
  
    (execute statements referencing objects at SVLLAB)
```

Example 2: Connect to a remote server specifying a userid and password, perform work for the user and then connect as another user to perform further work.

```
EXEC SQL CONNECT TO SVLLAB USER :AUTHID USING :PASSWORD;  
  
    (execute SQL statements accessing data on the server)  
  
EXEC SQL COMMIT;  
  
    (set AUTHID and PASSWORD to new values)  
  
EXEC SQL CONNECT TO SVLLAB USER :AUTHID USING :PASSWORD;  
  
    (execute SQL statements accessing data on the server)
```

Example 3: User JOE wants to connect to TOROLAB3 and execute SQL statements under the user ID ANONYMOUS which has a password of SHIBBOLETH. The RDB directory entry for TOROLAB3 specifies *IP for the connection type.

Before running the application, some setup must be done.

This command will be required to allow server security information to be retained in OS/400, if it has not been previously run:

```
CHGSYSVAL SYSVAL(QRETSVRSEC) VALUE('1')
```

This command adds the required server authorization entry:

```
ADDSVRAUTE USRPRF(JOE) SERVER(TOROLAB3) USRID(ANONYMOUS) +  
    PASSWORD(SHIBBOLETH)
```

This statement, run under JOE's user profile, will now make the desired connection:

```
EXEC SQL CONNECT TO TOROLAB3;  
  
    (execute statements referencing objects at TOROLAB3)
```

CREATE ALIAS

The CREATE ALIAS statement defines an alias on a table, view, or member of a database file at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - *USE to the Create DDM File (CRTDDMF) command
 - *EXECUTE, *READ and *ADD to the library into which the alias is created
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the alias is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

Syntax

```

▶▶ CREATE ALIAS alias-name FOR table-name | view-name | (member-name)

```

Description

alias-name

Names the alias. The name, including the implicit or explicit qualifier, must not be the same as an index, table, view, alias or file that already exists at the current server.

If SQL names were specified, the alias will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the alias will be created in the schema that is specified by the qualifier. If not qualified, the alias will be created in the same schema as the table or view for which the alias was created. If the table is not qualified and does not exist at the time the alias is created, the alias will be created in the current library (*CURLIB).

If the alias name is not a valid system name, DB2 UDB for iSeries will generate a system name. For information on the rules for generating a name, see “Rules for Table Name Generation” on page 553.

FOR *table-name* or *view-name*

Identifies the table or view at the current server for which the alias is to be defined. An alias name cannot be specified (an alias cannot refer to another alias).

CREATE ALIAS

The *table-name* or *view-name* need not identify a table or view that exists at the time the alias is created. If the table or view does not exist when the alias is created, a warning is returned. If the table or view does not exist when the alias is used, an error is returned.

If SQL names were specified and the *table-name* or *view-name* was not qualified, then the qualifier is the implicit qualifier. For more information, see “Naming Conventions” on page 45.

If system names were specified and the *table-name* or *view-name* is not qualified and does not exist when the alias is created, the *table-name* or *view-name* is qualified by the library in which the alias is created.

member-name

Identifies a member of a database file.

If a member is specified, you can only use the alias in data manipulation (DML) SQL statements. If a member name is not specified, *FIRST is used.

Notes

The Override Database File (OVRDBF) CL command allows the database manager to process individual members of a database file. Creating an alias over a member of a database file, however, is easier and performs better by eliminating the need to perform the override.

An alias can be defined to reference either the system name or SQL name. However, since system names are generated during create processing, it is recommended that the SQL name be specified.

Alias attributes: An alias is created as a special form of a DDM file.

An alias created over a distributed table is only created on the current server. For more information about distributed tables, see the DB2 Multisystem book.

Alias ownership: If SQL names were specified, the *owner* of the alias is the user profile with the same name as the schema into which the alias is created. Otherwise, the *owner* of the alias is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the alias is the user profile or group user profile of the job executing the statement.

Alias authority: If SQL names are used, aliases are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, aliases are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the alias is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the alias.

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword SYNONYM can be used as a synonym for ALIAS.

Examples

Example 1: Create an alias named CURRENT_PROJECTS for the PROJECT table.

```
CREATE ALIAS CURRENT_PROJECTS  
FOR PROJECT
```

Example 2: Create an alias named SALES_JANUARY on the JANUARY member of the SALES table. The sales table has 12 members (one for each month of the year).

```
CREATE ALIAS SALES_JANUARY  
FOR SALES(JANUARY)
```

CREATE DISTINCT TYPE

The CREATE DISTINCT TYPE statement defines a distinct type at the current server. A distinct type is always sourced on one of the built-in data types. Successful execution of the statement also generates:

- A function to cast from the distinct type to its source type
- A function to cast from the source type to its distinct type
- As appropriate, support for the use of comparison operators with the distinct type.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The system authorities *EXECUTE, *READ and *ADD to the library into which the distinct type is created, and
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSTYPES catalog table:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

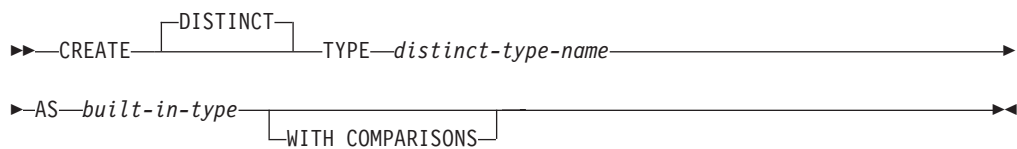
The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

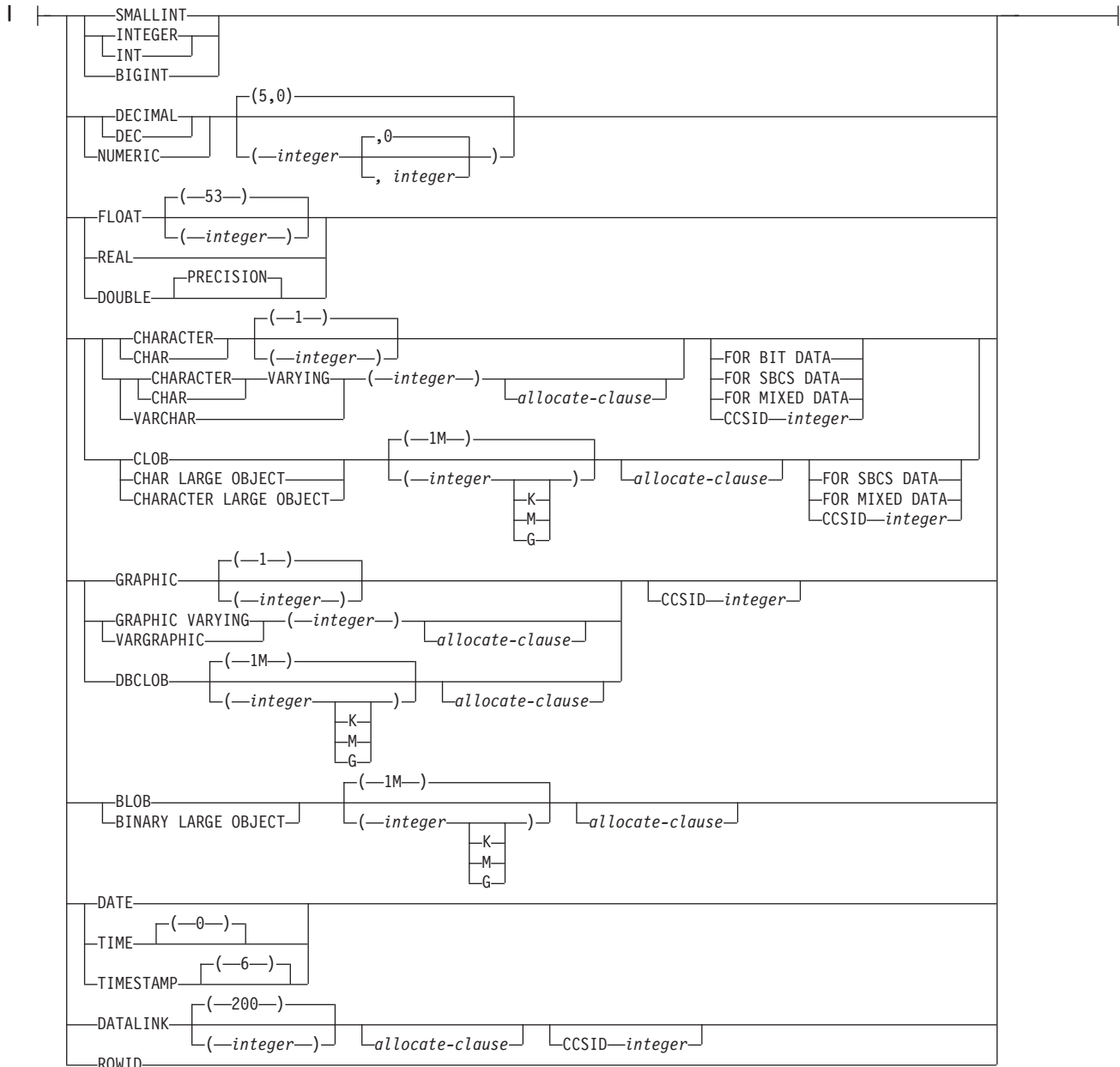
If SQL names are specified and a user profile exists that has the same name as the library into which the distinct type is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

Syntax



built-in-type:



Description

distinct-type-name

Names the distinct type. The name, including the implicit or explicit qualifier, must not be the same as a distinct type that already exists at the current server.

If SQL names were specified, the distinct type will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the distinct type will be created in the schema that is specified by the qualifier. If not qualified, the distinct type will be created in the current library (*CURLIB).

CREATE DISTINCT TYPE

If the distinct type name is not a valid system name, DB2 UDB for iSeries will generate a system name. For information on the rules for generating a name, see "Rules for Table Name Generation" on page 553.

distinct-type-name must not be the name of a built-in data type, or any of the following system-reserved keywords even if you specify them as delimited identifiers.

=	<	>	>=
<=	<>	~=	~<
~<	!=	!<	!>
ALL	FALSE	ONLY	TABLE
AND	FOR	OR	THEN
ANY	FROM	OVERLAPS	TRIM
BETWEEN	IN	PARTITION	TRUE
BOOLEAN	IS	POSITION	TYPE
CASE	LIKE	RRN	UNIQUE
CAST	MATCH	SELECT	UNKNOWN
CHECK	NODENAME	SIMILAR	WHEN
DISTINCT	NODENUMBER	SOME	
EXCEPT	NOT	STRIP	
EXISTS	NULL	SUBSTRING	

If a qualified *distinct-type-name* is specified, the schema name cannot be QSYS, QSYS2, QTEMP, or SYSIBM.

built-in-data-type

Specifies the built-in data type used as the basis for the internal representation of the distinct type. See "CREATE TABLE" on page 525 for a more complete description of each built-in data type.

For portability of applications across platforms, use the following recommended data type names:

- DOUBLE or REAL instead of FLOAT.
- DECIMAL instead of NUMERIC.

If you do not specify a specific value for the data types that have length, precision, or scale attributes, the default attributes of the data type as shown in the syntax diagram are implied.

If the distinct type is sourced on a string data type, a CCSID is associated with the distinct data type at the time the distinct type is created. For more information about data types, see "CREATE TABLE" on page 525.

WITH COMPARISONS

Specifies that system-generated comparison functions are to be created for comparing two instances of the distinct type. WITH COMPARISONS is the default. Comparison functions will be generated for all source types with the exception of a DATALINK whether or not WITH COMPARISONS is specified.⁴⁹ For compatibility with other DB2 products, WITH COMPARISONS should be specified.

The comparison functions do not support the LIKE predicate. In order to use the LIKE predicate on a distinct type, it must be cast to a built-in type.

⁴⁹ Service programs are not created for these comparison functions. These comparison functions are not registered in the SYSROUTINES catalog table.

Notes

Additional Generated Functions: The successful execution of the CREATE DISTINCT TYPE statement causes the database manager to generate the following cast functions:

- 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 DOUBLE to the distinct type if the source type is REAL
- 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.

The cast functions are created as if the following statements were executed (except that the service programs are not created, so you cannot grant or revoke privileges to these functions):

```
CREATE FUNCTION source-type-name (distinct-type-name)
RETURNS source-type-name
```

```
CREATE FUNCTION distinct-type-name (source-type-name)
RETURNS distinct-type-name
```

Names of the Generated Cast Functions: Table 44 on page 432 contains details about the generated cast functions. The unqualified name of the cast function that converts from the distinct type to the source type is the name of the source data type.

In cases in which a length, precision, or scale is specified for the source data type in the CREATE DISTINCT TYPE statement, the unqualified name of the cast function that converts from the distinct type to the source type is simply the name of the source data type. The data type of the value that the cast function returns includes any length, precision, or scale values that were specified for the source data type on the CREATE DISTINCT TYPE statement.

The name of the cast function that converts from the source type to the distinct type is the name of the distinct type. The input parameter of the cast function has the same data type as the source data type, including the length, precision, and scale.

The cast functions that are generated are created in the same schema as that of the distinct type. A function with the same name and same function signature must not already exist in the current server.

For example, assume that a distinct type named T_SHOESIZE is created with the following statement:

```
CREATE DISTINCT TYPE CLAIRE.T_SHOESIZE AS VARCHAR(2) WITH COMPARISONS
```

When the statement is executed, the database manager also generates the following cast functions. VARCHAR converts from the distinct type to the source type, and T_SHOESIZE converts from the source type to the distinct type.

```
FUNCTION CLAIRE.VARCHAR (CLAIRE.T_SHOESIZE) RETURNS VARCHAR(2)
```

```
FUNCTION CLAIRE.T_SHOESIZE (VARCHAR(2)) RETURNS CLAIRE.T_SHOESIZE
```

CREATE DISTINCT TYPE

Notice that function VARCHAR returns a value with a data type of VARCHAR(2) and that function T_SHOESIZE has an input parameter with a data type of VARCHAR(2).

A generated cast function cannot be explicitly dropped. The cast functions that are generated for a distinct type are implicitly dropped when the distinct type is dropped with the DROP statement.

For each built-in data type that can be the source data type for a distinct type, the following table gives the names of the generated cast functions, the data types of the input parameters, and the data types of the values that the functions returns.

Table 44. CAST Functions on Distinct Types

Source Type Name	Function Name	Parameter Type	Return Type
SMALLINT	<i>distinct-type-name</i>	SMALLINT	<i>distinct-type-name</i>
	<i>distinct-type-name</i>	INTEGER	<i>distinct-type-name</i>
	SMALLINT	<i>distinct-type-name</i>	SMALLINT
INTEGER	<i>distinct-type-name</i>	INTEGER	<i>distinct-type-name</i>
	INTEGER	<i>distinct-type-name</i>	INTEGER
BIGINT	<i>distinct-type-name</i>	BIGINT	<i>distinct-type-name</i>
	BIGINT	<i>distinct-type-name</i>	BIGINT
DECIMAL	<i>distinct-type-name</i>	DECIMAL(p,s)	<i>distinct-type-name</i>
	DECIMAL	<i>distinct-type-name</i>	DECIMAL(p,s)
NUMERIC	<i>distinct-type-name</i>	NUMERIC(p,s)	<i>distinct-type-name</i>
	NUMERIC	<i>distinct-type-name</i>	NUMERIC(p,s)
REAL or FLOAT(n) where n <= 24	<i>distinct-type-name</i>	REAL	<i>distinct-type-name</i>
	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	REAL	<i>distinct-type-name</i>	REAL
DOUBLE or DOUBLE PRECISION or FLOAT(n) where n > 24	<i>distinct-type-name</i>	DOUBLE	<i>distinct-type-name</i>
	DOUBLE	<i>distinct-type-name</i>	DOUBLE
CHAR	<i>distinct-type-name</i>	CHAR(n)	<i>distinct-type-name</i>
	CHAR	<i>distinct-type-name</i>	CHAR(n)
	<i>distinct-type-name</i>	VARCHAR(n)	<i>distinct-type-name</i>
VARCHAR	<i>distinct-type-name</i>	VARCHAR(n)	<i>distinct-type-name</i>
	VARCHAR	<i>distinct-type-name</i>	VARCHAR(n)
CLOB	<i>distinct-type-name</i>	CLOB(n)	<i>distinct-type-name</i>
	CLOB	<i>distinct-type-name</i>	CLOB(n)
GRAPHIC	<i>distinct-type-name</i>	GRAPHIC(n)	<i>distinct-type-name</i>
	GRAPHIC	<i>distinct-type-name</i>	GRAPHIC(n)
	<i>distinct-type-name</i>	VARGRAPHIC(n)	<i>distinct-type-name</i>
VARGRAPHIC	<i>distinct-type-name</i>	VARGRAPHIC(n)	<i>distinct-type-name</i>
	VARGRAPHIC	<i>distinct-type-name</i>	VARGRAPHIC(n)

Table 44. CAST Functions on Distinct Types (continued)

Source Type Name	Function Name	Parameter Type	Return Type
DBCLOB	<i>distinct-type-name</i>	DBCLOB(n)	<i>distinct-type-name</i>
	DBCLOB	<i>distinct-type-name</i>	DBCLOB(n)
BLOB	<i>distinct-type-name</i>	BLOB(n)	<i>distinct-type-name</i>
	BLOB	<i>distinct-type-name</i>	BLOB(n)
DATE	<i>distinct-type-name</i>	DATE	<i>distinct-type-name</i>
	DATE	<i>distinct-type-name</i>	DATE
TIME	<i>distinct-type-name</i>	TIME	<i>distinct-type-name</i>
	TIME	<i>distinct-type-name</i>	TIME
TIMESTAMP	<i>distinct-type-name</i>	TIMESTAMP	<i>distinct-type-name</i>
	TIMESTAMP	<i>distinct-type-name</i>	TIMESTAMP
DATALINK	<i>distinct-type-name</i>	DATALINK	<i>distinct-type-name</i>
	DATALINK	<i>distinct-type-name</i>	DATALINK
ROWID	<i>distinct-type-name</i>	ROWID	<i>distinct-type-name</i>
	ROWID	<i>distinct-type-name</i>	ROWID
Notes:			
* Conversion is only supported for UCS-2 graphic.			
Only a DATALINK can be cast to a DATALINK type.			

NUMERIC and FLOAT are not recommended when creating a distinct type for a portable application. DECIMAL and DOUBLE should be used instead.

Distinct type attributes: A distinct type is created as a *SQLUDT object.

Distinct type ownership: If SQL names were specified, the *owner* of the distinct type is the user profile with the same name as the schema into which the distinct type is created. Otherwise, the *owner* of the distinct type is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the distinct type is the user profile or group user profile of the job executing the statement.

Distinct type authority: If SQL names are used, distinct types are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, distinct types are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the distinct type is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the distinct type.

Built-in Functions: 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, and so on) are automatically supported for the distinct type. A built-in function can be used on a distinct type only after a sourced user-defined function, which is based on the

CREATE DISTINCT TYPE

built-in function, has been created for the distinct type. See “Extending or Overriding a Built-in Function” under “CREATE FUNCTION” on page 435.

The schema name of the distinct type must be included in the distinct type for successful use of these operators and cast functions in SQL statements.

Examples

Example 1: Create a distinct type named SHOESIZE that is sourced on the built-in INTEGER data type.

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

The successful execution of this statement also generates two cast functions. Function INTEGER(SHOESIZE) returns a value with data type INTEGER, and function SHOESIZE(INTEGER) returns a value with distinct type SHOESIZE.

Example 2: Create a distinct type named MILES that is sourced on the built-in DOUBLE data type.

```
CREATE DISTINCT TYPE MILES  
AS DOUBLE WITH COMPARISONS
```

The successful execution of this statement also generates two cast functions. Function DOUBLE(MILES) returns a value with data type DOUBLE, and function MILES(DOUBLE) returns a value with distinct type MILES.

Example 3: Create a distinct type T_DEPARTMENT that is sourced on the built-in CHAR data type.

```
CREATE DISTINCT TYPE CLAIRE.T_DEPARTMENT AS CHAR(3)  
WITH COMPARISONS
```

The successful execution of this statement also generates three cast functions:

- Function CLAIRE.CHAR takes a T_DEPARTMENT as input and returns a value with data type CHAR(3).
- Function CLAIRE.T_DEPARTMENT takes a CHAR(3) as input and returns a value with distinct type T_DEPARTMENT.
- Function CLAIRE.T_DEPARTMENT takes a VARCHAR(3) as input and returns a value with distinct type T_DEPARTMENT.

CREATE FUNCTION

The CREATE FUNCTION statement defines a user-defined function at the current server. The following types of functions can be defined:

- External Scalar

The function is written in a programming language such as C or Java and returns a scalar value. The external program is referenced by a function defined at the current server along with various attributes of the function. See “CREATE FUNCTION (External Scalar)” on page 439.

- External Table

The function is written in a programming language such as C or Java and returns a set of rows. The external program is referenced by a function defined at the current server along with various attributes of the function. See “CREATE FUNCTION (External Table)” on page 455.

- Sourced

The function is implemented by invoking another function (built-in, external, sourced, or SQL) that already exists at the current server. A sourced function can return a scalar result, or the result of a column function. See “CREATE FUNCTION (Sourced)” on page 469. The function inherits attributes of the underlying source function.

- SQL Scalar

The function is written exclusively in SQL and returns a scalar value. The function body is defined at the current server along with various attributes of the function. See “CREATE FUNCTION (SQL Scalar)” on page 478.

- SQL TABLE

The function is written exclusively in SQL and returns a set of rows. The function body is defined at the current server along with various attributes of the function. See “CREATE FUNCTION (SQL Table)” on page 487.

Notes

Choosing the Schema and Function Name: If a qualified function name is specified, the *schema-name* cannot be QSYS2, QSYS, QTEMP, or SYSIBM. If *function-name* is not qualified, it is implicitly qualified with the default schema name.

The unqualified function name must not be one of the following names reserved for system use even they are specified as delimited identifiers:

=	<	>	>=
<=	<>	~=	~<
~<	!=	!>	!<
ALL	FALSE	ONLY	TABLE
AND	FOR	OR	THEN
ANY	FROM	OVERLAPS	TRIM
BETWEEN	IN	PARTITION	TRUE
BOOLEAN	IS	POSITION	TYPE
CASE	LIKE	RRN	UNIQUE
CAST	MATCH	SELECT	UNKNOWN
CHECK	NODENAME	SIMILAR	WHEN
DISTINCT	NODENUMBER	SOME	
EXCEPT	NOT	STRIP	
EXISTS	NULL	SUBSTRING	

CREATE FUNCTION

Defining the Parameters: The input parameters for the function are specified as a list within parenthesis.

The maximum number of parameters allowed in CREATE FUNCTION is 90.

A function can have no input parameters. In this case, an empty set of parenthesis must be specified, for example:

```
CREATE FUNCTION WOOFER()
```

The data type of the result of the function is specified in the RETURNS clause for the function.

- **Choosing Data Types for Parameters:** When choosing the data types of the input and result parameters for a function, the rules of promotion that can affect the values of the parameters need to be considered. See “Promotion of Data Types” on page 75. For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, we recommend using the following data types for parameters:

- INTEGER instead of SMALLINT
- DOUBLE instead of REAL
- VARCHAR instead of CHAR
- VARGRAPHIC instead of GRAPHIC

For portability of functions across platforms that are not DB2 UDB for iSeries, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.
- **Specifying AS LOCATOR for a Parameter:** Passing a locator instead of a value can result in fewer bytes being passed in or out of the function. This can be useful when the value of the parameter is very large. The AS LOCATOR clause specifies that a locator to the value of the parameter is passed instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

AS LOCATOR can not be specified for SQL functions.

Determining the Uniqueness of Functions in a Schema: The same name can be used for more than one function in a schema if the function signature of each function is unique. The function signature is the qualified function name combined with the number and data types of the input parameters. The combination of name, schema name, the number of parameters, and the data type each parameter (without regard for other attributes such as length, precision, scale, or CCSID) must not identify a user-defined function that exists at the current server. The return type has no impact on the determining uniqueness of a function. Two different schemas can each contain a function with the same name that have the same data types for all of their corresponding data types. However, a schema must not contain two functions with the same name that have the same data types for all of their corresponding data types.

When determining whether corresponding data types match, the database manager does not consider any length, precision, or scale attributes in the comparison. The database manager considers the synonyms of data types a match. For example, REAL and FLOAT, and DOUBLE and FLOAT) are considered a match. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), and DECIMAL(4,3). Furthermore, the character and graphic types are considered to be the same. For example, the following are considered to be the same type: CHAR and GRAPHIC, VARCHAR and VARGRAPHIC, and CLOB and DBCLOB. CHAR(13) and GRAPHIC(8) are considered to be the same type. An error is returned if the signature of the function being created is a duplicate of a signature for an existing user-defined function with the same name and schema.

Assume that the following statements are executed to create four functions in the same schema. The second and fourth statements fail because they create functions that are duplicates of the functions that the first and third statements created.

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

Specifying a Specific Name for a Function: When defining multiple functions with the same name and schema (with different parameter lists), it is recommended that a specific name also be specified. The specific name can be used to uniquely identify the function such as when sourcing on this function, dropping the function, or commenting on the function. However, the function cannot be invoked by its specific name.

The specific name is implicitly or explicitly qualified with a schema name. If a schema name is not specified on CREATE FUNCTION, it is the same as the explicit or implicit schema name of the function name (*function-name*). If a schema name is specified, it must be the same as the explicit or implicit schema name of the function name. The name, including the schema name must not identify the specific name of another function or procedure that exists at the current server.

If the SPECIFIC clause is not specified, a specific name is generated.

Extending or Overriding a Built-in Function: Giving a user-defined function the same name as a built-in function is not a recommended practice unless the functionality of the built-in function needs to be extended or overridden.

- **Extending the Functionality of Existing Built-in Functions:**

Create the new user-defined function with the same name as the built-in function, and a unique function signature. For example, a user-defined function similar to the built-in function ROUND that accepts the distinct type MONEY as input rather than the built-in numeric types might be necessary. In this case, the signature for the new user-defined function named ROUND is different from all the function signatures supported by the built-in ROUND function.

- **Overriding a Built-in Function:**

Create the new user-defined function with the same name and signature as an existing built-in function. The new function has the same name and data type as the corresponding parameters of the built-in function but implements different logic. For example, a user-defined function similar to the built-in function ROUND that uses different rules for rounding than the built-in ROUND function might be necessary. In this case, the signature for the new user-defined function named ROUND will be the same as a signature that is supported by the built-in ROUND function.

CREATE FUNCTION

Once a built-in function has been overridden, an application that uses the unqualified function name and was previously successful using the built-in function of that name might fail, or perhaps even worse, appear to run successfully but provide a different result if the user-defined function is chosen by the data base manager rather than the built-in function.

Special Registers in Functions: The settings of the special registers of the invoker are inherited by the function on invocation and restored upon return to the invoker.

CREATE FUNCTION (External Scalar)

This CREATE FUNCTION (External Scalar) statement creates an external scalar function at the current server. A user-defined external scalar function returns a single value each time it is invoked.

Invocation

This statement can be embedded in an application program, or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization id of the statement must include at least one of the following:

- For the SYSFUNCS catalog view and SYSPARMS catalog table⁵⁰:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative Authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

If the external program or service program exists, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the external program or service program that is referenced in the SQL statement:
 - The system authority *EXECUTE on the library that contains the external program or service program.
 - The system authority *EXECUTE on the external program or service program, and
 - The system authority *CHANGE on the program or service program. The system needs this authority to update the program object to contain the information necessary to save/restore the function to another system. If user does not have this authority, the function is still created, but the program object is not updated.
- Administrative Authority

If SQL names are specified and a user profile exists that has the same name as the library into which the function is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:

50. The GRTOBJAUT CL command must be used to grant these privileges.

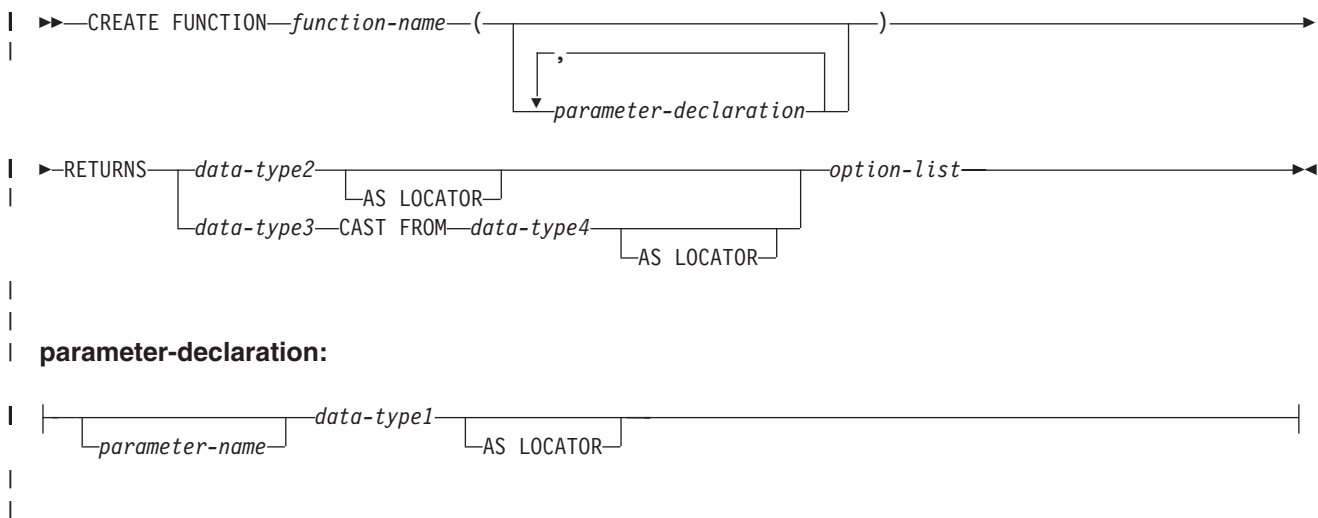
CREATE FUNCTION (External Scalar)

- The USAGE privilege on the distinct type, and
- The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

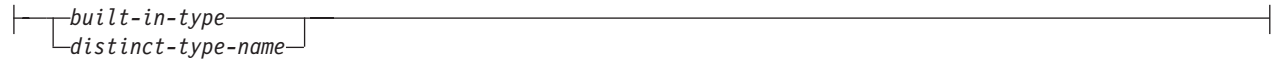
The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

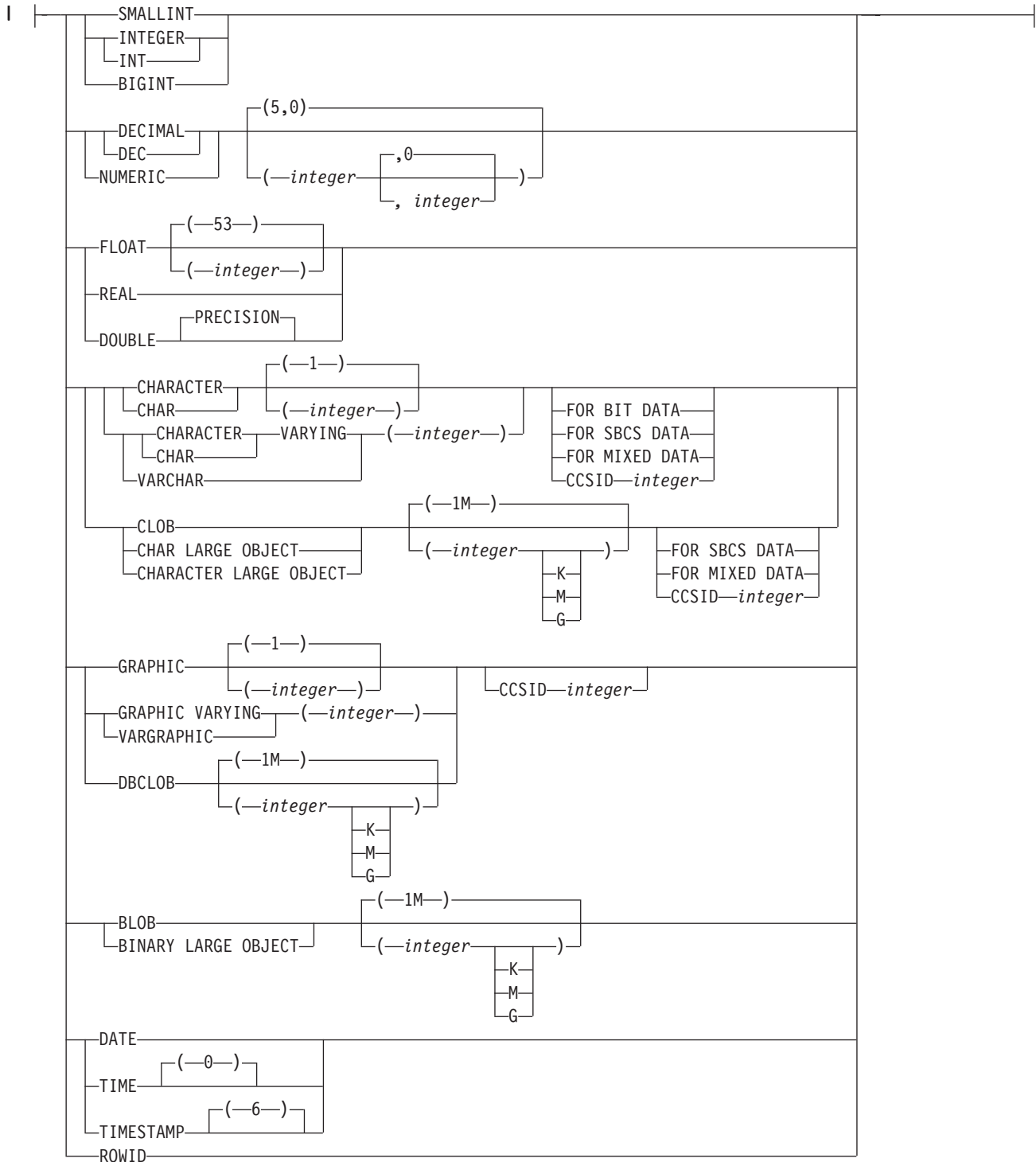
Syntax



data-type1, data-type2, data-type3, data-type4:

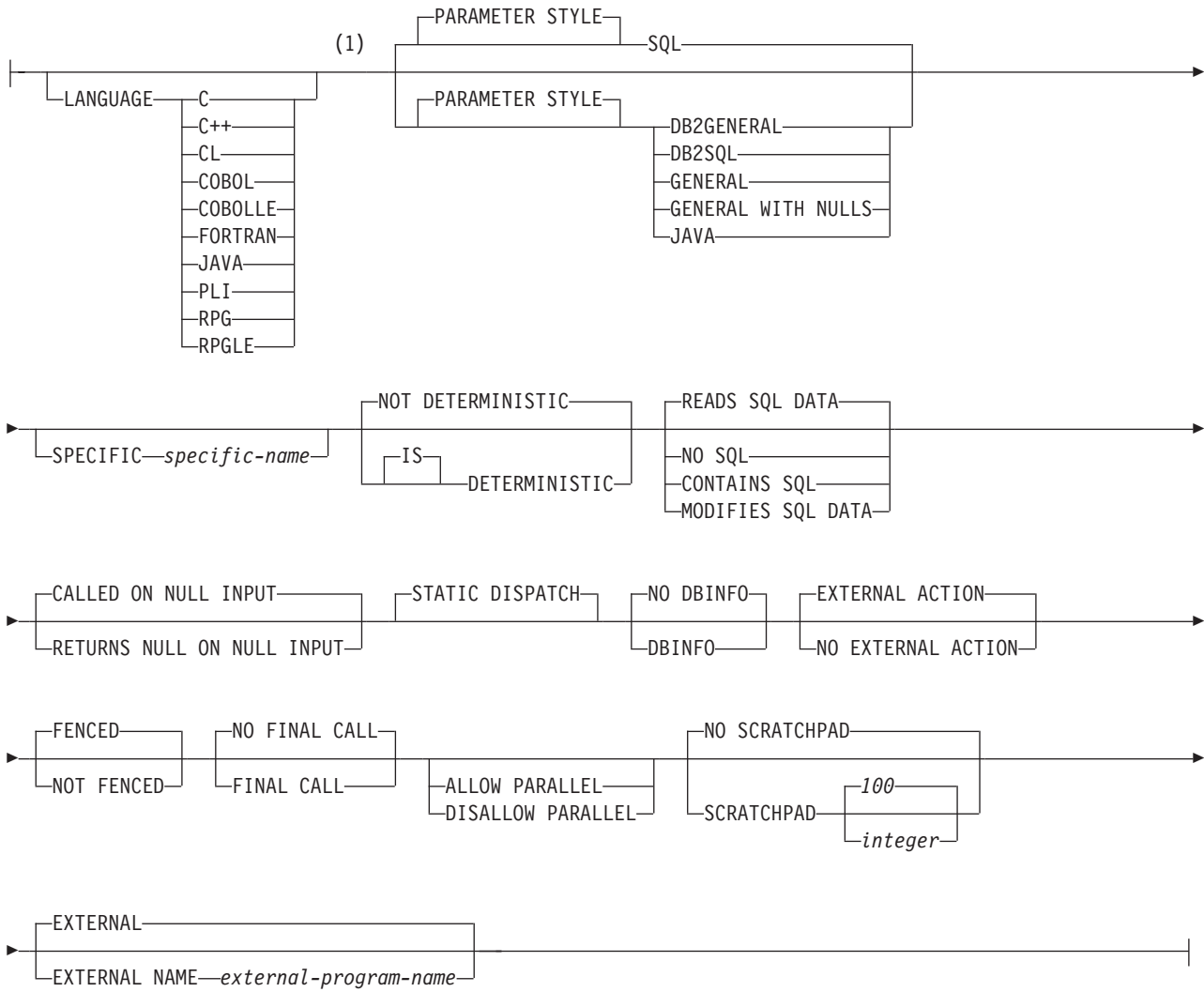


built-in-type:



CREATE FUNCTION (External Scalar)

option-list:



Notes:

- 1 The optional clauses can be specified in a different order.

Description

function-name

Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified, the function will be created in the current library (*CURLIB). If there is no current library, the function will be created in QGPL.

CREATE FUNCTION (External Scalar)

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see “Choosing the Schema and Function Name” on page 435.

(parameter-declaration,...)

Specifies the number of input parameters of the function and the data type of each parameter. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in CREATE FUNCTION is 90. For external functions created with PARAMETER STYLE SQL, the input and result parameters specified and the implicit parameters for indicators, SQLSTATE, function name, specific name, and message text, as well as any optional parameters are included. The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program that is used to compile the external program.

parameter-name

Specifies the name of the input parameter. Do not specify the same name more than once.

data-type1

Specifies the number of input parameters of the function and the data type of each input parameter. All the parameters for a function are input parameters. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

AS LOCATOR

Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. See “Specifying AS LOCATOR for a parameter” in “CREATE FUNCTION” on page 435 for more information.

RETURNS

Specifies the output of the function.

data-type2

Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, LONG VARGRAPHIC, or DataLink) or a distinct type (that is not based on a DataLink).

If a CCSID is specified,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in that CCSID.
- If AS LOCATOR is specified and the CCSID of the data the locator points to is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified,

CREATE FUNCTION (External Scalar)

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in the CCSID of the job (or associated graphic CCSID of the job for graphic string return values).
- If AS LOCATOR is specified, the data the locator points to is converted to the CCSID of the job, if the CCSID of the data the locator points to is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 13488 (UCS-2 graphic string data).

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. Specify AS LOCATOR only if the result of the function has a LOB data type or a distinct type based on a LOB data type. See “Specifying AS LOCATOR for a parameter” in “CREATE FUNCTION” on page 435 for more information.

data-type3 CAST FROM *data-type4*

Specifies the data type and attributes of the output (*data-type4*) and the data type in which that output is returned to the invoking statement (*data-type3*). The two data types can be different. For example, for the following definition, the function returns a CHAR(10) value, which the database manager converts to a DATE value and then passes to the statement that invoked the function:

```
CREATE FUNCTION GET_HIRE_DATE (CHAR6)
  RETURNS DATE CAST FROM CHAR(10)
```

The value of *data-type4* must not be a distinct type and must be castable to *data-type3*. The value for *data-type3* can be any built-in data type or distinct type. (For information on casting data types, see “Casting Between Data Types” on page 77).

For CCSID information, see the description of *data-type2* above.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. Specify AS LOCATOR only if the result of the function has a LOB data type or a distinct type based on a LOB data type. See “Specifying AS LOCATOR for a parameter” in “CREATE FUNCTION” on page 435 for more information.

SPECIFIC *specific-name*

Specifies a unique name for the function. See “Specifying a Specific Name for a Function” in “CREATE FUNCTION” on page 435.

LANGUAGE

Specifies the language interface convention to which the function body is written. All programs must be designed to run in the server’s environment.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program at the time the function is created. The language of the program is assumed to be C if:

- The program attribute information associated with the program does not identify a recognizable language
- The program cannot be found

C

The external program is written in C.

CREATE FUNCTION (External Scalar)

C++

The external program is written in C++.

CL

The external program is written in CL or ILE CL.

COBOL

The external program is written in COBOL.

COBOLLE

The external program is written in ILE COBOL.

FORTRAN

The external program is written in FORTRAN.

JAVA

The external program is written in JAVA. The database manager will call the user-defined function, which must be a public static method of the specified Java class

PLI

The external program is written in PL/I.

RPG

The external program is written in RPG.

RPGLE

The external program is written in ILE RPG.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the function is deterministic.

NOT DETERMINISTIC

Specifies that the function will not always return the same result from successive function invocations with identical input arguments. NOT DETERMINISTIC should be specified if the function contains a reference to a special register or a non-deterministic function.

DETERMINISTIC

Specifies that the function will always return the same result from successive invocations with identical input arguments.

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL

Specifies whether the function can execute any SQL statements and, if so, what type. The database manager verifies that the SQL issued by the function is consistent with this specification. See Appendix C, "Characteristics of SQL Statements" on page 835 for a detailed list of the SQL statements that can be executed under each data access indication.

CONTAINS SQL

The function does not execute SQL statements that read or modify data.

NO SQL

The function does not execute SQL statements.

READS SQL DATA

The function does not execute SQL statements that modify data.

MODIFIES SQL DATA

The function can execute any SQL statement except those statements that are not supported in any function.

CREATE FUNCTION (External Scalar)

FENCED or NOT FENCED

Specifies whether the function will run in the same thread as the invoking SQL statement or in a separate thread.

FENCED

The function will run in a separate thread.

NOT FENCED

The function may run in the same thread as the invoking SQL statement. NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON INPUT

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

CALLED ON NULL INPUT

Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function contains an external action.

EXTERNAL ACTION

The function performs some external action (outside the scope of the function program). Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

NO EXTERNAL ACTION

The function does not perform an external action. It need not be called with each successive function invocation.

This parameter implies that the function

SCRATCHPAD

Specifies whether the function requires a static memory area.

SCRATCHPAD *integer*

Specifies that the function requires a persistent memory area of length *integer*. The integer can range from 1 to 16,000,000. If the memory area is not specified, the size of the area is 100 bytes. If parameter style DB2SQL is specified, a pointer is passed following the required parameters that points to a static storage area. If PARALLEL is specified, a memory area is allocated for each user-defined function reference in the statement. If DISALLOW PARALLEL is specified, only 1 memory area will be allocated for the function.

The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

CREATE FUNCTION (External Scalar)

```
SELECT A, UDFX(A)
FROM TABLEB
WHERE UDFX(A) > 103 OR UDFX(A) < 19
```

If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the SQL statement. Specify the `DISALLOW PARALLEL` clause for functions that will not work correctly with parallelism.

`SCRATCHPAD` is only allowed with `PARAMETER STYLE DB2SQL` or `PARAMETER STYLE DB2GENERAL`.

NO SCRATCHPAD

Specifies that the function does not require a persistent memory area.

FINAL CALL

Specifies whether the function requires special call indication. If `PARAMETER STYLE DB2SQL` is specified and `FINAL CALL` is specified, an additional parameter is passed to the function indicating first call, normal call, or final call.

NO FINAL CALL

Specifies that a final call is not made to the function.

FINAL CALL

Specifies that a final call is made to the function. To differentiate between final calls and other calls, the function receives an additional argument that specifies the type of call.

`FINAL CALL` is only allowed with `PARAMETER STYLE DB2SQL` or `PARAMETER STYLE DB2GENERAL`.

The types of calls are:

First Call

Specifies the first call to the function for this reference to the function in this SQL statement. A first call is a normal call. SQL arguments are passed and the function is expected to return a result.

Normal Call

Specifies that SQL arguments are passed and the function is expected to return a result.

Final Call

Specifies the last call to the function to enable the function to free resources. A final call is not a normal call. If an error occurs, the database manager attempts to make the final call.

A final call occurs at these times:

- *End of statement*: When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of a parallel task*: When the function is executed by parallel tasks.
- *End of transaction*: When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

CREATE FUNCTION (External Scalar)

Some functions that use a final call can receive incorrect results if parallel tasks execute the function. For example, if a function sends a note for each final call to it, one note is sent for each parallel task instead of once for the function. Specify the **DISALLOW PARALLEL** clause for functions that have inappropriate actions when executed in parallel.

If a commit operation occurs while a cursor defined as **WITH HOLD** is open, a final call is made when the cursor is closed or the application ends. If a commit occurs at the end of a parallel task, a final call is made regardless of whether a cursor defined as **WITH HOLD** is open.

Commitable operations should not be performed during a **FINAL CALL**, because the **FINAL CALL** may occur during a close invoked as part of a **COMMIT** operation.

PARALLEL

Specifies whether the function can be run in parallel.

ALLOW PARALLEL

Specifies that the function can be run in parallel.

DISALLOW PARALLEL

Specifies that the function cannot be run in parallel.

The default is **DISALLOW PARALLEL**, if you specify one or more of the following clauses:

- **NOT DETERMINISTIC**
- **EXTERNAL ACTION**
- **FINAL CALL**
- **MODIFIES SQL DATA**
- **SCRATCHPAD**

Otherwise, **ALLOW PARALLEL** is the default.

DBINFO

Specifies whether or not the function requires the database information be passed.

DBINFO

Specifies that the database manager should pass a structure containing status information to the function. Table 45 contains a description of the **DBINFO** structure. Detailed information about the **DBINFO** structure can be found in include file **SQLUDF** in **QSYSINC.H**.

DBINFO is only allowed with **PARAMETER STYLE DB2SQL** or **PARAMETER STYLE DB2GENERAL**.

Table 45. *DBINFO* fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The run-time authorization ID.

Table 45. DBINFO fields (continued)

Field	Data Type	Description
CCSID Information	INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. The following information identifies the CCSID: <ul style="list-style-type: none"> • SBCS CCSID • DBCS CCSID • Mixed CCSID • Indication of which of the first three CCSIDs is appropriate. • Reserved <p>If a CCSID is not explicitly specified for a parameter on the CREATE FUNCTION statement, the input string is assumed to be encoded in the CCSID of the job at the time the function is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external function will be converted before calling the external program.</p>
Target column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	If a user-defined function is specified on the right-hand side of a SET clause in an UPDATE statement, the following information identifies the target column: <ul style="list-style-type: none"> • Schema name • Base table name • Column name <p>If the user-defined function is not on the right-hand side of a SET clause in an UPDATE statement, these fields are blank.</p>
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.

NO DBINFO

Specifies that the function does not require the database information to be passed.

STATIC DISPATCH

Specifies that the function is dispatched statically. All functions are statically dispatched.

EXTERNAL NAME *external-program-name*

Specifies the program, service program, or java class that will be executed when the function is invoked in an SQL statement. The name must identify a program, service program, or java class that exists at the server at the time the function is invoked. If the naming option is *SYS and the name is not qualified, the current path will be used to search for the program or service program at the time the function is invoked.

The validity of the name is checked at the server. If the format of the name is not correct, an error is returned.

If external-program-name is not specified, the external program name is assumed to be the same as the function name.

The program, service program, or java class need not exist at the time the function is created, but it must exist at the time the function is invoked.

A CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statement is not allowed in the external program of the function.

CREATE FUNCTION (External Scalar)

PARAMETER STYLE

Specifies the conventions used for passing parameters to and returning the values from functions:

SQL

All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- A parameter for the result of the function.
- N parameters for indicator variables for the input parameters.
- A parameter for the indicator variable for the result.
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the function. The SQLSTATE returned is an SQLSTATE that is assigned by the external program.

The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.

- A VARCHAR(517) input parameter for the fully qualified function name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(70) output parameter for the message text.

When control is returned to the invoking program, the message text can be found in the 6th token of the SQLERRMC field of the SQLCA. Only a portion of the message text is available. The information on the layout of the message data in the SQLERRMC, see the replacement data descriptions for message SQL0443 in message file QSQLMSG.

For more information about the parameters passed, see the include `sqludf` in the appropriate source file. For example, for C, `sqludf` can be found in `QSYSINC/H`.

DB2GENERAL

This parameter style is 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. All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- A parameter for the result of the function.

DB2GENERAL is only allowed when the LANGUAGE is JAVA.

GENERAL

All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.

Note that the result is returned through as a value of a value returning function. For example:

```
return_val func(parameter-1, parameter-2, ...)
```

GENERAL is only allowed when EXTERNAL NAME identifies a service program.

CREATE FUNCTION (External Scalar)

GENERAL WITH NULLS

All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- An additional argument is passed for an indicator variable array.
- A parameter for the indicator variable for the result.

Note that the result is returned through as a value of a value returning function. For example:

```
return_val func(parameter-1, parameter-2, ...)
```

GENERAL WITH NULLS is only allowed when EXTERNAL NAME identifies a service program.

JAVA

This parameter style is specifies a parameter passing convention that conforms to the Java language and SQLJ routines specification. All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.

Note that the result is returned through as a value of a value returning function. For example:

```
return_val func(parameter-1, parameter-2, ...)
```

JAVA is only allowed when the LANGUAGE is JAVA.

DB2SQL

All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- A parameter for the result of the function.
- N parameters for indicator variables for the input parameters.
- A parameter for the indicator variable for the result.
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the function. The SQLSTATE returned either be:
 - the SQLSTATE from the last SQL statement executed in the external program,
 - an SQLSTATE that is assigned by the external program.The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.
- A VARCHAR(517) input parameter for the fully qualified function name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(70) output parameter for the message text.
- Zero to three optional parameters:
 - A structure (consisting of an INTEGER followed by a CHAR(n)) input and output parameter for the scratchpad, if SCRATCH PAD was specified on the CREATE FUNCTION statement.

CREATE FUNCTION (External Scalar)

- An INTEGER input parameter for the call type, if FINAL CALL was specified on the CREATE FUNCTION statement.
- A structure for the dbinfo structure, if DBINFO was specified on the CREATE FUNCTION statement.

For more information about the parameters passed, see the include `sqludf` in the appropriate source file. For example, for C, `sqludf` can be found in `QSYSINC/H`.

Note that language of the external function determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see the SQL Programming Concepts book.

Notes

General Considerations for Defining User-defined Functions: See "CREATE FUNCTION" on page 435 for general information on defining user-defined functions.

Creating the Function: When an external function associated with an ILE external program or service program is created, an attempt is made to save the function's attributes in the associated program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

The attributes can be saved for external functions subject to the following restrictions:

- The external program library must not be SYSIBM, QSYS, or QSYS2.
- The external program must exist when the CREATE FUNCTION statement is issued.
- The external program must be an ILE *PGM or *SRVPGM object.
- The external program or service program must contain at least one SQL statement.

If the object cannot be updated, the function will still be created.

During restore of the function:

- If the specific name was specified when the function was originally created and it is not unique, an error is issued.
- If the specific name was not specified, a unique name is generated if necessary.
- If the signature is not unique, the function cannot be registered, and an error is issued.

Invoking the Function: When an external function is invoked, it runs in whatever activation group was specified when the external program or service program was created. However, ACTGRP(*CALLER) should normally be used so that the function runs in the same activation group as the calling program. ACTGRP(*NEW) is not allowed.

Notes for Java Functions: To be able to run Java functions, you must have the Developer Kit for Java (5722-JV1) installed on your system. Otherwise, an SQLCODE of -443 will be returned and a CPDB521 message will be placed in the job log.

CREATE FUNCTION (External Scalar)

If an error occurs while running a Java procedure, an SQLCODE of -443 will be returned. Depending on the error, other messages may exist in the job log of the job where the procedure was run.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords SIMPLE CALL can be used as a synonym for GENERAL.
- The value DB2GENRL may be used as a synonym for DB2GENERAL.

Examples

Example 1: Assume an external function program in C is needed that implements the following logic:

```
result = 2 * input - 4
```

The function should return a null value if and only if one of the input arguments is null. The simplest way to avoid a function call and get a null result when an input value is null is to specify RETURNS NULL ON NULL INPUT on the CREATE FUNCTION statement. The following statement defines the function, using the specific name MINENULL1.

```
CREATE FUNCTION NTEST1 (SMALLINT)
  RETURNS SMALLINT
  EXTERNAL NAME NTESTMOD
  SPECIFIC MINENULL1
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  FENCED
  PARAMETER STYLE DB2SQL
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION
```

The program code:

```
void nudft1
(int *input,          /* ptr to input arg          */
 int *output,        /* ptr to output arg        */
 short *input_ind,   /* ptr to input indicator   */
 short *output_ind,  /* ptr to output indicator  */
 char sqlstate[6], /* sqlstate                  */
 char fname[140], /* fully qualified function name */
 char finst[129], /* function specific name    */
 char msgtext[71]) /* msg text buffer          */
{
  if (*input_ind == -1)
    *output_ind = -1;
  else
  {
    *output = 2*(*input)-4;
    *output_ind = 0;
  }
  return;
}
```

Example 2: Assume that a user wants to define an external function named CENTER. The function program will be written in C. The following statement

CREATE FUNCTION (External Scalar)

defines the function, and lets the database manager generate a specific name for the function. The name of the program containing the function body is the same as the name of the function, so the EXTERNAL clause does not include 'NAME external-program-name'.

```
CREATE FUNCTION CENTER (INTEGER, FLOAT)
  RETURNS FLOAT
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  PARAMETER STYLE DB2SQL
  NO EXTERNAL ACTION
```

Example 3: Assume that user McBride (who has administrative authority) wants to define an external function named CENTER in the SMITH schema. McBride plans to give the function specific name FOCUS98. The function program uses a scratchpad to perform some one-time only initialization and save the results. The function program returns a value with a DOUBLE data type. The following statement written by user McBride defines the function and ensures that when the function is invoked, it returns a value with a data type of DECIMAL(8,4).

```
CREATE FUNCTION SMITH.CENTER (DOUBLE, DOUBLE, DOUBLE)
  RETURNS DECIMAL(8,4)
  CAST FROM DOUBLE
  EXTERNAL NAME CMOD
  SPECIFIC FOCUS98
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  FENCED
  PARAMETER STYLE DB2SQL
  NO EXTERNAL ACTION
  SCRATCHPAD
  NO FINAL CALL
```

Example 4: The following example defines a Java user-defined function that returns the position of the first vowel in a string. The user-defined function is written in Java, is to be run fenced, and is the FINDVWL method of class JAVAUDFS.

```
CREATE FUNCTION FINDV (VARCHAR(32000))
  RETURNS INTEGER
  FENCED
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'JAVAUDFS.FINDVWL'
  NO EXTERNAL ACTION
  CALLED ON NULL INPUT
  DETERMINISTIC
  NO SQL
```

CREATE FUNCTION (External Table)

This CREATE FUNCTION (External Table) statement creates an external table function at the current server. The function returns a result table.

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

You can embed this statement in an application program, or you can issue this statement interactively. 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:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative Authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

If the external program or service program exists, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the external program or service program that is referenced in the SQL statement:
 - The system authority *EXECUTE on the library that contains the external program or service program.
 - The system authority *EXECUTE on the external program or service program, and
 - The system authority *CHANGE on the program or service program. The system needs this authority to update the program object to contain the information necessary to save/restore the function to another system. If user does not have this authority, the function is still created, but the program object is not updated.
- Administrative Authority

If SQL names are specified and a user profile exists that has the same name as the library into which the function is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:

CREATE FUNCTION (External Table)

- The USAGE privilege on the distinct type, and
- The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

Syntax

►► CREATE FUNCTION *function-name* (*parameter-declaration*)

► RETURNS TABLE (*column-name data-type2*) *option-list*
AS LOCATOR

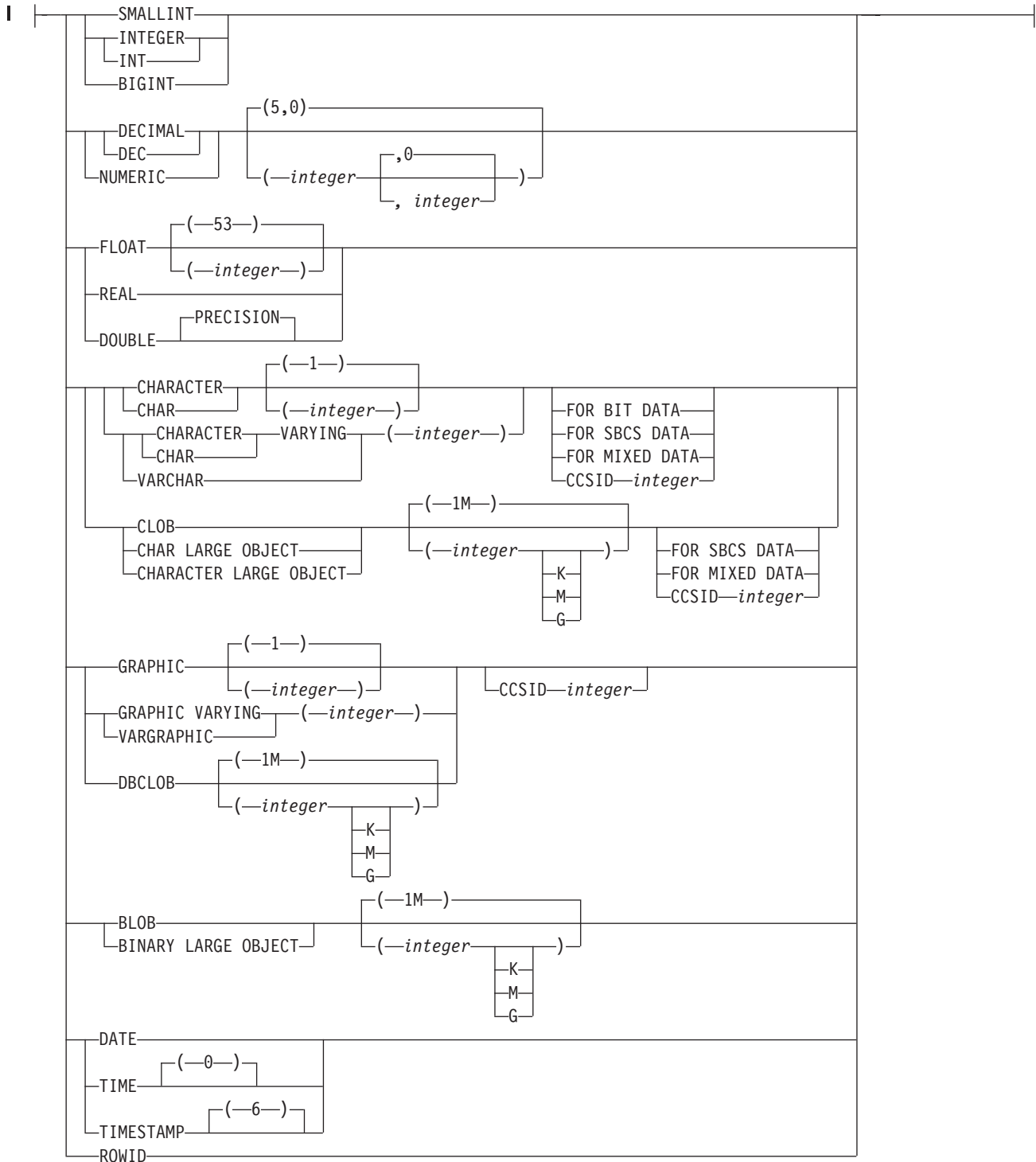
parameter-declaration:

parameter-name *data-type1* AS LOCATOR

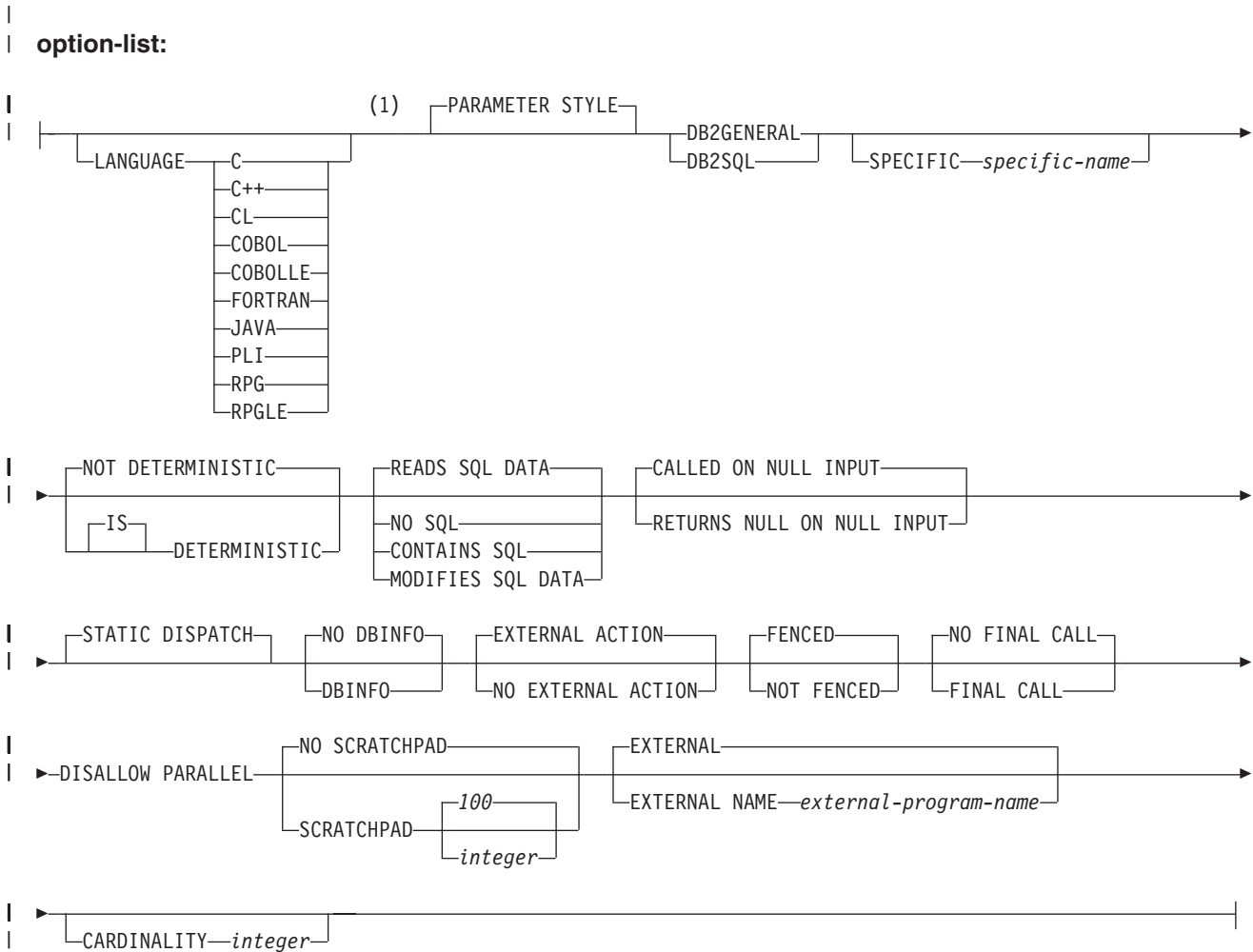
| data-type1, data-type2:



| built-in-type:



CREATE FUNCTION (External Table)



Notes:

- 1 The optional clauses can be specified in a different order.

Description

function-name

Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified, the function will be created in the current library (*CURLIB). If there is no current library, the function will be created in QGPL.

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see "Choosing the Schema and Function Name" on page 435.

CREATE FUNCTION (External Table)

(parameter-declaration,...)

Specifies the number of parameters of the function and the data type of each parameter. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in CREATE FUNCTION (External Table) is 90. The maximum number of parameters may be additionally limited by the maximum number of parameters allowed by the licensed program that is used to compile the external program.

parameter-name

Specifies the name of the input parameter. Do not specify the same name more than once.

data-type1

Specifies the number of input parameters of the function and the data type of each input parameter. All the parameters for a function are input parameters. There must be one entry in the list for each parameter that the function expects to receive. Although not required, you can give each parameter a name.

Parameters with a large object (LOB) data type are not supported when PARAMETER STYLE JAVA is specified.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

AS LOCATOR

Specifies that the input parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the input parameter has a LOB data type or a distinct type based on a LOB data type.

RETURNS TABLE

Specifies the output table of the function.

Assume the number of parameters is N. For PARAMETER STYLE DB2GENERAL, there must be no more than $(255-(N*2))/2$ columns. For PARAMETER STYLE DB2SQL, there must be no more than $(247-(N*2))/2$ columns.

column-name

Specifies the name of a column of the output table. Do not specify the same name more than once.

data-type2

Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, LONG VARGRAPHIC, or DataLink) or a distinct type (that is not based on a DataLink).

If a DATE or TIME is specified, the table function must return the date or time in ISO format.

If a CCSID is specified,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in that CCSID.

CREATE FUNCTION (External Table)

- If AS LOCATOR is specified and the CCSID of the data the locator points to is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified,

- If AS LOCATOR is not specified, the result returned is assumed to be encoded in the CCSID of the job (or associated graphic CCSID of the job for graphic string return values).
- If AS LOCATOR is specified, the data the locator points to is converted to the CCSID of the job, if the CCSID of the data the locator points to is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 13488 (UCS-2 graphic string data).

AS LOCATOR

Specifies that the function returns a locator to the value for the column rather than the actual value. You can specify AS LOCATOR only for a LOB data type or a distinct type based on a LOB data type.

SPECIFIC *specific-name*

Provides a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function or procedure that exists at the current server. If unqualified, the implicit qualifier is the same as the qualifier of the function name. If qualified, the qualifier must be the same as the qualifier of the function name.

If specific name is not specified, it is set to the function name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

LANGUAGE (language clause)

The language clause specifies the language of the external program.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program at the time the function is created. The language of the program is assumed to be C if:

- The program attribute information associated with the program does not identify a recognizable language
- The program cannot be found

C

The external program is written in C.

C++

The external program is written in C++.

CL

The external program is written in CL or ILE CL.

COBOL

The external program is written in COBOL.

COBOLLE

The external program is written in ILE COBOL.

FORTRAN

The external program is written in FORTRAN.

CREATE FUNCTION (External Table)

JAVA

The external program is written in JAVA. The database manager will call the user-defined function as a method in a Java class.

PLI

The external program is written in PL/I.

RPG

The external program is written in RPG.

RPGLE

The external program is written in ILE RPG.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the function is deterministic.

NOT DETERMINISTIC

Specifies that the function will not always return the same result from successive function invocations with identical input arguments. NOT DETERMINISTIC should be specified if the function contains a reference to a special register or a non-deterministic function.

DETERMINISTIC

Specifies that the function will always return the same result from successive invocations with identical input arguments.

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL

Specifies whether the function can execute any SQL statements and, if so, what type. The database manager verifies that the SQL issued by the function is consistent with this specification. See Appendix C, "Characteristics of SQL Statements" on page 835 for a detailed list of the SQL statements that can be executed under each data access indication.

CONTAINS SQL

The function does not execute SQL statements that read or modify data.

NO SQL

The function does not execute SQL statements.

READS SQL DATA

The function does not execute SQL statements that modify data.

MODIFIES SQL DATA

The function can execute any SQL statement except those statements that are not supported in any function.

FENCED or NOT FENCED

Specifies whether the function will run in the same thread as the invoking SQL statement or in a separate thread.

FENCED

The function will run in a separate thread. FENCED is the safest option if the function contains SQL cursors, because multiple invocations of the same function in the same SQL statement may conflict with each other.

NOT FENCED

The function may run in the same thread as the invoking SQL statement. NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function.

CREATE FUNCTION (External Table)

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON INPUT

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

CALLED ON NULL INPUT

Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function contains an external action.

EXTERNAL ACTION

The function performs some external action (outside the scope of the function program). Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

NO EXTERNAL ACTION

The function does not perform an external action. It need not be called with each successive function invocation.

SCRATCHPAD

Specifies whether the function requires a static memory area.

SCRATCHPAD *integer*

Specifies that the function requires a persistent memory area of length *integer*. The integer can range from 1 to 16,000,000. If the memory area is not specified, the size of the area is 100 bytes. If parameter style DB2SQL is specified, a pointer is passed following the required parameters that points to a static storage area. Only 1 memory area will be allocated for the function.

The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, two scratchpads are allocated for the two references to UDFX in the following SQL statement:

```
SELECT A.C1, B.C1
FROM TABLE(UDFX(:hv1)) AS A, TABLE(UDFX(:hv1)) AS B
```

NO SCRATCHPAD

Specifies that the function does not require a persistent memory area.

FINAL CALL

Specifies whether the function requires a final call (and a separate first call). For table functions, the call-type argument is ALWAYS present, regardless of which FINAL CALL option is chosen. The call-type argument indicates first call, open call, fetch call, close call, or final call.

FINAL CALL

Specifies that the function requires a final call (and a separate first call). It also controls when the scratchpad is re-initialized. If NO FINAL CALL is specified, then the database manager 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

CREATE FUNCTION (External Table)

call can be made to the table function. Specifies that a final call is made to the function. To differentiate between final calls and other calls, the function receives an additional argument that specifies the type of call.

The types of calls are:

First Call

Specifies the first call to the function for this reference to the function in this SQL statement.

Open Call

Specifies a call to open the table function result in this SQL statement.

Fetch Call

Specifies a call to fetch a row from the table function in this SQL statement.

Close Call

Specifies a call to close the table function result in this SQL statement.

Final Call

Specifies the last call to the function to enable the function to free resources. If an error occurs, the database manager attempts to make the final call.

A final call occurs at these times:

- *End of statement:* When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of transaction:* When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made when the cursor is closed or the application ends.

Commitable operations should not be performed during a FINAL CALL, because the FINAL CALL may occur during a close invoked as part of a COMMIT operation.

NO FINAL CALL

Specifies that the function does not require a final call (and a separate first call). However the open, fetch, and close calls are still made.

DISALLOW PARALLEL

Specifies that the function cannot be run in parallel. Table functions cannot run in parallel.

DBINFO

Specifies whether or not the function requires the database information be passed.

DBINFO

Specifies that the database manager should pass a structure containing status information to the function. Table 46 on page 464 contains a description of the DBINFO structure. Detailed information about the DBINFO structure can be found in include file SQLUDF in QSYSINC.H.

CREATE FUNCTION (External Table)

Table 46. DBINFO fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The run-time authorization ID.
CCSID Information	INTEGER INTEGER INTEGER INTEGER CHAR(8)	<p>The CCSID information of the job. The following information identifies the CCSID:</p> <ul style="list-style-type: none"> • SBCS CCSID • DBCS CCSID • Mixed CCSID • Indication of which of the first three CCSIDs is appropriate. • Reserved <p>If a CCSID is not explicitly specified for a parameter on the CREATE FUNCTION statement, the input string is assumed to be encoded in the CCSID of the job at the time the function is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external function will be converted before calling the external program.</p>
Target column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	<p>If a user-defined function is specified on the right-hand side of a SET clause in an UPDATE statement, the following information identifies the target column:</p> <ul style="list-style-type: none"> • Schema name • Base table name • Column name <p>If the user-defined function is not on the right-hand side of a SET clause in an UPDATE statement, these fields are blank.</p>
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.
Number of table function column list entries	SMALLINT	The number of non-zero entries in the table function column list specified in the "Table function column list" field below.
Reserved	CHAR(24)	Reserved for future use.
Table function column list	Pointer (16 Bytes)	<p>This field is a pointer to an array of short integers which is dynamically allocated by the database manager. Only the first n entries, where n is specified in the "Number of table function column list entries" field, are of interest, n may be equal to 0, and is less than or equal to the number of result columns defined for the function in the RETURNS TABLE clause. The values correspond to the ordinal numbers of the columns which this statement needs from the table function. A value of 1 means the first defined result column, 2 means the second defined result column, and so on. The values may be in any order. Note that n could be equal to zero for a statement that is similar to SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ, where no actual column values are needed by the query.</p> <p>This array represents an opportunity for optimization. The function need not return all values for all the result columns of the table function. Only a subset of the values may be needed in a particular context, and these are the columns identified (by number) in the array. Since this optimization may complicate the function logic, the function can choose to return every defined column.</p>

NO DBINFO

Specifies that the function does not require the database information to be passed.

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 2 147 483 647 inclusive.

If the CARDINALITY clause is not specified for a table function, the database manager will assume a finite value as a default.

Warning: If a function does in fact have infinite cardinality, i.e. it returns a row every time it is called and never returns the end-of-table condition, then queries which require the end-of-table condition will also 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.

STATIC DISPATCH

Specifies that the function is dispatched statically. All functions are statically dispatched.

EXTERNAL NAME *external-program-name*

Specifies the program, service program, or java class that will be executed when the function is invoked in an SQL statement. The name must identify a program, service program, or java class that exists at the server at the time the function is invoked. If the naming option is *SYS and the name is not qualified, the current path will be used to search for the program or service program at the time the function is invoked.

The validity of the name is checked at the server. If the format of the name is not correct, an error is returned.

If external-program-name is not specified, the external program name is assumed to be the same as the function name.

The program, service program, or java class need not exist at the time the function is created, but it must exist at the time the function is invoked.

A CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statement is not allowed in the external program of the function.

PARAMETER STYLE

Specifies the conventions used for passing parameters to and returning the values from functions:

DB2GENERAL

This parameter style is 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. All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- The next M parameters are the result columns of the function that are specified on the RETURNS TABLE clause.

DB2GENERAL is only allowed when the LANGUAGE is JAVA.

CREATE FUNCTION (External Table)

DB2SQL

All applicable parameters are passed. The parameters are defined to be in the following order:

- The first N parameters are the input parameters that are specified on the CREATE FUNCTION statement.
- The next M parameters are the result columns of the function that are specified on the RETURNS TABLE clause.
- N parameters for indicator variables for the input parameters.
- M parameters for the indicator variables of the result columns of the function that are specified on the RETURNS TABLE clause
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the function. The SQLSTATE returned either be:
 - the SQLSTATE from the last SQL statement executed in the external program,
 - an SQLSTATE that is assigned by the external program.The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.
- A VARCHAR(517) input parameter for the fully qualified function name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(70) output parameter for the message text.
- A structure (consisting of an INTEGER followed by a CHAR(n)) input and output parameter for the scratchpad, if SCRATCH PAD was specified on the CREATE FUNCTION statement.
- An INTEGER input parameter for the call type.
- A structure for the dbinfo structure, if DBINFO was specified on the CREATE FUNCTION statement.

For more information about the parameters passed, see the include sqludf in the appropriate source file. For example, for C, sqludf can be found in QSYSINC/H.

Note that language of the external function determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see the SQL Programming Concepts book.

Notes

General Considerations for Defining User-defined Functions: See “CREATE FUNCTION” on page 435 for general information on defining user-defined functions.

Creating the Function: When an external function associated with an ILE external program or service program is created, an attempt is made to save the function’s attributes in the associated program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

The attributes can be saved for external functions subject to the following restrictions:

- The external program library must not be SYSIBM, QSYS, or QSYS2.

CREATE FUNCTION (External Table)

- The external program must exist when the CREATE FUNCTION statement is issued.
- The external program must be an ILE *PGM or *SRVPGM object.
- The external program or service program must contain at least one SQL statement.

If the object cannot be updated, the function will still be created.

During restore of the function:

- If the specific name was specified when the function was originally created and it is not unique, an error is issued.
- If the specific name was not specified, a unique name is generated if necessary.
- If the signature is not unique, the function cannot be registered, and an error is issued.

Invoking the Function: When an external function is invoked, it runs in whatever activation group was specified when the external program or service program was created. However, ACTGRP(*CALLER) should normally be used so that the function runs in the same activation group as the calling program. ACTGRP(*NEW) is not allowed.

Notes for Java Functions: To be able to run Java functions, you must have the Developer Kit for Java (5722-JV1) installed on your system. Otherwise, an SQLCODE of -443 will be returned and a CPDB521 message will be placed in the job log.

If an error occurs while running a Java procedure, an SQLCODE of -443 will be returned. Depending on the error, other messages may exist in the job log of the job where the procedure was run.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The value DB2GENRL may be used as a synonym for DB2GENERAL.

Example

The following creates a table function written to return a row consisting of a single document identifier column for each known document in a text management system. The first parameter matches a given subject area and the second parameter contains a given string.

Within the context of a single session, the 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 optimizer.

CREATE FUNCTION (External Table)

```
|          CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
|          RETURNS TABLE (DOCID CHAR(16))
|          EXTERNAL NAME 'MYLIB/RAJIV(UDFMATCH) '
|          LANGUAGE C
|          PARAMETER STYLE DB2SQL
|          NO SQL
|          DETERMINISTIC
|          NO EXTERNAL ACTION
|          NOT FENCED
|          SCRATCHPAD
|          FINAL CALL
|          DISALLOW PARALLEL
|          CARDINALITY 20
|
```


CREATE FUNCTION (Sourced)

This CREATE FUNCTION (Sourced) statement is used to create a user-defined function, based on another existing scalar or column function, at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization id of the statement must include at least one of the following:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

If the source function is a user-defined function, the authorization ID of the statement must include at least one of the following for the source function:

- The EXECUTE privilege on the function
- Administrative authority

The authorization ID of the statement has the EXECUTE privilege on a function when:

- It is the owner of the function,
- It has been granted the EXECUTE privilege on the function, or
- It has been granted the system authorities of *OBJOPR and *EXECUTE on the function.

To create a sourced function, the privileges held by the authorization ID of the statement must also include at least one of the following:

- The following system authorities:
 - *USE to the Create Service Program (CRTSRVPGM) command or
 - *USE to the Create Program (CRTPGM) command
 - *EXECUTE and *ADD to the library into which the function is created.
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the function is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

CREATE FUNCTION (Sourced)

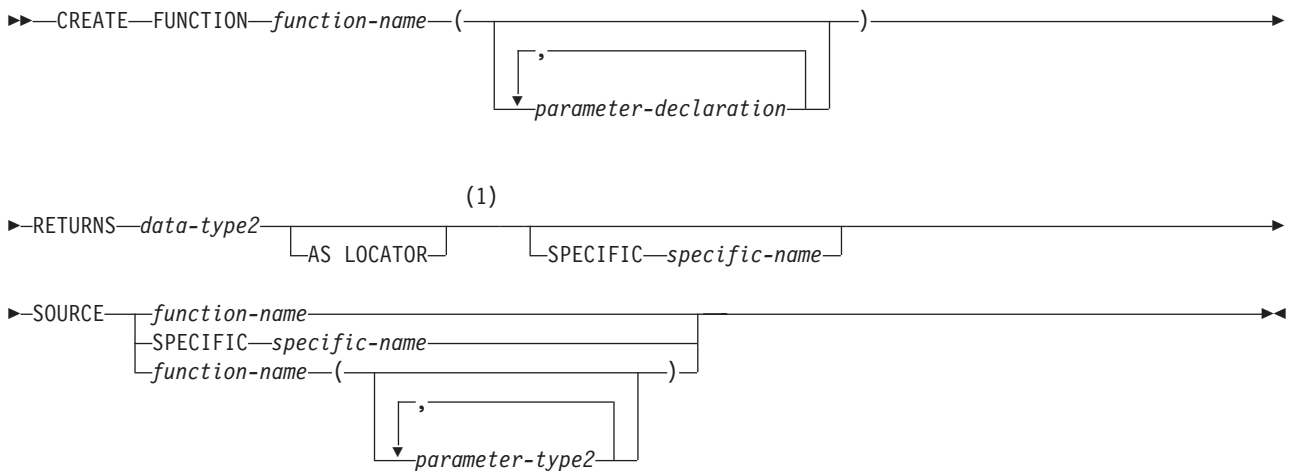
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

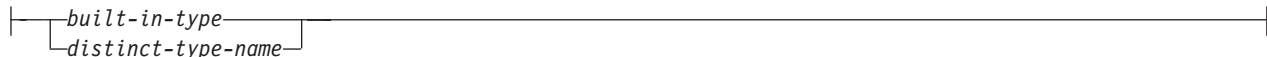
Syntax



parameter-declaration:



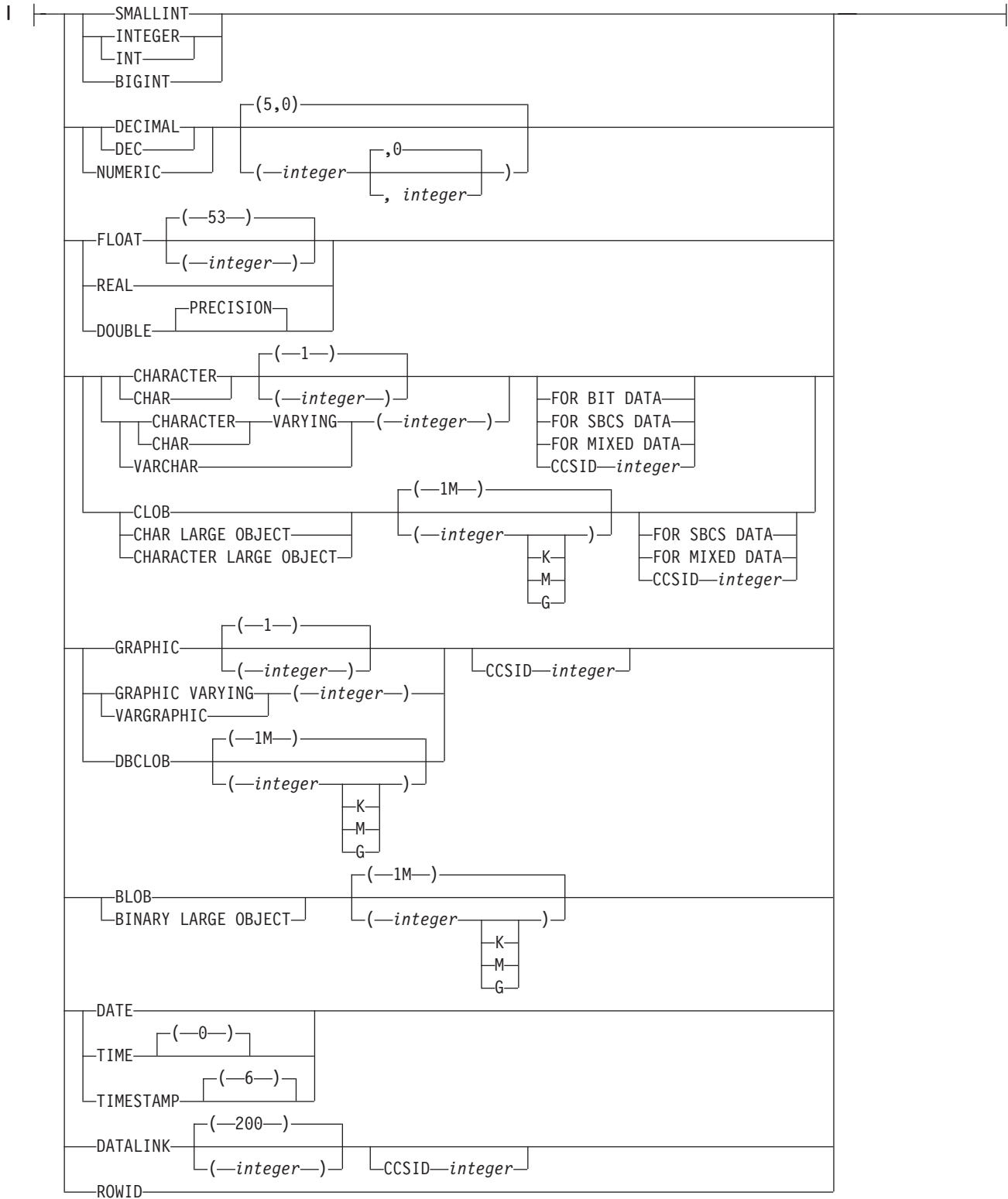
data-type1, data-type2, data-type3:



Notes:

1 The RETURNS, SPECIFIC, and SOURCE clauses can be specified in any order.

built-in-type:



CREATE FUNCTION (Sourced)

parameter-type:

|—*data-type3*—|
|—AS LOCATOR—|

Description

function-name

Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in schema that is specified by the qualifier. If no qualifier is specified, the function will be created in the current library (*CURLIB). If there is no current library, the function will be created in QGPL.

If the function is sourced on an existing function to enable the use of the existing function with a distinct type, the name can be the same name as the existing function. In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see “Choosing the Schema and Function Name” on page 435.

(*parameter-declaration*,...)

Specifies the number of input parameters of the function and the data type of each parameter. Each *parameter-declaration* is an input parameter for the function. A maximum of 90 parameters can be specified.

parameter-name

Specifies the name of the parameter. Although not required, a parameter name can be specified for each parameter. Each name in the parameter list must not be the same as any other name.

data-type1

Specifies the number of input parameters of the function and the data type of each input parameter. The data type can be a built-in data type or a distinct data type.

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 information see “Casting Between Data Types” on page 77). However, this checking does not guarantee that an error will not occur when the function is invoked. See “**Considerations for invoking a sourced user-defined function**” under “Notes” for this statement.

built-in-type

The data type of the input parameter is a built-in data type. See “CREATE TABLE” on page 525 for a more complete description of each built-in data type.

distinct-type-name

The data type of the input parameter is a distinct type. The length, precision, or scale attributes for the parameter are those of the source

CREATE FUNCTION (Sourced)

type of the distinct type (those specified on CREATE DISTINCT TYPE). See “CREATE DISTINCT TYPE” on page 428 for more information.

If the name of the distinct type is specified without a schema name, the database manager resolves the schema name by searching the schemas in the SQL path.

DataLinks are not allowed for functions sourced on external functions.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

AS LOCATOR

Specifies that the input parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the input parameter has a LOB data type or a distinct type based on a LOB data type.

RETURNS

Specifies the output of the function.

data-type

Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, LONG VARGRAPHIC, or a DataLink) or distinct type (that is not based on a DataLink), provided it is castable from the result type of the source function. (For information on casting data types, see “Casting Between Data Types” on page 77)

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. You can specify AS LOCATOR only if the output from the function has a LOB data type or a distinct type based on a LOB data type. The AS LOCATOR clause is not allowed for functions sourced on SQL functions.

SPECIFIC *specific-name*

Provides a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function or procedure that exists at the current server. If unqualified, the implicit qualifier is the same as the qualifier of the function name. If qualified, the qualifier must be the same as the qualifier of the function name.

If specific name is not specified, it is set to the function name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

SOURCE

Specifies that the new function is being defined is a sourced function. A *sourced function* is implemented by another function (the *source function*). The function must exist at the current server and it must be a function that was defined with the CREATE FUNCTION statement or a cast function that was generated by a CREATE DISTINCT TYPE statement. The particular function can be identified by its name, function signature, or specific name.

The source function can be any built-in scalar or column function except COALESCE, HASH, IFNULL, LAND, LOR, MAX, MIN, NODENAME, NODENUMBER, NULLIF, PARTITION, POSITION, RRN, STRIP, SUBSTRING,

CREATE FUNCTION (Sourced)

TRIM, VALUE, and XOR, or any previously created user-defined function. It can be a system-generated user-defined function (generated when a distinct type was created).

The source function can be one of the following built-in functions only if one argument is specified: BLOB, CHAR, CLOB, DBCLOB, DECIMAL, GRAPHIC, TRANSLATE, VARCHAR, VARGRAPHIC, and ZONED.

If you base the sourced function directly or indirectly on a scalar function, the sourced function inherits the attributes of the scalar function. This can involve several layers of sourced functions. For example, assume that function A is sourced on function B, which in turn is sourced on function C. Function C is a scalar function. Functions A and B inherit all of the attributes that are specified on CREATE FUNCTION statement for function C.

function-name

Identifies the function to be used as the source function by its function name. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned

function-name (parameter-type2, ...)

Identifies the function to be used as the source function by its function signature, which uniquely identifies the function. The *function-name (parameter-type,...)* must identify a function with the specified signature at the current server. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance. Synonyms for data types are considered a match.

If *function-name()* is specified, the function identified must have zero parameters.

To use a built-in function as the source function, this syntax variation must be used.

function-name

Identifies the name of the source function. If an unqualified name is specified, the schemas of the SQL path are searched. Otherwise, the specified schema is searched for the function.

parameter-type,...

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision or scale attribute, you can specify a value or use a set of empty parentheses.

- Empty parenthesis indicates that the data base manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).

CREATE FUNCTION (Sourced)

- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

For data types with a subtype or CCSID attribute, specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If you specify either clause, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified the data type must be a LOB or a distinct type based on a LOB. If AS LOCATOR is specified and a length is explicitly specified, the data type length is ignored.

SPECIFIC *specific-name*

Identifies the function to be used as the source function by its specific name. The *specific-name* must identify a specific function that exists in the specified or implicit schema. If an unqualified *specific-name* is specified, the default schema is used as the qualifier.

The number of input parameters in the function that is being created must be the same as the number of parameters in the source function. If the data type of each input parameter is not the same as or castable to the corresponding parameter of the source function, an error occurs. The data type of the final result of the source function must match or be castable to the result of the sourced function.

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 13488 (UCS-2 graphic string data).

Notes

General Considerations for Defining User-defined Functions: See “CREATE FUNCTION” on page 435 for general information on defining user-defined functions.

Function ownership: If SQL names were specified, the *owner* of the function is the user profile with the same name as the schema into which the function is created. Otherwise, the *owner* of the function is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the function is the user profile or group user profile of the job executing the statement.

CREATE FUNCTION (Sourced)

Function authority: If SQL names are used, functions are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, functions are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the function is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the function.

Considerations for invoking a sourced user-defined function: When a sourced function is invoked, each argument to the function is assigned to the associated parameter defined for the function. The values are then cast (if necessary) to the data type of the corresponding parameters of the underlying function. An error can occur either in the assignment or in the cast. For example: an argument passed on input to a function that matches the data type and length or precision attributes of the parameter for the function might not be castable if the corresponding parameter of the underlying source function has a shorter length or less precision. It is recommended that the data types of the parameters of a sourced function be defined with attributes that are less than or equal to the attributes of the corresponding parameters of the underlying function.

The result of the underlying function is assigned to the RETURNS data type of the sourced function. The RETURNS data type of the underlying function might not be castable to the RETURNS data type of the source function. This can occur when the RETURNS data type of this new source function has a shorter length or less precision than the RETURNS data type of the underlying function. For example, an error would occur when function A is invoked assuming the following functions exist. Function A returns an INTEGER. Function B is a sourced function, is defined to return a SMALLINT, and the definition references function A in the SOURCE clause. It is recommended that the RETURNS data type of a sourced function be defined with attributes that are the same or greater than the attributes defined for the RETURNS data type of the underlying function.

Considerations when the function is based on a user-defined function: If the sourced function is based directly or indirectly on an external scalar function, the sourced function inherits the attributes of the EXTERNAL clause of the external scalar function. This can involve several layers of sourced functions. For example, assume that function A is sourced on function B, which in turn is sourced on function C. Function C is an external scalar function. Functions A and B inherit all of the attributes that are specified on the EXTERNAL clause of the CREATE FUNCTION statement for function C.

Creating the Function: When a sourced function is created, a small service program object is created that represents the function. When this service program is saved and restored to another system, the attributes from the CREATE FUNCTION statement are automatically added to the catalog on that system.

Examples

Example 1: Assume that distinct type HATSIZE is defined and is based on the built-in data type INTEGER. An AVG function could be defined to compute the average hat size of different departments. Create a sourced function that is based on built-in function AVG.

```
CREATE FUNCTION AVG (HATSIZE)
  RETURNS HATSIZE
  SOURCE AVG (INTEGER)
```


CREATE FUNCTION (Sourced)

The syntax of the `SOURCE` clause includes an explicit parameter list because the source function is a built-in function.

When distinct type `HATSIZE` was created, two cast functions were generated, which allow `HATSIZE` to be cast to `INTEGER` for the argument and `INTEGER` to be cast to `HATSIZE` for the result of the function.

Example 2: After Smith created the external scalar function `CENTER` in his schema, there is a need to use this function, function, but the invocation of the function needs to accept two `INTEGER` arguments instead of one `INTEGER` argument and one `DOUBLE` argument. Create a sourced function that is based on `CENTER`.

```
CREATE FUNCTION MYCENTER (INTEGER, INTEGER)  
RETURNS DOUBLE  
SOURCE SMITH.CENTER (INTEGER, DOUBLE);
```

CREATE FUNCTION (SQL Scalar)

CREATE FUNCTION (SQL Scalar)

This CREATE FUNCTION (SQL Scalar) statement creates an SQL function at the current server. The function returns a single result.

Invocation

You can embed this statement in an application program, or you can issue this statement interactively. 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:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative Authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

The privileges held by the authorization ID of the statement must also include at least one of the following:

- The following system authorities:
 - *USE to the Create Service Program (CRTSRVPGM) command or
 - *EXECUTE and *ADD to the library into which the function is created.
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

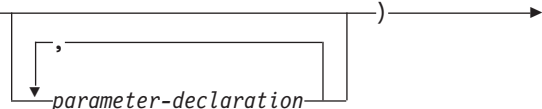
- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

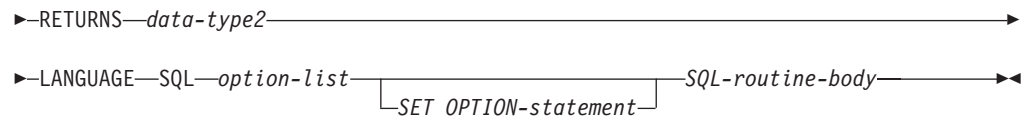
- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

Syntax

```
▶▶ CREATE FUNCTION function-name ( parameter-declaration )
```



CREATE FUNCTION (SQL Scalar)



parameter-declaration:

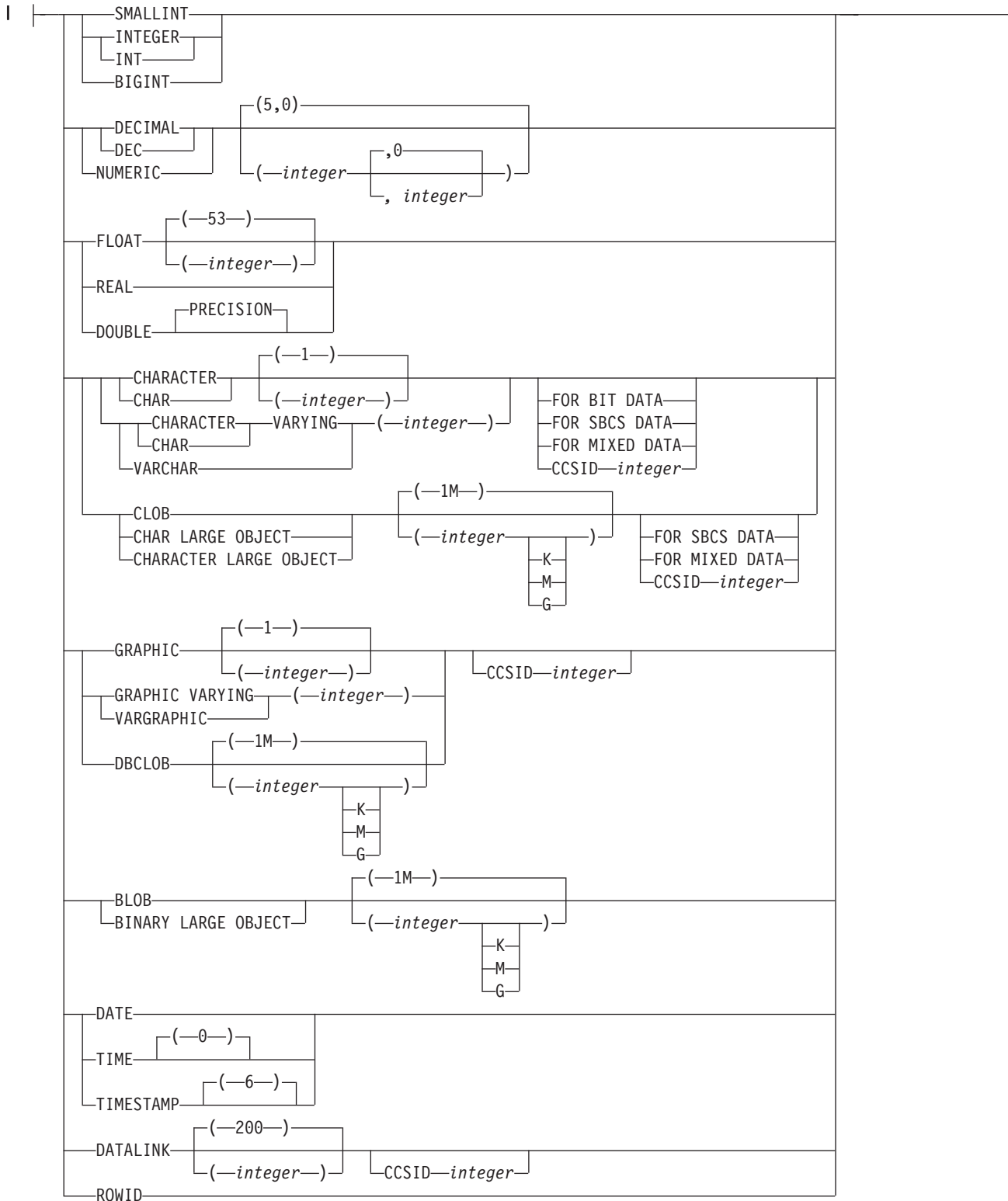


CREATE FUNCTION (SQL Scalar)

data-type:

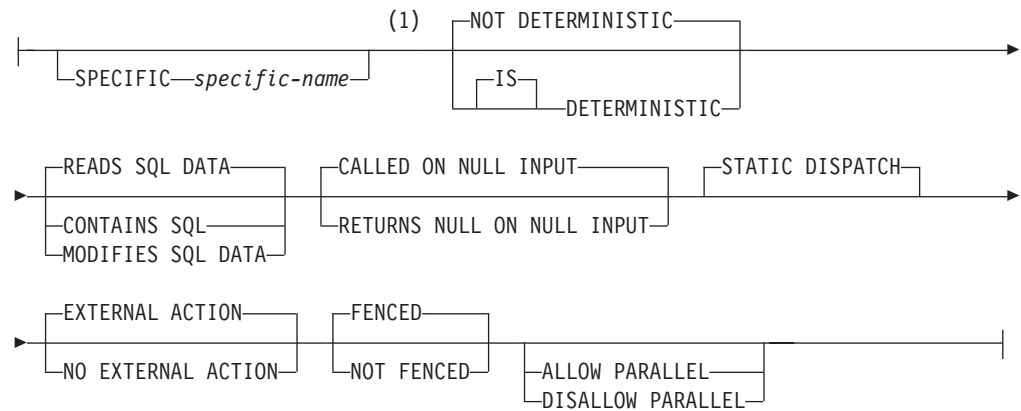


built-in-type:



CREATE FUNCTION (SQL Scalar)

option-list:



Notes:

1 The optional clauses can be specified in a different order.

SQL-routine-body:

|—*SQL-control-statement*—|

Description

function-name

Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified, the function will be created in the current library (*CURLIB). If there is no current library, the function will be created in QGPL.

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see “Choosing the Schema and Function Name” on page 435.

(parameter-declaration,...)

Specifies the number of parameters of the function and the data type of each parameter. Although not required, you can give each parameter a name.

The maximum number of parameters allowed is 90.

parameter-name

Specifies the name of the input parameter. Do not specify the same name more than once. A parameter name must be specified for parameters in SQL functions.

data-type1

Specifies the number of input parameters of the function and the data type

CREATE FUNCTION (SQL Scalar)

of each input parameter. All the parameters for a function are input parameters. There must be one entry in the list for each parameter that the function expects to receive.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

RETURNS

Specifies the output of the function.

data-type2

Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, or LONG VARGRAPHIC) or a distinct type.

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 13488 (UCS-2 graphic string data).

SPECIFIC *specific-name*

Provides a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function or procedure that exists at the current server. If unqualified, the implicit qualifier is the same as the qualifier of the function name. If qualified, the qualifier must be the same as the qualifier of the function name.

If specific name is not specified, it is set to the function name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

LANGUAGE SQL

Specifies that this is an SQL function.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the function is deterministic.

NOT DETERMINISTIC

| Specifies that the function will not always return the same result from
| successive function invocations with identical input arguments. NOT
| DETERMINISTIC should be specified if the function contains a reference to
| a special register or a non-deterministic function.

DETERMINISTIC

| Specifies that the function will always return the same result from
| successive invocations with identical input arguments.

CREATE FUNCTION (SQL Scalar)

CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA

Specifies whether the function can execute any SQL statements and, if so, what type. The database manager verifies that the SQL issued by the function is consistent with this specification. See Appendix C, “Characteristics of SQL Statements” on page 835 for a detailed list of the SQL statements that can be executed under each data access indication.

CONTAINS SQL

The function does not execute SQL statements that read or modify data.

READS SQL DATA

The function does not execute SQL statements that modify data.

MODIFIES SQL DATA

The function can execute any SQL statement except those statements that are not supported in any function.

FENCED or NOT FENCED

Specifies whether the function will run in the same thread as the invoking SQL statement or in a separate thread.

FENCED

The function will run in a separate thread. FENCED is the safest option if the function contains SQL cursors, because multiple invocations of the same function in the same SQL statement may conflict with each other.

NOT FENCED

The function may run in the same thread as the invoking SQL statement. NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON INPUT

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

CALLED ON NULL INPUT

Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function contains an external action.

EXTERNAL ACTION

The function performs some external action (outside the scope of the function program). Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

NO EXTERNAL ACTION

The function does not perform an external action. It need not be called with each successive function invocation.

PARALLEL

Specifies whether the function can be run in parallel.

CREATE FUNCTION (SQL Scalar)

ALLOW PARALLEL

Specifies that the function can be run in parallel.

DISALLOW PARALLEL

Specifies that the function cannot be run in parallel.

The default is DISALLOW PARALLEL, if you specify one or more of the following clauses:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- MODIFIES SQL DATA

Otherwise, ALLOW PARALLEL is the default.

STATIC DISPATCH

Specifies that the function is dispatched statically. All functions are statically dispatched.

SET OPTION-statement

Specifies the options that will be used to create the function. For example, to create a debuggable function, the following statement could be included:

```
SET OPTION DBGVIEW = *STMT
```

For more information, see “SET OPTION” on page 733.

The options CLOSQLCSR, CNULRQD, DFTRDBCOL, DYNDFTCOL, and NAMING are not allowed in the CREATE FUNCTION statement.

SQL-routine-body

Specifies a single SQL statement, including a compound statement. See Chapter 6, “SQL Control Statements” on page 779 for more information about defining SQL functions.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statement is not allowed in a function.

If the *SQL-routine-body* is a compound statement, it must contain at least one RETURN statement and a RETURN statement must be executed when the function is called.

Notes

General Considerations for Defining User-defined Functions: See “CREATE FUNCTION” on page 435 for general information on defining user-defined functions.

Function ownership: If SQL names were specified, the *owner* of the function is the user profile with the same name as the schema into which the function is created. Otherwise, the *owner* of the function is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the function is the user profile or group user profile of the job executing the statement.

Function authority: If SQL names are used, functions are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, functions are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

CREATE FUNCTION (SQL Scalar)

If the owner of the function is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the function.

Creating the Function: When an SQL function is created, the database manager creates a temporary source file that will contain C source code with embedded SQL statements. A *SRVPGM object is then created using the CRTSRVPGM command. The SQL options used to create the service program are the options that are in effect at the time the CREATE FUNCTION statement is executed. The service program is created with ACTGRP(*CALLER).

The specific name is used to determine the name of the source file member and *SRVPGM object. If the specific name is a valid system name, it will be used as the name of member and program. If the member already exists, it will be overlaid. If a program already exists in the specified library, a unique name is generated using the rules for generating system table names. If the specific name is not a valid system name, a unique name is generated using the rules for generating system table names.

The function's attributes are saved in the associated service program object. If the *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

During restore of the function:

- If the specific name was specified when the function was originally created and it is not unique, an error is issued.
- If the specific name was not specified, a unique name is generated if necessary.
- If the signature is not unique, the function cannot be registered, and an error is issued.

Identifier Resolution: If the tables specified in a routine body exist, all references in the routine body of an SQL routine are resolved to identify a particular column, SQL parameter, or SQL variable at the time the SQL routine is created. If the tables do not exist, all names that exist as SQL variables or parameters are resolved to identify the variable or parameter when the function is created. The remaining names are assumed to be columns bound to the tables when the function is invoked.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the function name for parameters, and the label name for SQL variables.

Invoking the Function: When an SQL function is invoked, it runs in the activation group of the calling program.

If a function is specified in the select-list of a select-statement and if the function specifies EXTERNAL ACTION or MODIFIES SQL DATA, the function will only be invoked for each row returned. Otherwise, the UDF may be invoked for rows that are not selected.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.

CREATE FUNCTION (SQL Scalar)

- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.

Example

Define a scalar function that returns the tangent of a value using the existing SIN and COS built-in functions.

```
CREATE FUNCTION TAN
  (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X)
```

Notice that a parameter name (X) is specified for the input parameter to function TAN. The parameter name is used within the body of the function to refer to the input parameter. The invocations of the SIN and COS functions, within the body of the TAN user-defined function, pass the parameter X as input.

CREATE FUNCTION (SQL Table)

This CREATE FUNCTION (SQL table) statement creates an SQL table function at the current server. The function returns a single result table.

Invocation

You can embed this statement in an application program, or you can issue this statement interactively. 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:

- For the SYSFUNCS catalog view and SYSPARMS catalog table:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative Authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

The privileges held by the authorization ID of the statement must also include at least one of the following:

- The following system authorities:
 - *USE to the Create Service Program (CRTSRVPGM) command or
 - *EXECUTE and *ADD to the library into which the function is created.
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

Syntax

CREATE FUNCTION (SQL Table)

▶▶ CREATE FUNCTION *function-name* (*parameter-declaration*)

▶ RETURNS TABLE (*column-name data-type2*)

▶ LANGUAGE SQL *option-list* [SET OPTION *statement*] *SQL-routine-body*

|
|
|

parameter-declaration:

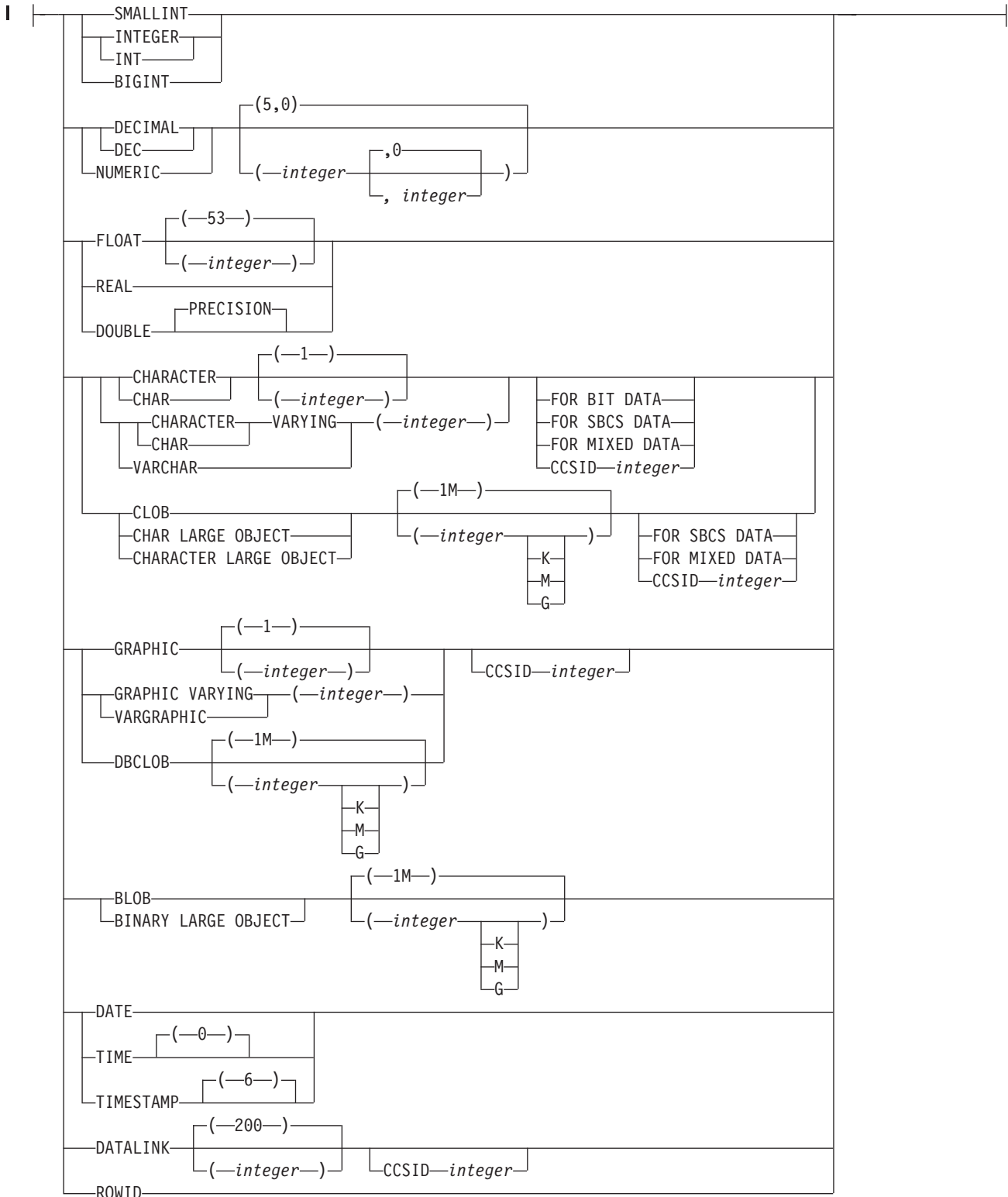
| *parameter-name data-type1* |

|
|

| data-type1, data-type2:

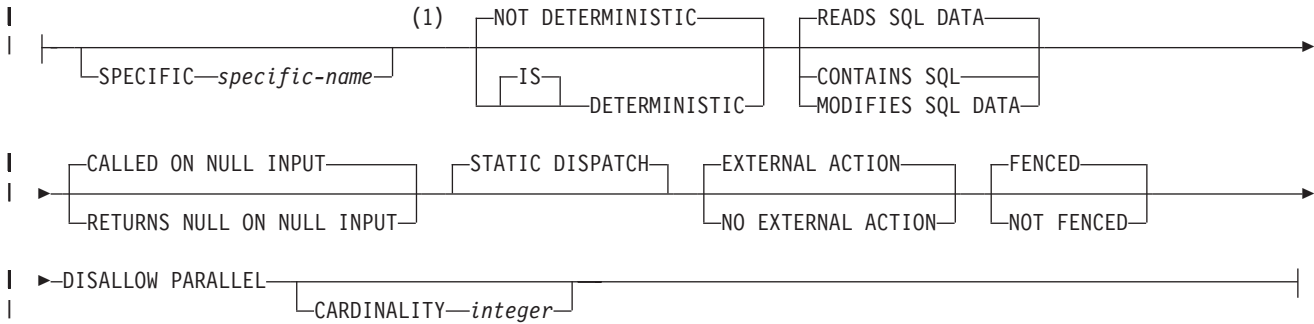


| built-in-type:



CREATE FUNCTION (SQL Table)

option-list:



Notes:

- 1 The optional clauses can be specified in a different order.

SQL-routine-body:

`SQL-control-statement`

Description

function-name

Names the user-defined function. The combination of name, schema name, the number of parameters, and the data type of each parameter (without regard for any length, precision, scale, or CCSID attributes of the data type) must not identify a user-defined function that exists at the current server.

For SQL naming, the function will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the function will be created in the schema that is specified by the qualifier. If no qualifier is specified, the function will be created in the current library (*CURLIB). If there is no current library, the function will be created in QGPL.

In general, more than one function can have the same name if the function signature of each function is unique.

Certain function names are reserved for system use. For more information see "Choosing the Schema and Function Name" on page 435.

(parameter-declaration,...)

Specifies the number of parameters of the function and the data type of each parameter. Although not required, you can give each parameter a name.

The maximum number of parameters allowed is 90.

parameter-name

Specifies the name of the input parameter. Do not specify the same name more than once. A parameter name must be specified for parameters in SQL functions.

data-type1

Specifies the number of input parameters of the function and the data type

CREATE FUNCTION (SQL Table)

of each input parameter. All the parameters for a function are input parameters. There must be one entry in the list for each parameter that the function expects to receive.

A function can have no parameters. In this case, you must code an empty set of parentheses, for example:

```
CREATE FUNCTION WOOFER()
```

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the function. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the function is invoked.

RETURNS TABLE

Specifies the output table of the function.

Assume the number of parameters is N. There must be no more than $(247-(N*2))/2$ columns.

column-name

Specifies the name of a column of the output table. Do not specify the same name more than once.

data-type2

Specifies the data type and attributes of the output.

You can specify any built-in data type (except LONG VARCHAR, or LONG VARGRAPHIC) or a distinct type.

If a CCSID is specified and the CCSID of the return data is encoded in a different CCSID, the data is converted to the specified CCSID.

If a CCSID is not specified the return data is converted to the CCSID of the job (or associated graphic CCSID of the job for graphic string return values), if the CCSID of the return data is encoded in a different CCSID. To avoid any potential loss of characters during the conversion, consider explicitly specifying a CCSID that can represent any characters that will be returned from the function. This is especially important if the data type is graphic string data. In this case, consider using CCSID 13488 (UCS-2 graphic string data).

SPECIFIC *specific-name*

Provides a unique name for the function. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another function or procedure that exists at the current server. If unqualified, the implicit qualifier is the same as the qualifier of the function name. If qualified, the qualifier must be the same as the qualifier of the function name.

If specific name is not specified, it is set to the function name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

LANGUAGE SQL

Specifies that this is an SQL function.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the function is deterministic.

NOT DETERMINISTIC

Specifies that the function will not always return the same result from successive function invocations with identical input arguments. NOT

CREATE FUNCTION (SQL Table)

DETERMINISTIC should be specified if the function contains a reference to a special register or a non-deterministic function.

DETERMINISTIC

Specifies that the function will always return the same result from successive invocations with identical input arguments.

CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA

Specifies whether the function can execute any SQL statements and, if so, what type. The database manager verifies that the SQL issued by the function is consistent with this specification. See Appendix C, "Characteristics of SQL Statements" on page 835 for a detailed list of the SQL statements that can be executed under each data access indication.

CONTAINS SQL

The function does not execute SQL statements that read or modify data.

READS SQL DATA

The function does not execute SQL statements that modify data.

MODIFIES SQL DATA

The function can execute any SQL statement except those statements that are not supported in any function.

FENCED or NOT FENCED

Specifies whether the function will run in the same thread as the invoking SQL statement or in a separate thread.

FENCED

The function will run in a separate thread. FENCED is the safest option if the function contains SQL cursors, because multiple invocations of the same function in the same SQL statement may conflict with each other.

NOT FENCED

The function may run in the same thread as the invoking SQL statement. NOT FENCED functions can keep SQL cursors open across individual calls to the function. Since cursors can be kept open, the cursor position will also be preserved between calls to the function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

Specifies whether the function is called if any of the input arguments is null at execution time.

RETURNS NULL ON INPUT

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

CALLED ON NULL INPUT

Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function contains an external action.

EXTERNAL ACTION

The function performs some external action (outside the scope of the function program). Thus, the function must be invoked with each successive function invocation. EXTERNAL ACTION should be specified if the function contains a reference to another function that has an external action.

CREATE FUNCTION (SQL Table)

NO EXTERNAL ACTION

The function does not perform an external action. It need not be called with each successive function invocation.

DISALLOW PARALLEL

Specifies that the function cannot be run in parallel. Table functions cannot run in parallel.

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 2 147 483 647 inclusive.

If the CARDINALITY clause is not specified for a table function, the database manager will assume a finite value as a default.

STATIC DISPATCH

Specifies that the function is dispatched statically. All functions are statically dispatched.

SET OPTION *statement*

Specifies the options that will be used to create the function. For example, to create a debuggable function, the following statement could be included:

```
SET OPTION DBGVIEW = *STMT
```

For more information, see “SET OPTION” on page 733.

The options CLOSQLCSR, CNULRQD, DFTRDBCOL, DYNDFTCOL, and NAMING are not allowed in the CREATE FUNCTION statement.

SQL-routine-body

Specifies a single SQL statement, including a compound statement. See Chapter 6, “SQL Control Statements” on page 779 for more information about defining SQL functions.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statement is not allowed in a function.

If the *SQL-routine-body* is a compound statement, it must contain exactly one RETURN statement and it must be executed when the function is called.

Notes

General Considerations for Defining User-defined Functions: See “CREATE FUNCTION” on page 435 for general information on defining user-defined functions.

Function ownership: If SQL names were specified, the *owner* of the function is the user profile with the same name as the schema into which the function is created. Otherwise, the *owner* of the function is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the function is the user profile or group user profile of the job executing the statement.

Function authority: If SQL names are used, functions are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, functions are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

CREATE FUNCTION (SQL Table)

If the owner of the function is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the function.

Creating the Function: When an SQL function is created, the database manager creates a temporary source file that will contain C source code with embedded SQL statements. A *SRVPGM object is then created using the CRTSRVPGM command. The SQL options used to create the service program are the options that are in effect at the time the CREATE FUNCTION statement is executed. The service program is created with ACTGRP(*CALLER).

The specific name is used to determine the name of the source file member and *SRVPGM object. If the specific name is a valid system name, it will be used as the name of member and program. If the member already exists, it will be overlaid. If a program already exists in the specified library, a unique name is generated using the rules for generating system table names. If the specific name is not a valid system name, a unique name is generated using the rules for generating system table names.

The function's attributes are saved in the associated service program object. If the *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

During restore of the function:

- If the specific name was specified when the function was originally created and it is not unique, an error is issued.
- If the specific name was not specified, a unique name is generated if necessary.
- If the signature is not unique, the function cannot be registered, and an error is issued.

Identifier Resolution: If the tables specified in a routine body exist, all references in the routine body of an SQL routine are resolved to identify a particular column, SQL parameter, or SQL variable at the time the SQL routine is created. If the tables do not exist, all names that exist as SQL variables or parameters are resolved to identify the variable or parameter when the function is created. The remaining names are assumed to be columns bound to the tables when the function is invoked.

If duplicate names are used for columns and SQL variables and parameters, qualify the duplicate names by using the table designator for columns, the function name for parameters, and the label name for SQL variables.

Invoking the Function: When an SQL function is invoked, it runs in the activation group of the calling program.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.

Example

Define a table function that returns the employees in a specified department number.

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
  RETURNS TABLE (EMPNO CHAR(6),
                 LASTNAME VARCHAR(15),
                 FIRSTNAME VARCHAR(12))
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  DISALLOW PARALLEL
  RETURN
  SELECT EMPNO, LASTNAME, FIRSTNAME
  FROM EMPLOYEE
  WHERE EMPLOYEE.WORKDEPT =DEPTEMPLOYEES.DEPTNO
```

CREATE INDEX

The CREATE INDEX statement creates an index on a table at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - *USE to the Create Logical File (CRTLF) command
 - *EXECUTE and *ADD to the library into which the index is created
 - *CHANGE to the data dictionary if the library into which the index is created is an SQL schema with a data dictionary
- Administrative authority

The privileges held by the authorization ID of the statement must also include at least one of the following:

- The INDEX privilege on the table
- Administrative authority

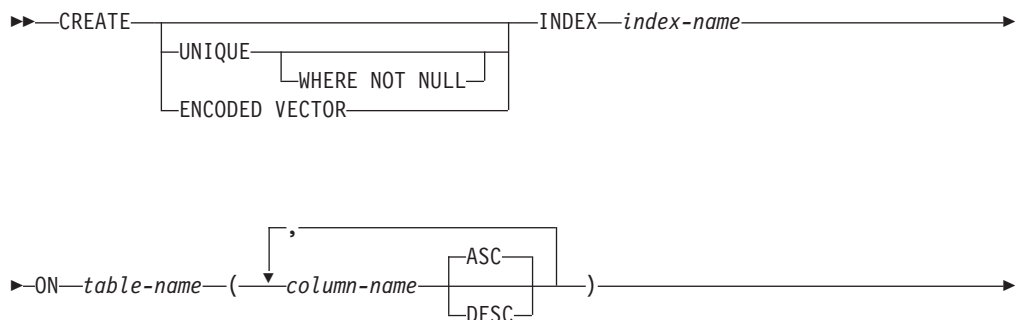
The authorization ID of the statement has the INDEX privilege on the table when:

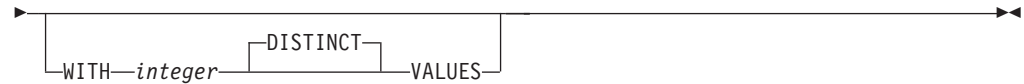
- It is the owner of the table,
- It has been granted the INDEX or ALTER privilege to the table, or
- It has been granted the system authorities of either *OBJALTER or *OBJMGT to the table

If SQL names are specified and a user profile exists that has the same name as the library into which the table is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

Syntax





Description

UNIQUE

Prevents the table from containing two or more rows with the same value of the index key. The constraint is enforced when rows of the table are updated or new rows are inserted.

The constraint is also checked during the execution of the CREATE INDEX statement. If the table already contains rows with duplicate key values, the index is not created.

When UNIQUE is used, null values are treated as any other values. For example, if the key is a single column that can contain null values, that column can contain no more than one null value.

UNIQUE WHERE NOT NULL

Prevents the table from containing two or more rows with the same nonnull value of the index key. Multiple null values are allowed; otherwise, this is identical to UNIQUE.

ENCODED VECTOR

Specifies that the resulting index will be an encoded vector index (EVI).

An encoded vector index cannot be used to ensure an ordering of rows. It is used by the database manager to improve the performance of queries. For more information, see the Database Performance and Query Optimization book.

index-name

Names the index. The name, including the implicit or explicit qualifier, must not be the same as an index, table, view, alias, or file that already exists at the current server.

If SQL names were specified, the index will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the index name will be created in the schema that is specified by the qualifier. If not qualified, the index name will be created in the same schema as the table over which the index is created.

If the index name is not a valid system name, DB2 UDB for iSeries will generate a system name. For information on the rules for generating a name, see "Rules for Table Name Generation" on page 553.

ON *table-name*

Identifies the table on which the index is to be created. The *table-name* must identify a base table (not a view) that exists at the current server.

(*column-name, ...*)

Identifies the list of columns that will be part of the index key.

Each *column-name* must be an unqualified name that identifies a column of the table. The same column may be specified more than once. A *column-name* must not identify a LOB or DATALINK column, or a distinct type based on a LOB

CREATE INDEX

or datalink column. The number of columns must not exceed 120, and the sum of their byte lengths must not exceed 2000–n, where n is the number of columns specified that allows nulls.

ASC

Puts the index entries in ascending order by the column. That is the default.

DESC

Puts the index entries in descending order by the column.

WITH *integer* DISTINCT VALUES

Specifies the estimated number of distinct key values. This clause may be specified for any type of index.

For encoded vector indexes this is used to determine the initial size of the codes assigned to each distinct key value. The default value is 256.

For non-encoded vector indexes, this is used as a hint to the optimizer.

Notes

Effects of the statement: CREATE INDEX creates a description of the index. If the named table already contains data, CREATE INDEX creates the index entries for it. If the table does not yet contain data, the index entries are created when data is inserted into the table.

Sort sequence: Any index created over columns containing SBCS or mixed data is created with the sort sequence in effect at the time the statement is executed. For sort sequences other than *HEX, the key for SBCS data or mixed data is the weighted value of the key based on the sort sequence.

Index attributes: An index is created as a keyed logical file. When an index is created, the file wait time and record wait time attributes are set to the default that is specified on the WAITFILE and WAITRCD keywords of the Create Logical File (CRTLF) command.

An index created over a distributed table is created on all of the servers across which the table is distributed. For more information about distributed tables, see the DB2 Multisystem book.

Index ownership: If SQL names were specified, the *owner* of the index is the user profile with the same name as the schema into which the index is created. Otherwise, the *owner* of the index is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the index is the user profile or group user profile of the job executing the statement.

Index authority: If SQL names are used, indexes are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, indexes are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the index is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the index.

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 PROCEDURE

The CREATE PROCEDURE statement defines a procedure at the current server.

The following types of procedures can be defined:

- External

The procedure program is written in a programming language such as C, COBOL or Java. The external executable is referenced by a procedure defined at the current server along with various attributes of the procedure. See “CREATE PROCEDURE (External)” on page 501.

- SQL

The procedure is written exclusively in SQL. The procedure body is defined at the current server along with various attributes of the procedure. See “CREATE PROCEDURE (SQL)” on page 512.

Notes

Choosing Data Types for Parameters: For portability of procedures across platforms that are not DB2 UDB for iSeries, do not use the following data types, which might have different representations on different platforms:

- FLOAT. Use DOUBLE or REAL instead.
- NUMERIC. Use DECIMAL instead.

Specifying AS LOCATOR for a Parameter: Passing a locator instead of a value can result in fewer bytes being passed in or out of the procedure. This can be useful when the value of the parameter is very large. The AS LOCATOR clause specifies that a locator to the value of the parameter is passed instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type.

AS LOCATOR can not be specified for SQL procedures.

Determining the Uniqueness of Procedures in a Schema: At the current server, each procedure signature must be unique. The signature of a procedure is the qualified procedure name combined with the number of the input parameters (the data types of the parameters are not part of a procedure’s signature). This means that two different schemas can each contain a procedure with the same name that have the same number of parameters. However, a schema must not contain two procedures with the same name that have the same number of parameters.

The Specific Name for a Procedure: When defining multiple procedures with the same name and schema (with different number of parameters), it is recommended that a specific name also be specified. The specific name can be used to uniquely identify the procedure when dropping, granting to, revoking from, or commenting on the procedure.

If the SPECIFIC clause is not specified, a specific name is generated.

CREATE PROCEDURE (External)

The CREATE PROCEDURE (External) statement defines an external procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPROCS catalog view and SYSPARMS catalog table:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

If the external program exists, the privileges held by the authorization ID of the statement must include at least one of the following:

- For the external program referenced in the SQL statement:
 - The system authority *EXECUTE on the external program, and
 - The system authority *EXECUTE on the library containing the external program
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

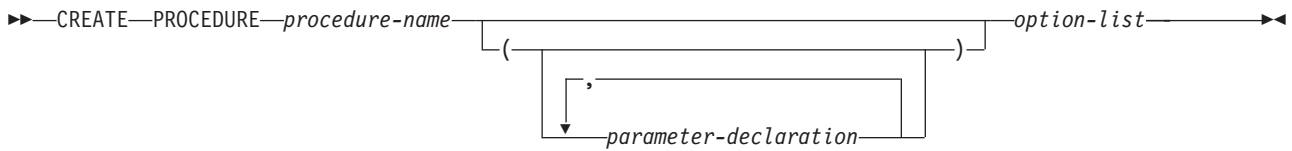
- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

Syntax

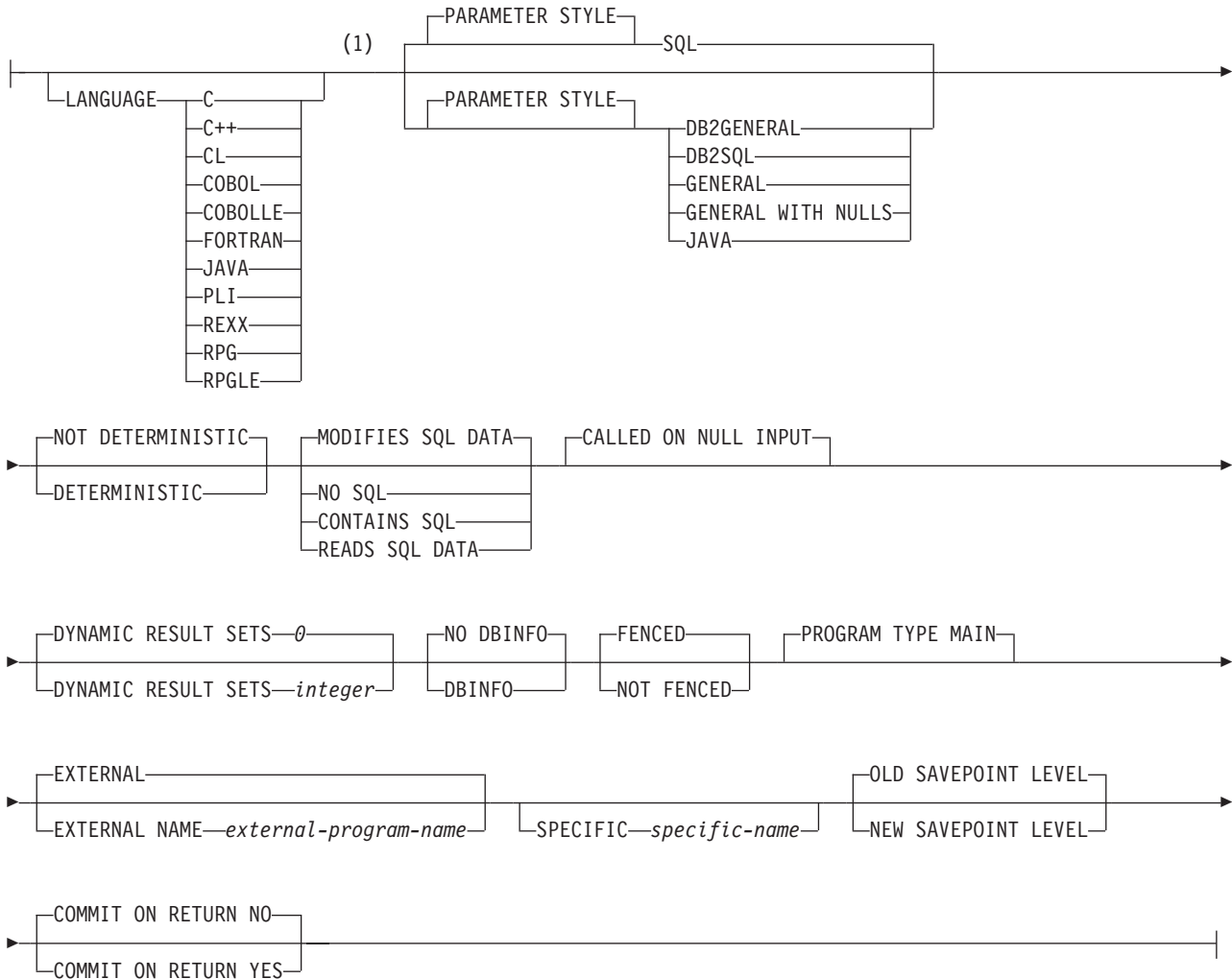
CREATE PROCEDURE (External)



parameter-declaration:



option-list:



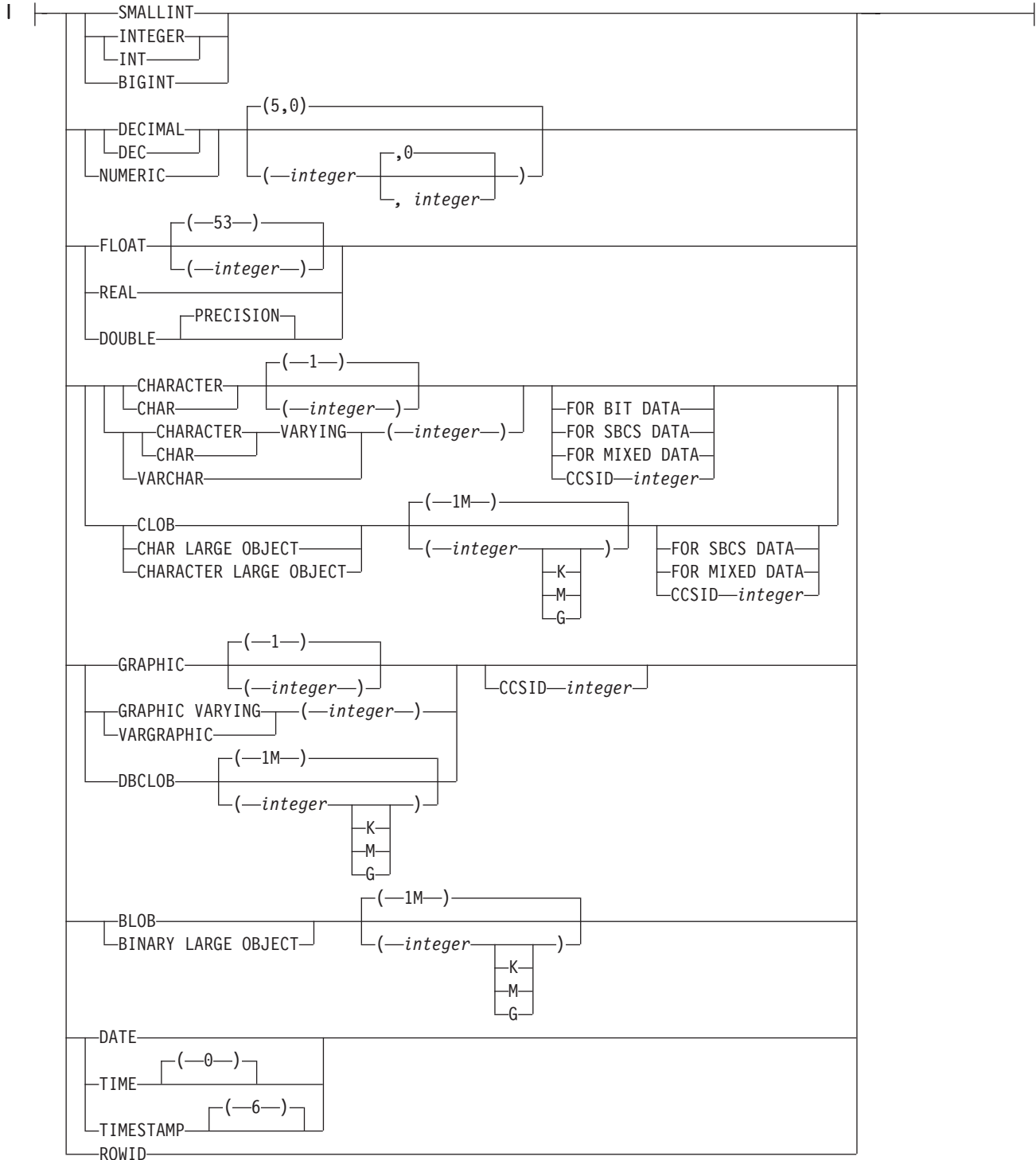
Notes:

- 1 The optional clauses can be specified in a different order.

data-type:



built-in-type:



CREATE PROCEDURE (External)

Description

procedure-name

Names the procedure. The combination of name, schema name, the number of parameters must not identify a procedure that exists at the current server.

For SQL naming, the procedure will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the procedure will be created in the schema specified by the qualifier. If no qualifier is specified, the procedure will be created in the current library (*CURLIB).

(parameter-declaration,...)

Specifies the number of parameters of the procedure and the data type of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in CREATE PROCEDURE is 255. If GENERAL WITH NULLS is specified, the maximum is 254. If parameter style SQL is specified, only 90 parameters are allowed. The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program used to compile the external program.

IN Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned.⁵¹

OUT

Identifies the parameter as an output parameter that is returned by the procedure.

A DataLink or a distinct type based on a DataLink may not be specified as an output parameter.

INOUT

Identifies the parameter as both an input and output parameter for the procedure.

A DataLink or a distinct type based on a DataLink may not be specified as an input and output parameter.

parameter-name

Names the parameter. The name cannot be the same as any other *parameter-name* for the procedure.

data-type

Specifies the data type of the parameter.

The data type must be valid for the language specified in the language clause. DataLinks are not valid for external procedures. For more information about data types, see "CREATE TABLE" on page 525, and the SQL Programming Concepts book.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the procedure. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the procedure is called.

51. When the language type is REXX, all parameters must be input parameters.

CREATE PROCEDURE (External)

AS LOCATOR

Specifies that the input parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the input parameter has a LOB data type or a distinct type based on a LOB data type.

LANGUAGE

Specifies the language that the external program is written in. The language clause is required if the external program is a REXX procedure.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program at the time the procedure is created. If the program attribute information associated with the program does not identify a recognizable language or the program cannot be found, then the language is assumed to be C.

C The external program is written in C.

C++

The external program is written in C++.

CL

The external program is written in CL.

COBOL

The external program is written in COBOL.

COBOLLE

The external program is written in ILE COBOL.

FORTRAN

The external program is written in FORTRAN.

JAVA

The external program is written in JAVA.

PLI

The external program is written in PL/I.

REXX

The external program is a REXX procedure.

RPG

The external program is written in RPG.

RPGLE

The external program is written in ILE RPG.

PARAMETER STYLE

Specifies the conventions used for passing parameters to and returning the values from procedures:

SQL

Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. The parameters are defined to be in the following order:

- The first N parameters are the parameters that are specified on the CREATE PROCEDURE statement.
- N parameters for indicator variables for the parameters.
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the procedure. The SQLSTATE returned is assigned by the external program.

CREATE PROCEDURE (External)

The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.

- A VARCHAR(517) input parameter for the fully qualified procedure name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(70) output parameter for the message text.

For more information about the parameters passed, see the include `sqludf` in the appropriate source file. For example, for C, `sqludf` can be found in `QSYSINC/H`.

PARAMETER STYLE SQL cannot be used with LANGUAGE JAVA.

DB2GENERAL

| Specifies that the procedure will use a parameter passing convention that
| is defined for use with Java methods.

| PARAMETER STYLE DB2GENERAL can only be specified with
| LANGUAGE JAVA. For details on passing parameters in JAVA, see the
| Developer Kit for Java book.

DB2SQL

Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. DB2SQL is identical to the SQL parameter style, except that the following additional parameter may be passed as the last parameter:

- A parameter for the `dbinfo` structure, if `DBINFO` was specified on the CREATE PROCEDURE statement.

For more information about the parameters passed, see the include `sqludf` in the appropriate source file. For example, for C, `sqludf` can be found in `QSYSINC/H`.

PARAMETER STYLE DB2SQL cannot be used with LANGUAGE JAVA.

GENERAL

Specifies that the procedure will use a parameter passing mechanism where the procedure receives the parameters specified on the CALL. Additional arguments are not passed for indicator variables.

PARAMETER STYLE GENERAL cannot be used with LANGUAGE JAVA.

GENERAL WITH NULLS

Specifies that in addition to the parameters on the CALL statement as specified in GENERAL, another argument is passed to the procedure. This additional argument contains an indicator array with an element for each of the parameters of the CALL statement. In C, this would be an array of short ints. For more information about how the indicators are handled, see the SQL Programming Concepts book.

PARAMETER STYLE GENERAL WITH NULLS cannot be used with LANGUAGE JAVA.

JAVA

| Specifies that the procedure will use a parameter passing convention that
| conforms to the Java language and SQLJ Routines specification. INOUT
| and OUT parameters will be passed as single entry arrays to facilitate
| returning values. For increased portability, you should write Java
| procedures that use the PARAMETER STYLE JAVA conventions.

CREATE PROCEDURE (External)

PARAMETER STYLE JAVA can only be specified with LANGUAGE JAVA. For details on passing parameters in JAVA, see the Developer Kit for Java book.

Note that language of the external procedure determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see the SQL Programming Concepts book.

EXTERNAL NAME *external-program-name*

Specifies the program that will be executed when the procedure is called by the CALL statement. The program name must identify a program that exists at the server at the time the procedure is called. If the naming option is *SYS and the program name is not qualified, the library list will be used to search for the program at the time the procedure is called. The program cannot be an ILE service program.

The validity of the name is checked at the server. If the format of the name is not correct, an error is returned.

If external-program-name is not specified, the external program name is assumed to be the same as the procedure name.

The external program need not exist at the time the procedure is created, but it must exist at the time the procedure is called.

CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statements are not allowed in a procedure that is running on a remote server. COMMIT and ROLLBACK statements are not allowed in an ATOMIC SQL procedure.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of result sets that can be returned from the procedure. *integer* must be greater than or equal to zero. If zero is specified, no result sets are returned. A procedure can have any number of result sets, but at any time, only 100 procedures can have result sets that are waiting to be fetched. If the SET RESULT SETS statement is issued, the number of results returned is the minimum of the number of result sets specified on this keyword and the SET RESULTS SET statement.

The result sets are scrollable. If a cursor is used to return a result set, the result set starts with the current position. Thus, if 5 FETCH NEXT operations have been performed prior to returning from the procedure, the result set will start with the 6th row of the result set.

Result sets are only returned if both the following are true:

- the procedure is called from a iSeries Access client ODBC or JDBC driver, JDBC on the iSeries server, or the SQL Call Level Interface, and
- the external program does not have an attribute of ACTGRP(*NEW).

For more information about result sets see "SET RESULT SETS" on page 750.

SPECIFIC *specific-name*

Provides a unique name for the procedure. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another procedure or function that exists at the current server. If unqualified, the implicit qualifier is the same as the qualifier of the procedure name. If qualified, the qualifier must be the same as the qualifier of the procedure name.

CREATE PROCEDURE (External)

If *specific-name* is not specified, it is the same as the procedure name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

NOT DETERMINISTIC

The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, provided the referenced data in the database has not changed.

DETERMINISTIC

The procedure may not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL

Specifies which SQL statements, if any, may be executed in the procedure or any routine called from this procedure. See Appendix C, "Characteristics of SQL Statements" on page 835 for a detailed list of the SQL statements that can be executed under each data access indication.

CONTAINS SQL

Specifies that SQL statements that neither read nor modify SQL data can be executed by the procedure.

NO SQL

Specifies that the procedure cannot execute any SQL statements.

READS SQL DATA

Specifies that SQL statements that do not modify SQL data can be included in the procedure.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

CALLED ON NULL INPUT

Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

FENCED or NOT FENCED

This parameter is allowed for compatibility with other products and is not used by DB2 UDB for iSeries.

PROGRAM TYPE MAIN

Specifies that the procedure executes as a main routine.

DBINFO

Specifies that the database manager should pass a structure containing status information to the procedure. Table 47 contains a description of the DBINFO structure. Detailed information about the DBINFO structure can be found in include file SQLUDF in QSYSINC.H.

DBINFO is only allowed with PARAMETER STYLE DB2SQL.

Table 47. DBINFO fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.

Table 47. DBINFO fields (continued)

Field	Data Type	Description
Authorization ID	VARCHAR(128)	The run-time authorization ID.
CCSID Information	INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. The following information identifies the CCSID: <ul style="list-style-type: none"> • SBCS CCSID • DBCS CCSID • Mixed CCSID • Indication of which of the first three CCSIDs is appropriate. • Reserved <p>If a CCSID is not explicitly specified for a parameter on the CREATE PROCEDURE statement, the input string is assumed to be encoded in the CCSID of the job at the time the function is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external function will be converted before calling the external program.</p>
Target Column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	Not applicable for a call to a procedure.
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.

OLD SAVEPOINT LEVEL or NEW SAVEPOINT LEVEL

Specifies whether a new savepoint level is to be created on entry to the procedure.

OLD SAVEPOINT LEVEL

A new savepoint level is not created. Any SAVEPOINT statements issued within the procedure with OLD SAVEPOINT LEVEL implicitly or explicitly specified on the SAVEPOINT statement are created at the same savepoint level as the caller of the procedure. This is the default.

NEW SAVEPOINT LEVEL

A new savepoint level is created on entry to the procedure. Any savepoints set within the procedure are created at a savepoint level that is nested deeper than the level at which this procedure was invoked. Therefore, the name of any new savepoint set within the procedure will not conflict with any existing savepoints set in higher savepoint levels (such as the savepoint level of the calling program) with the same name.

COMMIT ON RETURN

Specifies whether the database manager commits the transaction immediately on return from the procedure.

NO

The database manager does not issue a commit when the procedure returns. NO is the default.

YES

The database manager issues a commit if the procedure returns successfully. If the procedure returns with an error, a commit is not issued.

CREATE PROCEDURE (External)

The commit operation includes the work that is performed by the calling application process and the procedure.⁵²

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

Notes

General Considerations for Defining Procedures: See “CREATE PROCEDURE” on page 500 for general information on defining procedures.

Creating the Procedure: When an external procedure associated with an ILE external program is created, an attempt is made to save the procedure’s attributes in the associated program object. If the *PGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

The attributes can be saved for external procedures subject to the following restrictions:

- The external program library must not be SYSIBM, QSYS, or QSYS2.
- The external program must exist when the CREATE PROCEDURE statement is issued.
- The external program must be an ILE *PGM object.
- The external program must contain at least one SQL statement.

If the object cannot be updated, the procedure will still be created.

During restore of the procedure:

- If the specific name was specified when the procedure was originally created and it is not unique, an error is issued.
- If the specific name was not specified, a unique name is generated if necessary.
- If the procedure name and number of parameters is not unique, the procedure cannot be registered, and an error is issued.

Invoking the Procedure: If a DECLARE PROCEDURE statement defines a procedure with the same name as a created procedure, and a static CALL statement where the procedure name is not identified by a host variable is executed from the same source program, the attributes from the DECLARE PROCEDURE statement will be used rather than the attributes from the CREATE PROCEDURE statement.

The CREATE PROCEDURE statement applies to static and dynamic CALL statements as well as to a CALL statement where the procedure name is identified by a host variable.

When an external procedure is invoked, it runs in whatever activation group was specified when the external program was created. However, ACTGRP(*CALLER) should normally be used so that the procedure runs in the same activation group as the calling program.

52. If the external program was created with ACTGRP(*NEW) and the job commitment definition is not used, the work that is performed in the procedure will be committed or rolled back as a result of the activation group ending.

CREATE PROCEDURE (External)

Notes for Java Procedures: To be able to run Java procedures, you must have the Developer Kit for Java (5722-JV1) installed on your system. Otherwise, an SQLCODE of -443 will be returned and a CPDB521 message will be placed in the job log.

If an error occurs while running a Java procedure, an SQLCODE of -443 will be returned. Depending on the error, other messages may exist in the job log of the job where the procedure was run.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords SIMPLE CALL can be used as a synonym for GENERAL.
- The value DB2GENRL may be used as a synonym for DB2GENERAL.
- DYNAMIC RESULT SET, RESULT SETS, and RESULT SET may be used as synonyms for DYNAMIC RESULT SETS.

Example

Example 1: Create the procedure definition for a 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 PARTS_ON_HAND (IN PARTNUM INTEGER,
                                OUT COST    DECIMAL(7,2),
                                OUT QUANTITY INTEGER)
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'parts.onhand'
```

Example 2: Create the procedure definition for a 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)
LANGUAGE C
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 1
FENCED
EXTERNAL NAME ASSEMBLY
```

CREATE PROCEDURE (SQL)

CREATE PROCEDURE (SQL)

The CREATE PROCEDURE (SQL) statement creates an SQL procedure at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the SYSPROCS catalog view and SYSPARMS catalog table:
 - The INSERT privilege on the table, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - *USE on the Create Program (CRTPGM) command, and
 - *EXECUTE and *ADD on the library into which the procedure is created.
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the procedure is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

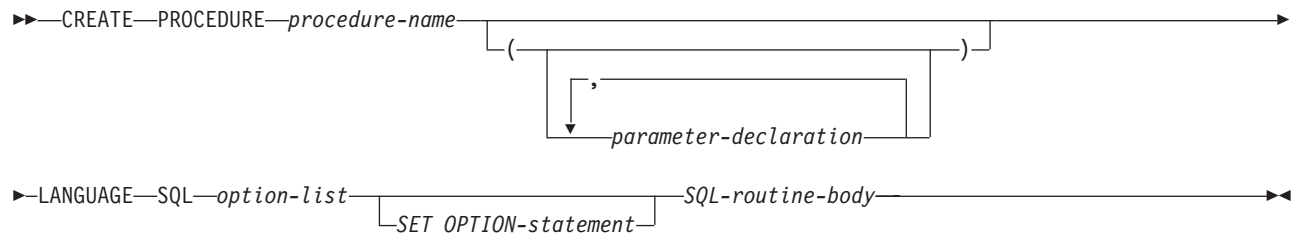
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

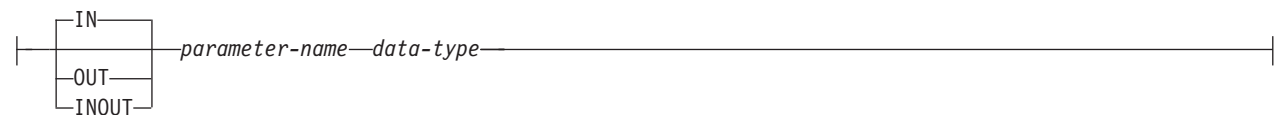
The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

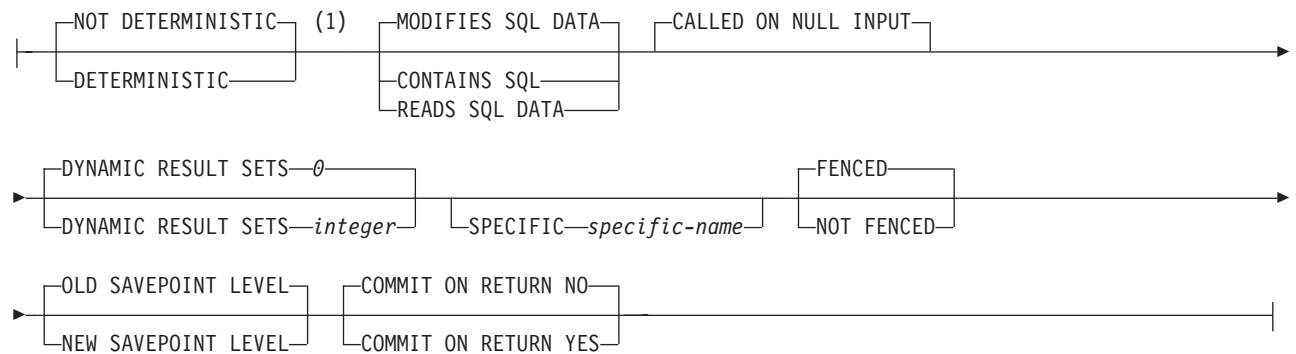
Syntax



parameter-declaration:



option-list:



Notes:

- 1 The optional clauses can be specified in a different order.

CREATE PROCEDURE (SQL)

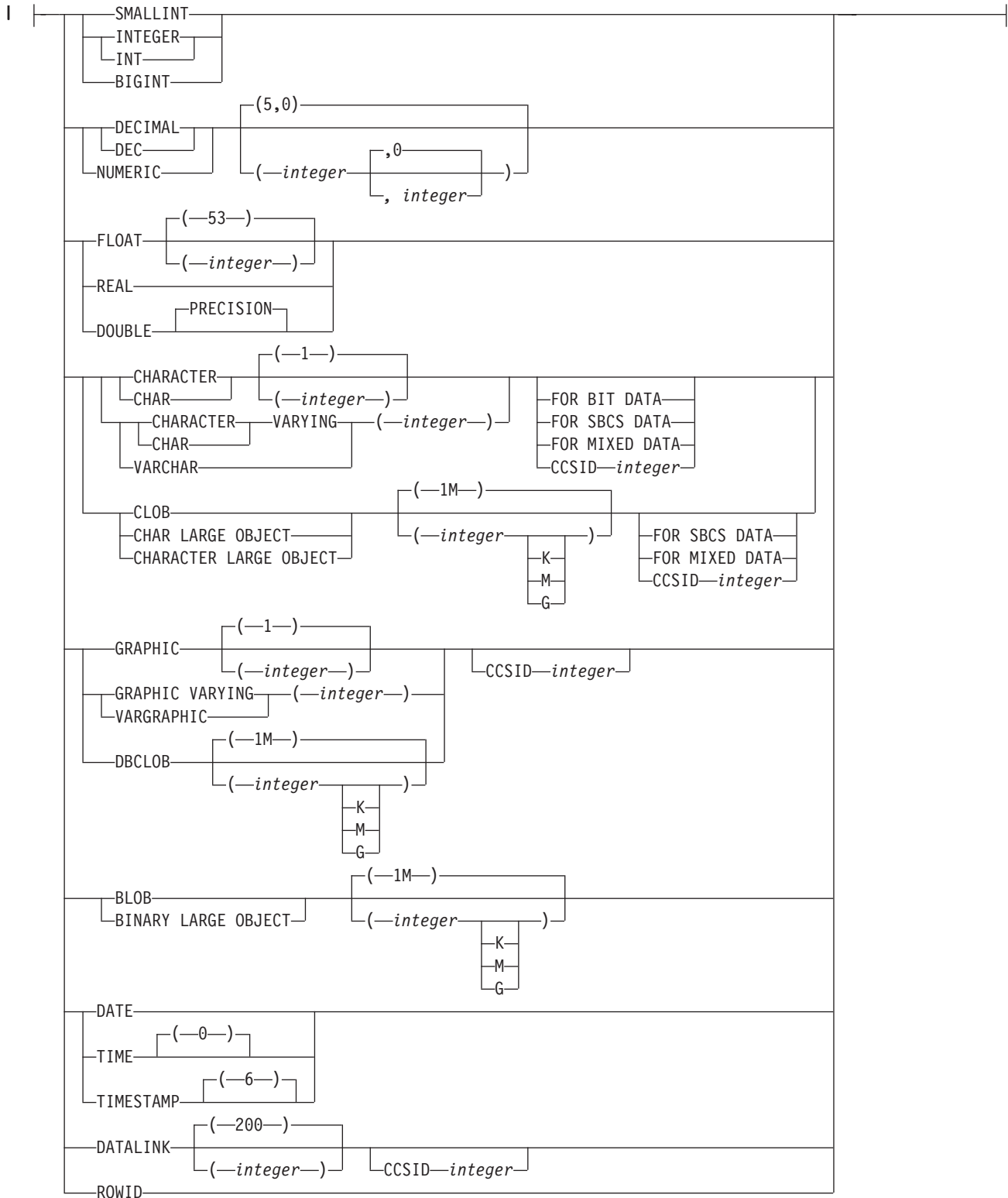
SQL-routine-body:

SQL-control-statement
ALTER-statement
COMMENT-statement
COMMIT-statement
CONNECT-statement
CREATE ALIAS-statement
CREATE DISTINCT TYPE-statement
CREATE FUNCTION (External Scalar)-statement
CREATE FUNCTION (External Table)-statement
CREATE FUNCTION (Sourced)-statement
CREATE INDEX-statement
CREATE PROCEDURE (External)-statement
CREATE SCHEMA-statement
CREATE TABLE-statement
CREATE VIEW-statement
DECLARE GLOBAL TEMPORARY TABLE-statement
DELETE-statement
DISCONNECT-statement
DROP-statement
EXECUTE IMMEDIATE-statement
GRANT-statement
INSERT-statement
LABEL-statement
LOCK TABLE-statement
RELEASE-statement
RELEASE SAVEPOINT-statement
RENAME-statement
REVOKE-statement
ROLLBACK-statement
SAVEPOINT-statement
SELECT INTO-statement
SET CONNECTION-statement
SET PATH-statement
SET SCHEMA-statement
SET RESULT SETS-statement
SET TRANSACTION-statement
UPDATE-statement
VALUES INTO-statement

data-type:



built-in-type:



CREATE PROCEDURE (SQL)

Description

procedure-name

Names the procedure. The combination of name, schema name, the number of parameters must not identify a procedure that exists at the current server.

For SQL naming, the procedure will be created in the schema specified by the implicit or explicit qualifier.

For system naming, the procedure will be created in the schema specified by the qualifier. If no qualifier is specified, the procedure will be created in the current library (*CURLIB). If there is no current library, the procedure will be created in QGPL.

(parameter-declaration,...)

Specifies the number of parameters of the procedure and the data type of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in an SQL procedure is 253.

IN Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned.

OUT

Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT

Identifies the parameter as both an input and output parameter for the procedure.

parameter-name

Names the parameter. The name cannot be the same as any other *parameter-name* for the procedure.

data-type

Specifies the data type of the parameter.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the procedure. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the procedure is called.

LANGUAGE SQL

Specifies that this is an SQL procedure.

DYNAMIC RESULT SETS *integer*

| Specifies the maximum number of result sets that can be returned from the
| procedure. *integer* must be greater than or equal to zero. If zero is specified, no
| result sets are returned. A procedure can have any number of result sets, but at
| any time, only 100 procedures can have result sets that are waiting to be
| fetched. If the SET RESULT SETS statement is issued, the number of results
| returned is the minimum of the number of result sets specified on this
| keyword and the SET RESULTS SET statement.

| The result sets are scrollable. If a cursor is used to return a result set, the result
| set starts with the current position. Thus, if 5 FETCH NEXT operations have
| been performed prior to returning from the procedure, the result set will start
| with the 6th row of the result set.

Result sets are only returned if:

- the procedure is called from a iSeries Access client ODBC or JDBC driver, JDBC on the iSeries server, or the SQL Call Level Interface, and

For more information about result sets, see “SET RESULT SETS” on page 750.

SPECIFIC *specific-name*

Provides a unique name for the procedure. The name is implicitly or explicitly qualified with a schema name. The name, including the schema name, must not identify the specific name of another procedure or function that exists at the current server. If unqualified, the implicit qualifier is the same as the qualifier of the procedure name. If qualified, the qualifier must be the same as the qualifier of the procedure name.

If *specific-name* is not specified, it is the same as the procedure name. If a function or procedure with that specific name already exists, a unique name is generated similar to the rules used to generate unique table names.

DETERMINISTIC or **NOT DETERMINISTIC**

Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

NOT DETERMINISTIC

The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, provided the referenced data in the database has not changed.

DETERMINISTIC

The procedure may not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA

Specifies which SQL statements may be executed in the procedure or any routine called from this procedure. See Appendix C, “Characteristics of SQL Statements” on page 835 for a detailed list of the SQL statements that can be executed under each data access indication.

CONTAINS SQL

Specifies that SQL statements that neither read nor modify SQL data can be executed by the procedure.

READS SQL DATA

Specifies that SQL statements that do not modify SQL data can be included in the procedure.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

CALLED ON NULL INPUT

Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

FENCED or **NOT FENCED**

This parameter is allowed for compatibility with other products and is not used by DB2 UDB for iSeries.

SET OPTION-*statement*

Specifies the options that will be used to create the procedure. For example, to create a debuggable procedure, the following statement could be included:

CREATE PROCEDURE (SQL)

SET OPTION DBGVIEW = *STMT

For more information, see “SET OPTION” on page 733.

The options CLOSQLCSR, CNULRQD, DFTRDBCOL, DYNDFTCOL, and NAMING are not allowed in the CREATE PROCEDURE statement.

OLD SAVEPOINT LEVEL or NEW SAVEPOINT LEVEL

Specifies whether a new savepoint level is to be created on entry to the procedure.

OLD SAVEPOINT LEVEL

A new savepoint level is not created. Any SAVEPOINT statements issued within the procedure with OLD SAVEPOINT LEVEL implicitly or explicitly specified on the SAVEPOINT statement are created at the same savepoint level as the caller of the procedure. This is the default.

NEW SAVEPOINT LEVEL

A new savepoint level is created on entry to the procedure. Any savepoints set within the procedure are created at a savepoint level that is nested deeper than the level at which this procedure was invoked. Therefore, the name of any new savepoint set within the procedure will not conflict with any existing savepoints set in higher savepoint levels (such as the savepoint level of the calling program) with the same name.

COMMIT ON RETURN

Specifies whether the database manager commits the transaction immediately on return from the procedure.

NO

The database manager does not issue a commit when the procedure returns. NO is the default.

YES

The database manager issues a commit if the procedure returns successfully. If the procedure returns with an error, a commit is not issued.

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

SQL-routine-body

Specifies a single SQL statement, including a compound statement. See Chapter 6, “SQL Control Statements” on page 779 for more information about defining SQL procedures.

CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK and SET TRANSACTION statements are not allowed in a procedure that is running on a remote server. COMMIT and ROLLBACK statements are not allowed in an ATOMIC SQL procedure.

Notes

General Considerations for Defining Procedures: See “CREATE PROCEDURE” on page 500 for general information on defining procedures.

CREATE PROCEDURE (SQL)

Procedure ownership: If SQL names were specified, the *owner* of the procedure is the user profile with the same name as the schema into which the procedure is created. Otherwise, the *owner* of the procedure is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the procedure is the user profile or group user profile of the job executing the statement.

Procedure authority: If SQL names are used, procedures are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, procedures are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the procedure is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the procedure.

Error Handling in Procedures: Consideration should be given to possible exceptions that can occur for each SQL statement in the body of a procedure. Any exception SQLSTATE that is not handled within the procedure using a handler within a compound compound statement, results in the exception SQLSTATE being returned to the caller of the procedure.

Creating the Procedure: When an SQL procedure is created, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program object is then created using the CRTPGM command. The SQL options used to create the program are the options that are in effect at the time the CREATE PROCEDURE statement is executed. The program is created with ACTGRP(*CALLER).

When an SQL procedure is created, the procedure's attributes are stored in the created program object. If the *PGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes.

During restore of the procedure:

- If the specific name was specified when the procedure was originally created and it is not unique, an error is issued.
- If the specific name was not specified, a unique name is generated if necessary.
- If the procedure name and number of parameters is not unique, the procedure cannot be registered, and an error is issued.

The procedure name is used as the name of the member in the source file and the name of the program object, if it is a valid system name. If the procedure name is not a valid system name, a unique name is generated. If a source file member with the same name already exists, the member is overlaid. If a module or a program with the same name already exists, the objects are not overlaid, and a unique name is generated. The unique names are generated according to the rules for generating system table names.

Invoking the Procedure: If a DECLARE PROCEDURE statement defines a procedure with the same name as a created procedure, and a static CALL statement where the procedure name is not identified by a host variable is executed from the same source program, the attributes from the DECLARE PROCEDURE statement will be used rather than the attributes from the CREATE PROCEDURE statement.

CREATE PROCEDURE (SQL)

The CREATE PROCEDURE statement applies to static and dynamic CALL statements as well as to a CALL statement where the procedure name is identified by a host variable.

SQL procedures must be called using the SQL CALL statement. When called, the SQL procedure runs in the activation group of the calling program.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- DYNAMIC RESULT SET, RESULT SETS, and RESULT SET may be used as synonyms for DYNAMIC RESULT SETS.

Example

Create an SQL procedure that returns the median staff salary. Return a result set containing the name, position, and salary of all employees who earn more than the median salary.

```
CREATE PROCEDURE MEDIAN_RESULT_SET (OUT medianSalary DECIMAL(7,2))
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE c1 CURSOR FOR
    SELECT salary
      FROM staff
     ORDER BY salary;
  DECLARE c2 CURSOR WITH RETURN FOR
    SELECT name, job, salary
      FROM staff
     WHERE salary > medianSalary
     ORDER BY salary;
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET medianSalary = 6666;
  SET medianSalary = 0;
  SELECT COUNT(*) INTO v_numRecords FROM STAFF;
  OPEN c1;
  WHILE v_counter < (v_numRecords / 2 + 1)
    DO FETCH c1 INTO medianSalary;
    SET v_counter = v_counter + 1;
  END WHILE;
  CLOSE c1;
  OPEN c2;
END
```

CREATE SCHEMA

The CREATE SCHEMA statement defines a schema at the current server and optionally creates tables, views, aliases, indexes, and distinct types. Comments and labels may be added in the catalog description of tables, views, aliases, indexes, columns, and distinct types. Table, view, and distinct type privileges can be granted to users.

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 of the statement must include at least one of the following:

- The *USE system authority to the following CL commands:
 - Create Library (CRTLIB)
 - If WITH DATA DICTIONARY is specified, Create Data Dictionary (CRTDTADCT)
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- The privileges defined for each SQL statement included in the CREATE SCHEMA statement
- Administrative authority

If the AUTHORIZATION clause is specified, the privileges held by the authorization ID of the statement must also include at least one of the following:

- The system authority *ADD to the user profile identified by authorization-name
- Administrative authority

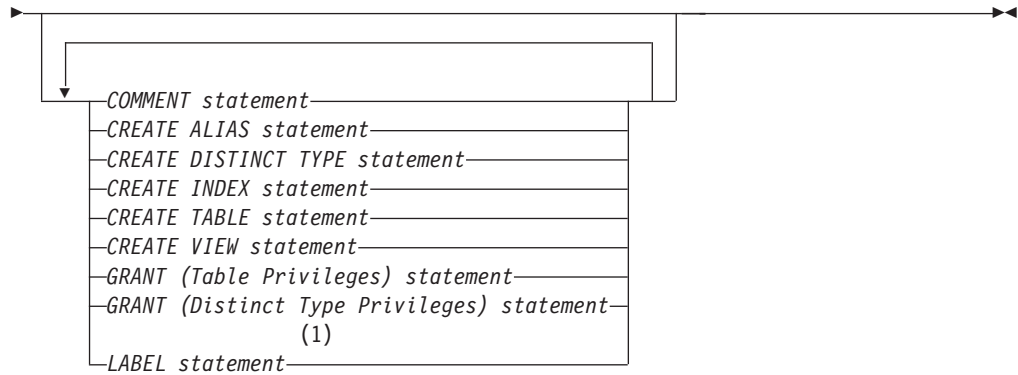
Syntax

```

▶▶ CREATE SCHEMA schema-name
                [AUTHORIZATION authorization-name]
▶
[IN ASP integer
  [ASP-name]] [WITH DATA DICTIONARY]

```

CREATE SCHEMA



Notes:

- 1 Labels and comments on packages, procedures, functions, and parameters are not supported in the CREATE SCHEMA statement.

Description

schema-name

Names the schema. A schema is created using this name. If *schema-name* is specified, the authorization ID of the statement is the run-time authorization ID. The name must not be the same as the name of an existing schema at the current server. The *owner* of the schema is the user profile or group user profile of the job executing the statement.

If the owner of the schema is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the schema.

authorization-name

Identifies the authorization ID of the statement. This authorization name is also the *schema-name*. The name must not be the same as the name of an existing schema at the current server.

IN ASP *integer*

Specifies the auxiliary storage pool (ASP) in which to create the schema. The integer must be between 1 and 32. If 1 is specified, the schema is created on the system ASP. If this clause is omitted, an ASP of 1 is assumed.

IN ASP *ASP-name*

Specifies the auxiliary storage pool (ASP) in which to create the schema. The name must identify an auxiliary storage pool that exists at the current server.

WITH DATA DICTIONARY

If this clause is specified, an IDDU data dictionary is created in the schema.

A schema created with a data dictionary cannot contain tables with LOB or DATALINK columns.

COMMENT statement

Adds or replaces comments in the catalog descriptions of tables, views, or columns. Comments on packages are not allowed. See the COMMENT statement "COMMENT" on page 404.

CREATE ALIAS statement

Creates an alias into the schema. See the CREATE ALIAS statement on page 425.

CREATE DISTINCT TYPE statement

Creates a user-defined distinct type into the schema. See the CREATE DISTINCT TYPE statement “CREATE DISTINCT TYPE” on page 428.

CREATE INDEX statement

Creates an index into the schema. See the CREATE INDEX statement on page 496.

CREATE TABLE statement

Creates a table into the schema. See the CREATE TABLE statement on page 525.

CREATE VIEW statement

Creates a view into the schema. See the CREATE VIEW statement 569.

GRANT (Table Privileges) statement

Grants privileges for tables and views in the schema. See the GRANT statement “GRANT (Table or View Privileges)” on page 664.

GRANT (Distinct Type Privileges) statement

Grants privileges for distinct types in the schema. See the GRANT statement “GRANT (Distinct Type Privileges)” on page 652.

LABEL statement

Adds or replaces labels in the catalog descriptions of tables, views, or columns in the schema. Labels on packages are not allowed. See the LABEL statement on page 681.

Notes

Schema attributes: A schema is created as:

- A library: A library groups related objects, and allows you to find objects by name.
- A catalog: A catalog contains descriptions of the tables, views, indexes, and packages in the schema. A catalog consists of a set of views and if WITH DATA DICTIONARY is specified, an IDDU data dictionary. For more information, see the SQL Programming Concepts book.
- A journal and journal receiver: A journal QSQRN and journal receiver QSQRN0001 is created in the schema, and is used to record changes to all tables subsequently created in the schema. For more information, see the Journal Management topic in the iSeries Information Center.

An index created over a distributed table is created on all of the servers across which the table is distributed. For more information about distributed tables, see the DB2 Multisystem book.

Object ownership: The owner of the created objects is determined as follows:

- If an AUTHORIZATION clause is specified, the specified authorization ID owns all objects created by the statement.
- If an AUTHORIZATION clause is not specified and SQL names are specified, the *owner* of all objects created by the statement is the user profile with the same name as the schema-name (if a user profile with that name exists).
- Otherwise, the *owner* of all objects created by the statement is the user profile or the group user profile of the job executing the statement.

Object authority: If SQL names are used, the schema and any other objects are created with the system authority of *EXCLUDE on *PUBLIC and the library is created with the create authority parameter CRTAUT(*EXCLUDE). The owner is

CREATE SCHEMA

the only user having any authority to the schema. If other users require authority to the schema, the owner can grant authority to the objects created; using the CL command Grant Object Authority (GRTOBJAUT).

If system names are used, the schema and any other objects are created with the system authority given to *PUBLIC is determined by the system value QCRTAUT, and the library is created with CRTAUT(*SYSVAL). For more information about

system security, see the books iSeries Security Reference , and the SQL Programming Concepts book.

If the owner of the schema is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the schema.

Object names: If a CREATE TABLE, CREATE INDEX, CREATE ALIAS, CREATE DISTINCT TYPE, or CREATE VIEW statement contains a qualified name for the table, index, alias, distinct type, or view being created, the schema name specified in that qualified name must be the same as the name of the schema being created. Any other table or view names referenced within the schema definition may be qualified by any schema name. Unqualified table, index, alias, distinct type, or view names in any SQL statement are implicitly qualified with the name of the created schema.

Delimiters are not used between the SQL statements.

SQL statement length: The maximum length of any individual CREATE TABLE, CREATE INDEX, CREATE DISTINCT TYPE, CREATE VIEW, COMMENT, LABEL, or GRANT statements within the CREATE SCHEMA statement is 65536.

Syntax alternatives: The COLLECTION keyword can be used as a synonym for SCHEMA for compatibility to prior releases. This keyword is non-standard and should not be used.

Examples

Example 1: Create a schema that has an inventory part table and an index over the part number. Give authority to the schema to the user profile JONES.

```
CREATE SCHEMA INVENTORY

CREATE TABLE PART (PARTNO SMALLINT NOT NULL,
DESCR VARCHAR(24),
QUANTITY INT)

CREATE INDEX PARTIND ON PART (PARTNO)

GRANT ALL ON PART TO JONES
```

Example 2: Create a schema using the authorization ID of SMITH. Create a student table that has a comment on the student number column.

```
CREATE SCHEMA AUTHORIZATION SMITH

CREATE TABLE SMITH.STUDENT (STUDNBR SMALLINT NOT NULL UNIQUE,
LASTNAME CHAR(20),
FIRSTNAME CHAR(20),
ADDRESS CHAR(50))

COMMENT ON STUDENT (STUDNBR IS 'THIS IS A UNIQUE ID#')
```


CREATE TABLE

The CREATE TABLE statement defines a table at the current server. The definition must include its name and the names and attributes of its columns. The definition may include other attributes of the table such as primary key.

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 of the statement must include at least one of the following:

- The following system authorities:
 - *USE to the Create Physical File (CRTPF) command
 - *EXECUTE and *ADD to the library into which the table is created
 - *OBJOPR and *OBJMGT to the journal⁵³
 - *CHANGE to the data dictionary if the library into which the table is created is an SQL schema with a data dictionary
- Administrative authority

If SQL names are specified and a user profile exists that has the same name as the library into which the table is created, and that name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- The system authority *ADD to the user profile with that name
- Administrative authority

To define a foreign key, the privileges held by the authorization ID of the statement must include at least one of the following on the parent table:

- The REFERENCES privilege or object management authority for the table
- The REFERENCES privilege on each column of the specified parent key
- Ownership of the table
- Administrative authority

The authorization ID of the statement has the REFERENCES privilege on a table when one of the following is true:

- It is the owner of the table.
- It was granted the REFERENCES privilege to the table.
- It was granted the system authorities of either *OBJREF or *OBJMGT to the table.

The authorization ID of the statement has the REFERENCES privilege on a column of the table when one of the following is true:

- It is the owner of the table.
- It was granted the REFERENCES privilege to the column.

53. If the authorization ID of the statement does not have the appropriate authority to the journal, the table will be created, but it will not be journaled.

CREATE TABLE

- It was granted the system authority of *OBJREF to the column or the system authority of *OBJMGT to the table.

If the LIKE clause or AS-select-statement is specified, the privileges held by the authorization ID of the statement must include at least one of the following on the tables or views specified in these clauses:

- The SELECT privilege for the table or view
- Ownership of the table or view
- Administrative authority

The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table.

The authorization ID of the statement has the SELECT privilege on a view when:

- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views that this view is directly or indirectly dependent on. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

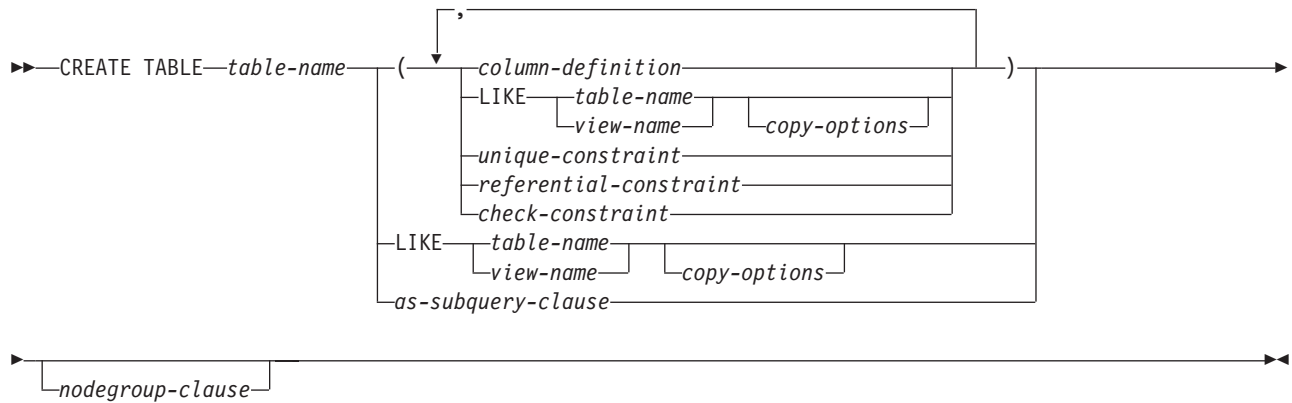
- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

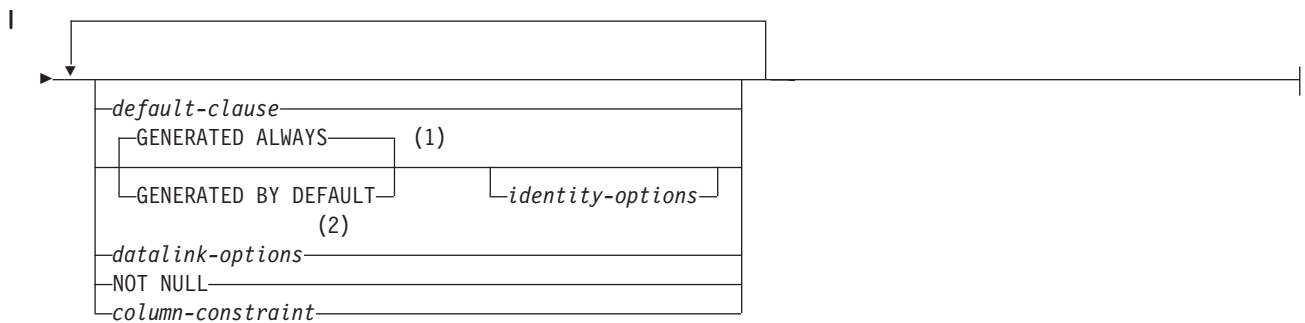
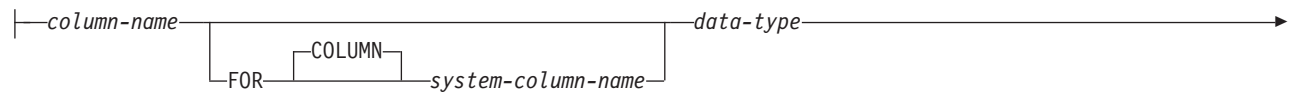
- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

Syntax

CREATE TABLE



column-definition:



Notes:

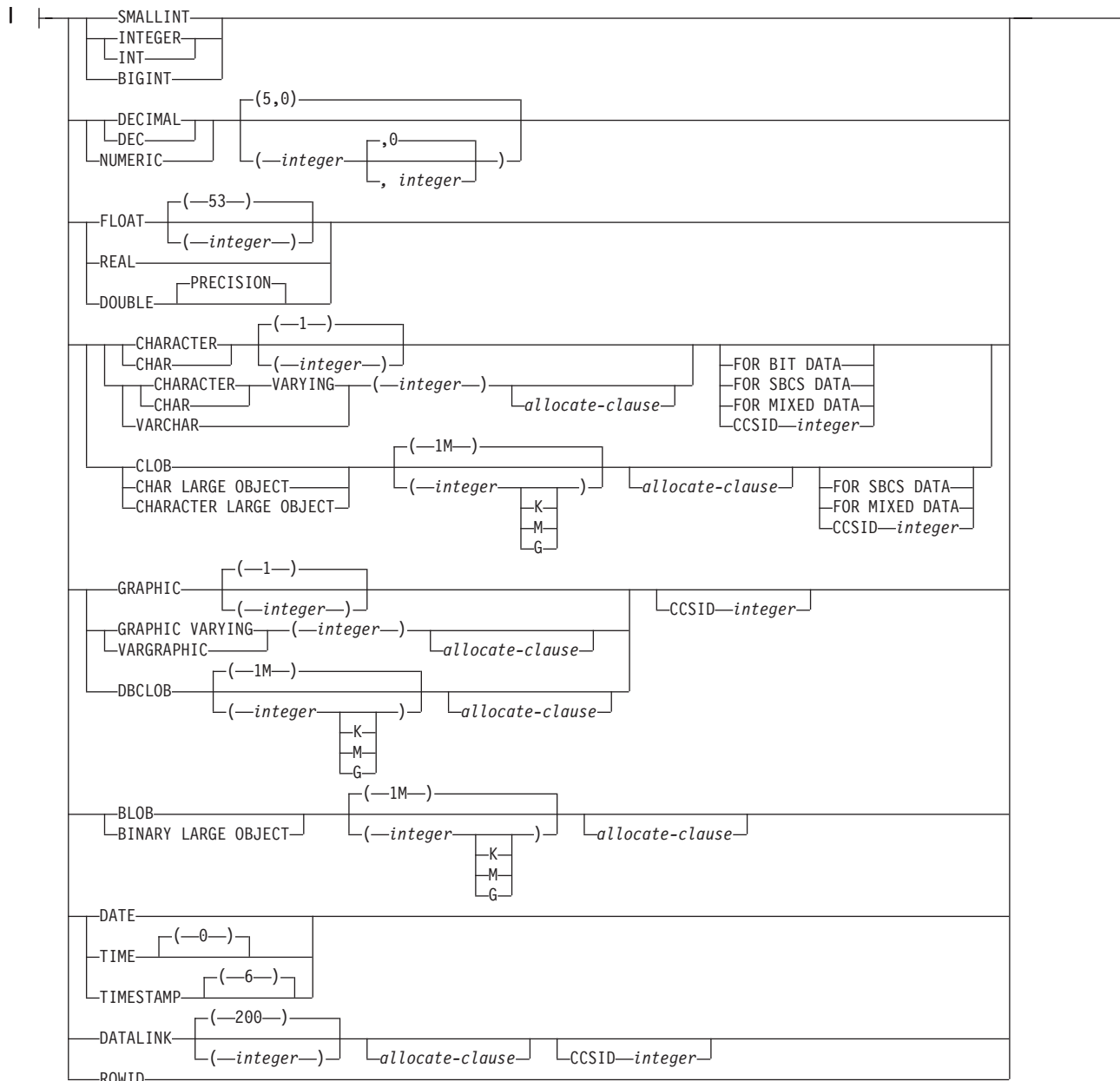
- 1 GENERATED can be specified only if the column has a ROWID data type (or a distinct type that is based on a ROWID data type), or the column is an identity column.
- 2 The datalink-options can only be specified for DATALINKs and distinct-types sourced on DATALINKs.

CREATE TABLE

data-type:

| *built-in-type* |
| *distinct-type-name* |

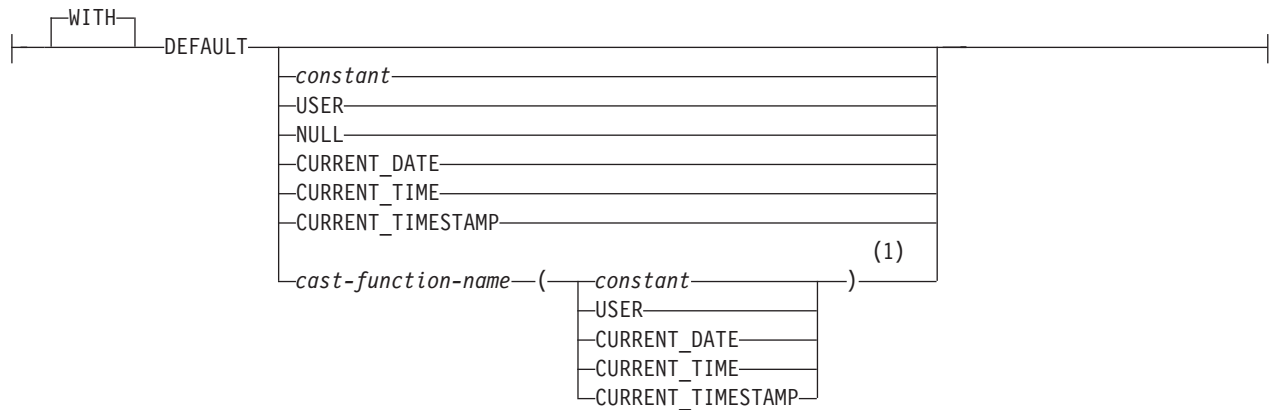
built-in-type:



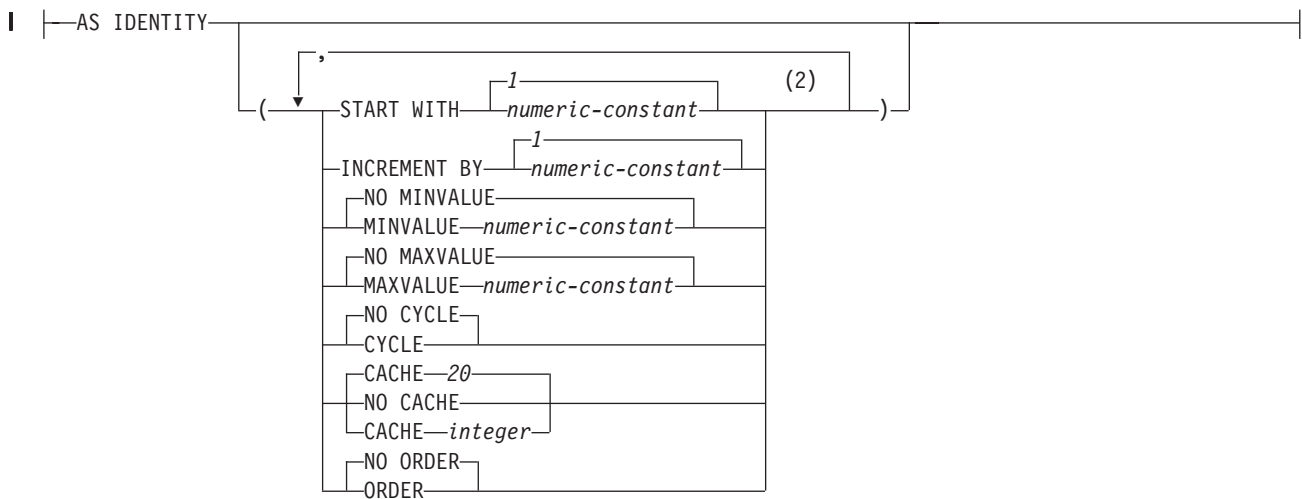
allocate-clause:

| ALLOCATE—(integer)— |

default-clause:



identity-options:

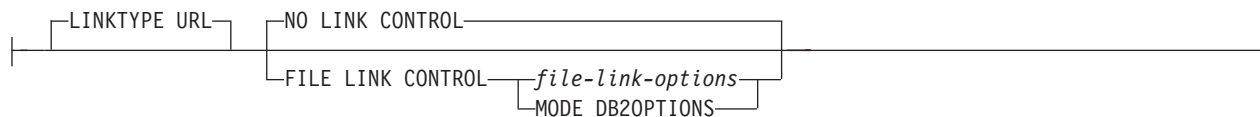


Notes:

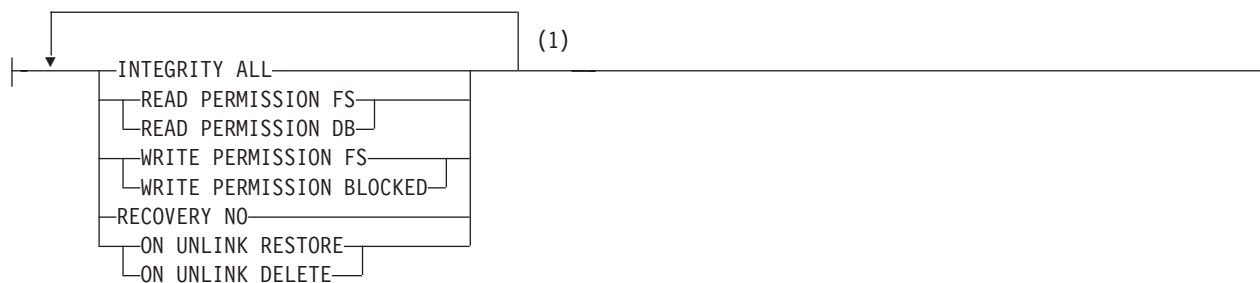
- 1 This form of the DEFAULT value can only be used with columns that are defined as a distinct type.
- 2 Each clause may be specified only once.

CREATE TABLE

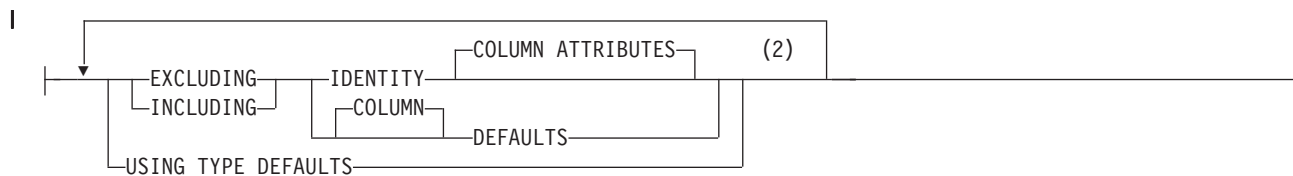
datalink-options:



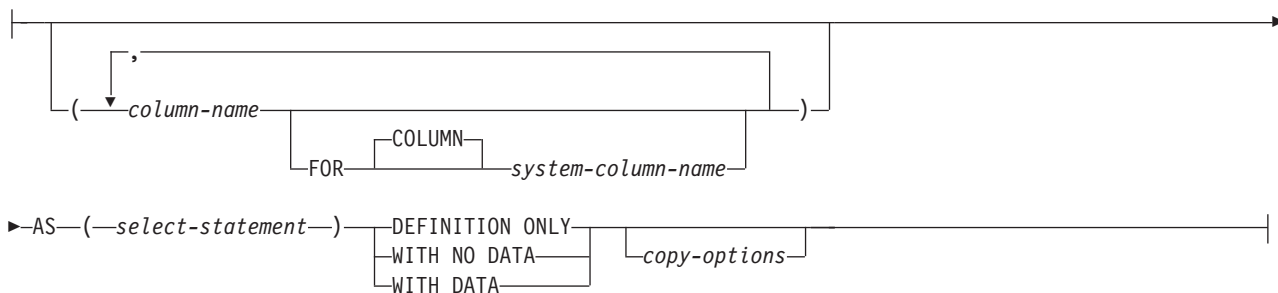
file-link-options:



copy-options:



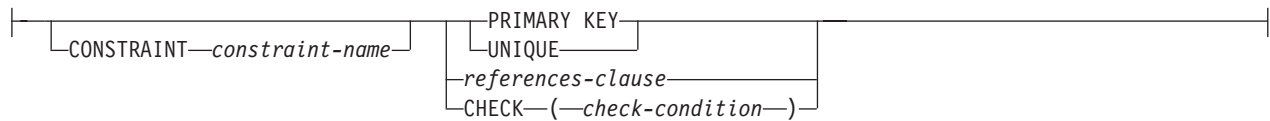
as-subquery-clause:



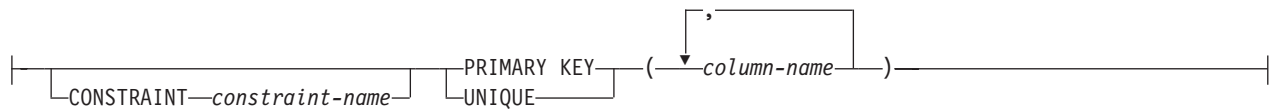
Notes:

- 1 All five *file-link-options* must be specified, but they can be specified in any order.
- 2 Each clause may be specified only once.

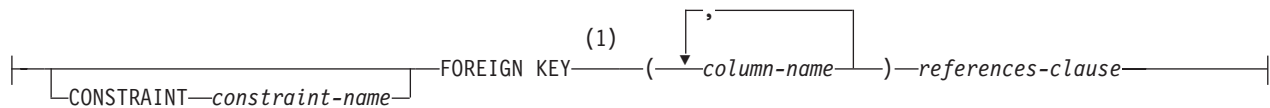
column-constraint:



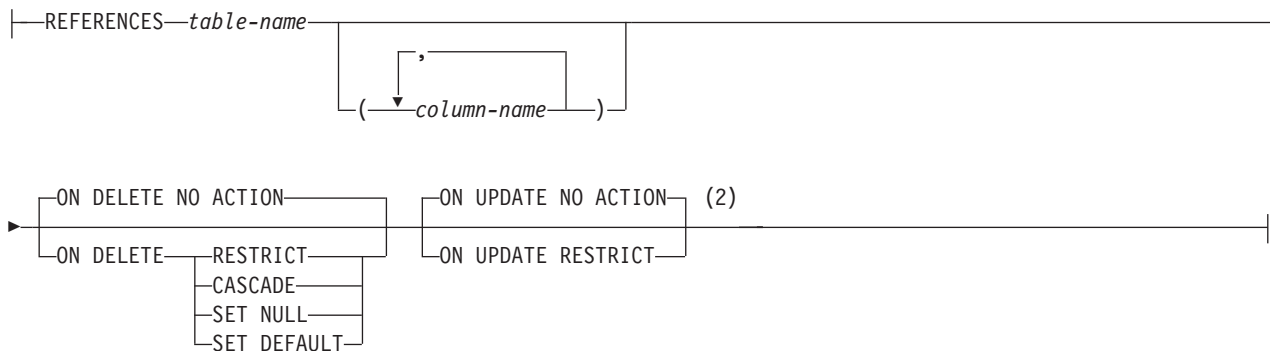
unique-constraint:



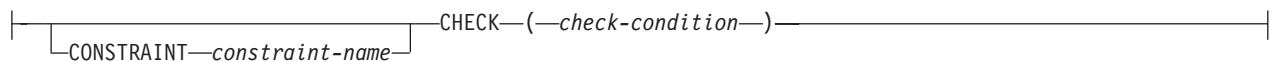
referential-constraint:



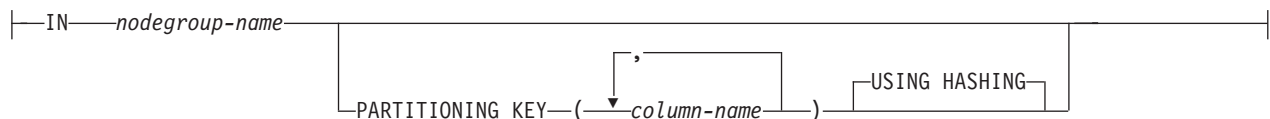
references-clause:



check-constraint:



nodegroup-clause:



Notes:

- 1 For compatibility with other products, *constraint-name* (without the `CONSTRAINT` keyword) may be specified following `FOREIGN KEY`.
- 2 The `ON DELETE` and `ON UPDATE` clauses may be specified in either order.

Description

table-name

Names the table. The name, including the implicit or explicit qualifier, must not identify an alias, file, index, table, or view that already exists at the current server.

If SQL names were specified, the table will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the table will be created in the schema that is specified by the qualifier. If not qualified, the table will be created in the current library (*CURLIB). If there is no current library, the table will be created in QGPL.

column-definition

Defines the attributes of a column. There must be at least one column definition and no more than 8000 column definitions.

The sum of the row buffer byte counts of the columns must not be greater than 32766 or, if a VARCHAR or VARGRAPHIC column is specified, 32740.

Additionally, if a LOB is specified, the sum of the row data byte counts of the columns must not be greater than 3 758 096 383 at the time of insert or update. For information on the byte counts of columns according to data type, see “Notes” on page 550.

column-name

Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one column of the table or for a *system-column-name* of the table.

FOR COLUMN *system-column-name*

Provides an OS/400 name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see “Rules for Column Name Generation” on page 553.

data-type

Specifies the data type of the column.

built-in-type

For *built-in-types*, use:

SMALLINT

For a small integer.

INTEGER or **INT**

For a large integer.

BIGINT

For a big integer.

DECIMAL(*integer,integer*) or **DEC**(*integer,integer*)

DECIMAL(*integer*) or **DEC**(*integer*)

DECIMAL or **DEC**

For a packed decimal number. The first integer is the precision of the

number; that is, the total number of digits; it can range from 1 to 31. The second integer is the scale of the number (the number of digits to the right of the decimal point). It can range from 0 to the precision of the number.

You can use `DECIMAL(p)` for `DECIMAL(p,0)`, and `DECIMAL` for `DECIMAL(5,0)`.

NUMERIC(*integer*,*integer*)

NUMERIC(*integer*)

NUMERIC

For a zoned 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, (the number of digits to the right of the decimal point). It may range from 0 to the precision of the number.

You can use `NUMERIC(p)` for `NUMERIC(p,0)`, and `NUMERIC` for `NUMERIC(5,0)`.

FLOAT

For a double-precision floating-point number.

FLOAT(*integer*)

For a single- or double-precision floating-point number, depending on the value of integer. The value of integer must be in the range 1 through 53. The values 1 through 24 indicate single-precision, the values 25 through 53 indicate double-precision. The default is 53.

REAL

For single-precision floating point.

DOUBLE PRECISION or **DOUBLE**

For double-precision floating point.

CHARACTER(*integer*) or **CHAR(*integer*)**

CHARACTER or **CHAR**

For a fixed-length character string of length *integer*. The integer can range from 1 through 32766 (32765 if null capable). If `FOR MIXED DATA` or a mixed data CCSID is specified, the range is 4 through 32766 (32765 if null capable). If the length specification is omitted, a length of 1 character is assumed.

CHARACTER VARYING (*integer*) or **CHAR VARYING (*integer*)** or **VARCHAR(*integer*)**

For a varying-length character string of maximum length *integer*, which can range from 1 through 32740 (32739 if null capable). If `FOR MIXED DATA` or a mixed data CCSID is specified, the range is 4 through 32740 (32739 if null capable).

CLOB(*integer*[K**|**M**|**G**])** or **CHAR LARGE OBJECT(*integer*[**K**|**M**|**G**])** or

CHARACTER LARGE OBJECT(*integer*[K**|**M**|**G**])**

CLOB or **CHAR LARGE OBJECT** or **CHARACTER LARGE OBJECT**

For a character large object string of the specified maximum length. The maximum length must be in the range of 1 through 2 147 483 647. If `FOR MIXED DATA` or a mixed data CCSID is specified, the range is 4 through 2 147 483 647. If the length specification is omitted, a length of 1 megabyte is assumed. A CLOB is not allowed in a distributed table.

54. This option is provided for compatibility with other products. It is recommended that `VARCHAR(integer)` or `VARGRAPHIC(integer)` be specified instead.

CREATE TABLE

integer

The maximum value for integer is 2 147 483 647. The maximum length of the string is *integer*.

integer K

The maximum value for integer is 2 097 152. The maximum length of the string is 1024 times *integer*.

integer M

The maximum value for integer is 2 048. The maximum length of the string is 1 048 576 times *integer*.

integer G

The maximum value for integer is 2. The maximum length of the string is 1 073 741 824 times *integer*.

GRAPHIC(*integer*)

GRAPHIC

For a fixed-length graphic string of length *integer*, which can range from 1 through 16383 (16382 if null capable). If the length specification is omitted, a length of 1 character is assumed.

VARGRAPHIC(*integer*) or GRAPHIC VARYING(*integer*)

For a varying-length graphic string of maximum length *integer*, which can range from 1 through 16370 (16369 if null capable).

DBCLOB(*integer*[K**|**M**|**G**])**

DBCLOB

For a double-byte character large object string of the specified maximum length.

The maximum length must be in the range of 1 through 1 073 741 823. If the length specification is omitted, a length of 1 megabyte is assumed. A DBCLOB is not allowed in a distributed table.

integer

The maximum value for integer is 1 073 741 823. The maximum length of the string is *integer*.

integer K

The maximum value for integer is 1 028 576. The maximum length of the string is 1024 times *integer*.

integer M

The maximum value for integer is 1 024. The maximum length of the string is 1 048 576 times *integer*.

integer G

The maximum value for integer is 1. The maximum length of the string is 1 073 741 824 times *integer*.

BLOB(*integer*[K**|**M**|**G**]) or BINARY LARGE OBJECT(*integer*[**K**|**M**|**G**])**

BLOB or BINARY LARGE OBJECT

For a binary large object string of the specified maximum length. The maximum length must be in the range of 1 through 2 147 483 647. If the length specification is omitted, a length of 1 megabyte is assumed. A BLOB is not allowed in a distributed table.

integer

The maximum value for integer is 2 147 483 647. The maximum length of the string is *integer*.

integer K

The maximum value for integer is 2 097 152. The maximum length of the string is 1024 times *integer*.

integer M

The maximum value for integer is 2 048. The maximum length of the string is 1 048 576 times *integer*.

integer G

The maximum value for integer is 2. The maximum length of the string is 1 073 741 824 times *integer*.

DATE

For a date.

TIME

For a time.

TIMESTAMP

For a timestamp.

DATALINK(*integer*) or DATALINK

For a DataLink of the specified maximum length. The maximum length must be in the range of 1 through 32717. If FOR MIXED DATA or a mixed data CCSID is specified, the range is 4 through 32717. The specified length must be sufficient to contain both the largest expected URL and any DataLink comment. If the length specification is omitted, a length of 200 is assumed. A DATALINK is not allowed in a distributed table.

A DATALINK value is an encapsulated value with a set of built-in scalar functions. The DLVALUE function creates 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 cannot be part of any index. Therefore, it cannot be included as a column of a primary key, foreign key, or unique constraint.

ROWID

For a row ID. Only one ROWID column is allowed in a table.

distinct-type-name

Specifies that the data type of the column is a distinct type (a user-defined data type). The length, precision, and scale of the column are respectively the length, precision, and scale of the source type of the distinct type. 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.

ALLOCATE(*integer*)

Specifies for VARCHAR, VARGRAPHIC, and LOB types the space to be reserved for the column in each row. Column values with lengths less than or equal to the allocated value are stored in the fixed-length portion of the row. Column values with lengths greater than the allocated value are

CREATE TABLE

stored in the variable-length portion of the row and require additional input/output operations to retrieve. The allocated value may range from 1 to maximum length of the string, subject to the maximum row buffer size limit. For information on the maximum row buffer size, see “Maximum row sizes” on page 550. If FOR MIXED or a mixed data CCSID is specified, the range is 4 to the maximum length of the string. If the allocated length specification is omitted, an allocated length of 0 is assumed. For VARGRAPHIC, the integer is the number of DBCS or UCS-2 characters. If a constant is specified for the default value and the ALLOCATE length is less than the length of the default value, the ALLOCATE length is assumed to be the length of the default value.

FOR BIT DATA

Specifies that the values of the column are not associated with a coded character set and are never converted. FOR BIT DATA is only valid for CHARACTER or VARCHAR columns. The CCSID of a FOR BIT DATA column is 65535. FOR BIT DATA is not allowed for CLOB columns.

FOR SBCS DATA

Specifies that the values of the column contain SBCS (single-byte character set) data. FOR SBCS DATA is the default for CHAR, VARCHAR, and CLOB columns if the default CCSID at the current server at the time the table is created is not DBCS-capable or if the length of the column is less than 4. FOR SBCS DATA is only valid for CHARACTER, VARCHAR, or CLOB columns. The CCSID of FOR SBCS DATA is determined by the default CCSID at the current server at the time the table is created.

FOR MIXED DATA

Specifies that the values of the column contain both SBCS data and DBCS data. FOR MIXED DATA is the default for CHAR, VARCHAR, and CLOB columns if the default CCSID at the current server at the time the table is created is DBCS-capable and the length of the column is greater than 3. Every FOR MIXED DATA column is a DBCS-open database field. FOR MIXED DATA is only valid for CHARACTER, VARCHAR, or CLOB columns. The CCSID of FOR MIXED DATA is determined by the default CCSID at the current server at the time the table is created.

CCSID integer

Specifies that the values of the column contain data of CCSID integer. If the integer is an SBCS CCSID, the column is SBCS data. If the integer is a mixed data CCSID, the column is mixed data and the length of the column must be greater than 3. For character columns, the CCSID must be an SBCS CCSID or a mixed data CCSID. For graphic columns, the CCSID must be a DBCS or UCS-2 CCSID. If a CCSID is not specified for a graphic column, the CCSID is determined by the default CCSID at the current server at the time the table is created. For a list of valid CCSIDs, see Appendix E, “CCSID Values” on page 863.

DEFAULT

Specifies a default value for the column. This clause cannot be specified more than once in a *column-definition*. DEFAULT cannot be specified for a ROWID column or an identity column (a column that is defined AS IDENTITY). The database manager generates default values for ROWID columns and identity columns. If a value is not specified following the DEFAULT keyword, then:

- if the column is nullable, the default value is the null value.
- if the column is not nullable, the default depends on the data type of the column:

Data type	Default value
Numeric	0
Fixed-length string	Blanks
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
<i>distinct-type</i>	The default value of the corresponding source type of the distinct type.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

constant

Specifies the constant as the default 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 “Assignments and Comparisons” on page 80. A floating-point constant must not be used for a SMALLINT, INTEGER, DECIMAL, or NUMERIC column. A decimal constant must not contain more digits to the right of the decimal point than the specified scale of the column.

USER

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value of the column. The data type of the column must be CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the USER special register.

NULL

Specifies null as the default for the column. If NOT NULL is specified, DEFAULT NULL must not be specified within the same *column-definition*.

NULL is the only default value allowed for a datalink column.

CURRENT_DATE

Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the column must be DATE or a distinct type based on a DATE.

CURRENT_TIME

Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the column must be TIME or a distinct type based on a TIME.

CURRENT_TIMESTAMP

Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the column must be TIMESTAMP or a distinct type based on a TIMESTAMP.

cast-function-name

This form of a default value can only be used with columns defined as a distinct type, BLOB, CLOB, DBCLOB, DATE, TIME or TIMESTAMP data types. The following table describes the allowed uses of these *cast-functions*.

CREATE TABLE

Data Type	Cast Function Name
Distinct type N based on a BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
Distinct type N based on a DATE, TIME, or TIMESTAMP	N (the user-defined cast function that was generated when N was created) ** or DATE, TIME, or TIMESTAMP *
Distinct type N based on other data types	N (the user-defined cast function that was generated when N was created) **
BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *
Notes:	
* The name of the function must match the name of the data type (or the source type of the distinct type) with an implicit or explicit schema name of QSYS2.	
** 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.	

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. For BLOB, CLOB, DBCLOB, DATE, TIME, and TIMESTAMP functions, the constant must be a string constant.

USER

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the source type of the distinct type of the column must be CHAR or VARCHAR with a length attribute greater than or equal to the length attribute of the USER special register.

CURRENT_DATE

Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the source type of the distinct type of the column must be DATE.

CURRENT_TIME

Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the source type of the distinct type of the column must be TIME.

CURRENT_TIMESTAMP

Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the source type of the distinct type of the column must be TIMESTAMP.

If the value specified is not valid, an error is returned.

GENERATED

Specifies that the database manager generates values for the column. GENERATED must be specified if the column is to be considered an identity column (defined with the AS IDENTITY clause). It may also be specified if the data type of the column is a ROWID (or a distinct type that is based on a ROWID). Otherwise, it must not be specified.

ALWAYS

Specifies that the database manager will always generate a value for the column when a row is inserted into the table. ALWAYS is the recommended value.

BY DEFAULT

Specifies that the database manager will generate a value for the column when a row is inserted only if a value is not specified for the column. If a value is specified, the database manager uses that value.

For a ROWID column, the database manager uses a specified value, but it must be a valid unique row ID value that was previously generated by DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries.

For an identity column, the database manager inserts a specified value but does not verify that it is a unique value for the column unless the identity column has a unique constraint or a unique index that solely specifies the identity column.

AS IDENTITY

Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC with a scale of zero, or a distinct type based on one of these types).

An identity column is implicitly NOT NULL.

START WITH *numeric-constant*

Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending sequence and the MAXVALUE for a descending sequence. This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value that is not 0, does not exceed the value of a large integer constant, and could be assigned to the column without any non-zero digits existing to the right of the decimal point. The default is 1.

If the value is positive, the sequence of values for the identity column ascends. If the value is negative, the sequence of values descends.

MAXVALUE *numeric-constant*

Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be greater than the minimum value.

If a value is not explicitly specified when the identity column is defined, this is the maximum value of the data type (and precision, if DECIMAL) for an ascending sequence; or the START WITH value, or -1 if START WITH was not specified, for a descending sequence.

CREATE TABLE

MINVALUE *numeric-constant*

Specifies the numeric constant that is the minimum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be less than the maximum value.

If a value is not explicitly specified when the identity column is defined, this is the **START WITH** value, or 1 if **START WITH** was not specified, for an ascending sequence; or the minimum value of the data type (and precision, if **DECIMAL**) for a descending sequence.

CACHE or NO CACHE

Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table.

CACHE *integer*

Specifies the number of values of the identity column sequence that the database manager preallocates and keeps in memory. The minimum value that can be specified is 2, and the maximum is the largest value that can be represented as an integer. The default is 20.

During a system failure, all cached identity column values that are yet to be assigned are lost, and thus, will never be used. Therefore, the value specified for **CACHE** also represents the maximum number of values for the identity column that could be lost during a system failure.

NO CACHE

Specifies that values for the identity column are not preallocated.

CYCLE or NO CYCLE

Specifies whether this identity column should continue to generate values after reaching either the maximum or minimum value of the sequence.

CYCLE

Specifies that values continue to be generated for this column after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches the maximum value of the sequence, it generates its minimum value. After a descending sequence reaches its minimum value of the sequence, it generates its maximum value. The maximum and minimum values for the column determine the range that is used for cycling.

When **CYCLE** is in effect, duplicate values can be generated by the database manager for an identity column. If a unique constraint or unique index exists on the identity column, and a non-unique value is generated for it, an error occurs.

NO CYCLE

Specifies that values will not be generated for the identity column once the maximum or minimum value for the sequence has been reached. This is the default.

ORDER or NO ORDER

Specifies whether the identity values must be generated in order of request.

ORDER

Specifies that the values are generated in order of request.

NO ORDER

Specifies that the values do not need to be generated in order of request. This is the default.

datalink-options

Specifies the options associated with a DATALINK data type.

LINKTYPE URL

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 linked files exist. Only the syntax of the URL will be checked. There is no database manager control over the linked files.

FILE LINK CONTROL

Specifies that a check should be made for the existence of the linked files. Additional options may be used to give the database manager further control over the linked files.

If FILE LINK CONTROL is specified, each file can only be linked once. That is, its URL can only be specified in a single FILE LINK CONTROL column in a single table.

file-link-options

Additional options to define the level of database manager control of the linked files.

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. If READ PERMISSION DB is specified, WRITE PERMISSION BLOCKED must be specified.

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

CREATE TABLE

perform 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 the database manager will support point in time recovery of files referenced by values in this column.

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 depends on the file system that contains the files. If the files are in the AIX file system, the owner is "dfmunknown". If the files are in IFS, the owner is QDLFM. 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 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

NOT NULL

Prevents the column from containing null values. Omission of NOT NULL implies that the column can be null.

column-constraint

CONSTRAINT *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

PRIMARY KEY

Provides a shorthand method of defining a primary key composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

This clause must not be specified in more than one column definition and must not be specified at all if the UNIQUE clause is specified in the column definition. The column must not be a LOB or DATALINK column.

When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in the column that makes up the primary key.

UNIQUE

Provides a shorthand method of defining a unique key composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE (C) clause is specified as a separate clause.

This clause cannot be specified more than once in a column definition and must not be specified if PRIMARY KEY is specified in the column definition. The column must not be a LOB or DATALINK column.

references-clause

The *references-clause* of a *column-definition* provides a shorthand method of defining a foreign key composed of a single column. Thus, if a *references-clause* is specified in the definition of column C, the effect is the same as if that *references-clause* were specified as part of a FOREIGN KEY clause in which C is the only identified column.

CHECK(*check-condition*)

The CHECK(*check-condition*) of a *column-definition* provides a shorthand method of defining a check constraint whose *check-condition* only references a single column. Thus, if CHECK is specified in the column definition of column C, no columns other than C can be referenced in the *check-condition* of the check constraint. The effect is the same as if the check constraint were specified as a separate clause.

ROWID or DATALINK with FILE LINK CONTROL columns cannot be referenced in a CHECK constraint. For additional restrictions see, “check-constraint” on page 548.

LIKE

table-name or *view-name*

Specifies that the columns of the table have exactly the same name and description as the columns of the identified table (*table-name*) or view (*view-name*). The name must identify a table or view that exists at the current server.

The use of LIKE is an implicit definition of n columns, where n is the number of columns in the identified table or view. The implicit definition includes the following attributes of the n columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID

If the LIKE clause is specified immediately following the *table-name* and not enclosed in parenthesis, the following column attributes are also included, otherwise they are not included (the default value and identity attributes can also be controlled by using the *copy-options*):

- Default value, if a *table-name* is specified (*view-name* is not specified)
- Nullability

CREATE TABLE

- Identity attributes
- Column heading and text (see “LABEL” on page 681)

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically include primary keys, foreign keys, or triggers. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

If the specified table or view is a non-SQL created physical file or logical file, any non-SQL attributes are removed. For example, the date and time format will be changed to ISO.

as-subquery-clause

column-name

Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one column of the table or for a system-column-name of the table.

FOR COLUMN *system-column-name*

Provides an OS/400 name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see “Rules for Column Name Generation” on page 553.

select-statement

Specifies that the columns of the table are to have the same name and description as the columns that would appear in the derived result table of the select-statement if the select-statement were to be executed. The use of AS select-statement is an implicit definition of n columns for the table, where n is the number of columns that would result from the select-statement. The implicit definition includes the following attributes of the n columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID
- Nullability
- Column heading and text (see “LABEL” on page 681)

The following attributes are not included (the default value and identity attributes may be included by using the *copy-options*):

- Default value
- Identity attributes

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically include a primary key or foreign key from a table. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

The implicitly defined columns of the table inherit the names of the columns from the result table of the *select-statement*. Therefore, a column name must be specified in the *select-statement* or in the column name list for all result columns. For result columns that are derived from expressions, constants, and

functions, the *select-statement* must include the AS column-name clause immediately after the result column or a name must be specified in the column list preceding the *select-statement*.

The *select-statement* must not refer to host variables or include parameter markers (question marks).

WITH DATA

Specifies that the *select-statement* is executed. After the table is created, the result table rows of the *select-statement* are automatically inserted into the table.

WITH NO DATA or DEFINITION ONLY

Specifies that the *select-statement* is not executed. Therefore, there is no result table with a set of rows with which to automatically populate the table.

copy-options

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table inherits the identity attribute, if any, of the columns resulting from *select-statement*, *table-name* or *view-name*. In general, the identity attribute is copied if the element of the corresponding column in the table, view, or select-statement is the name of a table column or the name of a view column that directly or indirectly maps to the name of a base table column with the identity attribute.

If the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified with the AS *select-statement* clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the *select-statement* includes multiple instances of an identity column name (that is, selecting the same column more than once).
- The select list of the *select-statement* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *select-statement* includes a set operation (union).

If INCLUDING IDENTITY is not specified, the table will not have an identity column.

EXCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table does not inherit the identity attribute, if any, of the columns resulting from the *select-statement*, *table-name*, or *view-name*.

INCLUDING COLUMN DEFAULTS

Specifies that the table inherits the default values of the columns resulting from the *select-statement*, *table-name*, or *view-name*. A default value is the value assigned to a column when a value is not specified on an INSERT.

Do not specify INCLUDING COLUMN DEFAULTS, if you specify USING TYPE DEFAULTS.

If INCLUDING COLUMN DEFAULTS is not specified, the default values are not inherited.

EXCLUDING COLUMN DEFAULTS

Specifies that the table does not inherit the default values of the columns resulting from the *select-statement*, *table-name*, or *view-name*.

USING TYPE DEFAULTS

Specifies that the default values for the table depend on the data type of the columns that result from the *select-statement*, *table-name*, or *view-name*. If the

CREATE TABLE

column is nullable, then the default value is the null value. Otherwise, the default value is as follows:

Data type	Default value
Numeric	0
Fixed-length string	Blanks
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
<i>distinct-type</i>	The default value of the corresponding source type of the distinct type.

Do not specify USING TYPE DEFAULTS, if INCLUDING COLUMN DEFAULTS is specified.

unique-constraint

CONSTRAINT *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

PRIMARY KEY(*column-name*,...)

Defines a primary key composed of the identified columns. A table can only have one primary key. Thus, this clause cannot be specified more than once and cannot be specified at all if the shorthand form has been used to define a primary key for the table. The identified columns cannot be the same as the columns specified in another UNIQUE constraint specified earlier in the CREATE TABLE statement. For example, PRIMARY KEY(A,B) would not be allowed if UNIQUE (B,A) had already been specified.

Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB or DATALINK column. The number of identified columns must not exceed 120, and the sum of their byte counts must not exceed 2000-n, where n is the number of columns specified that allow nulls. For information on byte-counts see Table 48 on page 551.

The unique index is created as part of the system physical file, not a separate system logical file. When a primary key is added, a CHECK constraint is implicitly added to enforce the rule that the NULL value is not allowed in any of the columns that make up the primary key.

UNIQUE (*column-name*,...)

Defines a unique key composed of the identified columns. The UNIQUE clause can be specified more than once. The identified columns cannot be the same as the columns specified in another UNIQUE constraint or PRIMARY KEY that was specified earlier in the CREATE TABLE statement. For determining if a unique constraint is the same as another constraint specification, the column lists are compared. For example, UNIQUE (A,B) is the same as UNIQUE (B,A).

Each column-name must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB or DATALINK column. The number of identified columns must not exceed 120, and the sum of their byte counts must not exceed 2000-n, where n is the number of columns specified that allows nulls. For information on byte-counts see Table 48 on page 551.

A unique index on the identified column is created during the execution of the CREATE TABLE statement. The unique index is created as part of the system physical file, not as a separate system logical file.

referential-constraint

CONSTRAINT *constraint-name*

Names the constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

FOREIGN KEY

Each specification of the FOREIGN KEY clause defines a referential constraint.

(*column-name*,...)

The foreign key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. The column must not be a LOB or DATALINK column. The number of identified columns must not exceed 120, and the sum of their lengths must not exceed 2000-n, where n is the number of columns specified that allow nulls.

REFERENCES *table-name*

The *table-name* specified in a REFERENCES clause must identify the table being created or a base table that already exists at the server, but it must not identify a catalog table or a global temporary 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 allowed, but not recommended.

Let T2 denote the identified parent table and let T1 denote the table being created.

The specified foreign key must have the same number of columns as the parent key of T2. The description of the *n*th column of the foreign key and the description of the *n*th column of that parent key must have identical data types and lengths.

(*column-name*,...)

The parent key of the referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than once. The column must not be a LOB or DATALINK column. The number of identified columns must not exceed 120, and the sum of their byte counts must not exceed 2000-n, where n is the number of columns specified that allow nulls. For information on byte-counts see Table 48 on page 551.

CREATE TABLE

The list of column names must be identical to the list of column names in the primary key of T2 or a UNIQUE constraint that exists on T2. The names need not be specified in the same order as in the primary key; however, they must be specified in corresponding order to the list of columns in the *foreign key* clause. If a column name list is not specified, then T2 must have a primary key. Omission of the column name list is an implicit specification of the columns of that primary key.

The referential constraint specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

ON DELETE

Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are five possible actions:

- NO ACTION (default)
- RESTRICT
- CASCADE
- SET NULL
- SET DEFAULT

SET NULL must not be specified unless some column of the foreign key allows null values.

CASCADE must not be specified if T1 contains a DataLink column with FILE LINK CONTROL.

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.
- If SET DEFAULT is specified, each column of the foreign key of each dependent of p in T1 is set to its default value.

ON UPDATE

Specifies what action is to take place on the dependent tables when a row of the parent table is updated.

The update rule applies when a row of T2 is the object of an UPDATE or propagated update 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 updated.

check-constraint

CONSTRAINT *constraint-name*

Names the check constraint. A *constraint-name* must not identify a constraint that was previously specified in the CREATE TABLE statement and must not identify a constraint that already exists at the current server.

If the clause is not specified, a unique constraint name is generated by the database manager.

CHECK (*check-condition*)

Defines a check constraint. At any time, the *check-condition* must be true or unknown for every row of the table.

The *check-condition* is a *search-condition* except:

- It can only refer to columns of the table
- It cannot reference ROWID or DATALINK with FILE LINK CONTROL columns.
- It must not contain any of the following:
 - Subqueries
 - Column functions
 - Host variables
 - Parameter markers
 - Complex expressions that contain LOBs (such as concatenation)
 - CURRENT TIMEZONE, CURRENT SCHEMA, CURRENT SERVER, CURRENT PATH, and USER special registers
 - CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP special registers
 - User-defined functions other than functions that were implicitly generated with the creation of a distinct type
 - NOW, CURDATE, and CURTIME scalar functions
 - NODENAME scalar function
 - ATAN2, DIFFERENCE, RADIANS, RAND, and SOUNDEX scalar functions
 - DLVALUE, DLURLPATH, DLURLPATHONLY, DLURLSERVER, or DLURLSCHEME scalar functions
 - DLURLCOMPLETE scalar function (for DataLinks with an attribute of FILE LINK CONTROL and READ PERMISSION DB)

For more information about search-condition, see “Search Conditions” on page 161. For more information about check constraints involving LOB data types and expressions, see the Database Programming book.

nodegroup-clause

IN *nodegroup-name*

Specifies the nodegroup across which the data in the table will be partitioned. The name must identify a nodegroup that exists at the current server. If this clause is specified, the table is created as a distributed table across all the systems in the nodegroup.

A LOB or DATALINK column is not allowed in a distributed table.

The DB2 Multisystem product must be installed to create a distributed table. For more information about distributed tables, see the DB2 Multisystem book.

PARTITIONING KEY(*column-name*,...)

Specifies the partitioning key. The partitioning key is used to determine on which node in the nodegroup a row will be placed. Each *column-name* must be an unqualified name that identifies a column of the table. The same column must not be identified more than once. If the PARTITIONING KEY clause is not specified, the first column of the primary key is used as the partitioning

CREATE TABLE

key. If there is no primary key, the first column of the table that is not floating point, date, time, or timestamp is used as the partitioning key.

The columns that make up the partitioning key must be a subset of the columns that make up any unique constraints over the table. Floating point, date, time, and timestamp columns cannot be used in a partitioning key.

USING HASHING

Specifies that the data in the partitioning key will be hashed in order to distribute the row to the appropriate server in the nodegroup.

Notes

Table attributes: Tables are created as physical files. When an table is created, the file wait time and record wait time attributes are set to the default that is specified on the WAITFILE and WAITRCD keywords of the Create Physical File (CRTLF) command.

SQL tables are created so that space used by deleted rows will be reclaimed by future insert requests. This attribute can be changed via the command *CHGPF* and specifying the *REUSEDLT(*NO)* parameter. For more information about the *CHGPF* command, see the CL Reference information in the **Programming** category of the iSeries Information Center.

When a table is created, journaling is automatically started on the journal named QSQRN in the schema.

A distributed table is created on all of the servers across which the table is distributed. For more information about distributed tables, see the DB2 Multisystem book.

Table ownership: If SQL names were specified, the *owner* of the table is the user profile with the same name as the schema into which the table is created. Otherwise, the *owner* of the table is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the table is the user profile or group user profile of the job executing the statement.

Table authority: If SQL names are used, tables are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, tables are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the table is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the table.

Maximum row sizes

There are two maximum row size restrictions referred to in the description of *column-definition*.

- The maximum row buffer size is 32766 or, if a VARCHAR, VARGRAPHIC, or LOB column is specified, 32740.
- The maximum row data size is 3 758 096 383, if a LOB is specified. If a LOB is not specified, then the maximum row data size is 32766 or, if a VARCHAR or VARGRAPHIC column is specified, 32740.

CREATE TABLE

To determine the length of a row buffer and/or row data add the corresponding length of each column of that row based on the byte counts of the data type.

The follow table gives the byte counts of columns by data type for columns that do not allow null values. If any column allows null values, one byte is required for every eight columns.

Table 48. Byte Counts of Columns by Data Type

Data Type	Row Buffer Byte Count	Row Data Byte Count
SMALLINT	2	2
INTEGER	4	4
BIGINT	8	8
DECIMAL(<i>p</i> , <i>s</i>)	The integral part of $(p/2) + 1$	The integral part of $(p/2) + 1$
NUMERIC(<i>p</i> , <i>s</i>)	<i>p</i>	<i>p</i>
FLOAT (single precision)	4	4
FLOAT (double precision)	8	8
CHAR(<i>n</i>)	<i>n</i>	<i>n</i>
VARCHAR(<i>n</i>)	$n+2$	$n+2$
CLOB(<i>n</i>)	$29+pad$	$n+29$
GRAPHIC(<i>n</i>)	$n*2$	$n*2$
VARGRAPHIC (<i>n</i>)	$n*2+2$	$n*2+2$
DBCLOB(<i>n</i>)	$29+pad$	$n*2+29$
BLOB(<i>n</i>)	$29+pad$	$n+29$
DATE	10	4
TIME	8	3
TIMESTAMP	26	10
DATALINK(<i>n</i>)	$n+24$	$n+24$
ROWID	42	28
<i>distinct-type</i>	The byte count for the source type.	The byte count for the source type.
Notes:		
<i>pad</i> is a value from 1 to 15 necessary for boundary alignment.		

Precision as described to the database:

- Floating-point fields are defined in the iSeries database with a decimal precision, not a bit precision. The algorithm used to convert the number of bits to decimal is $decimal\ precision = CEILING(n/3.31)$, where *n* is the number of bits to convert. The decimal precision is used to determine how many digits to display using interactive SQL.
- SMALLINT fields are stored with a decimal precision of 4,0.
- INTEGER fields are stored with a decimal precision of 9,0.
- BIGINT fields are stored with a decimal precision of 19,0.

LONG VARCHAR and LONG VARGRAPHIC

The non-standard syntax of LONG VARCHAR and LONG VARGRAPHIC is supported, but deprecated. The alternative standard syntax of VARCHAR(integer) and VARGRAPHIC(integer), is preferred. VARCHAR(integer) and

CREATE TABLE

VARGRAPHIC(integer) are recommended. After the CREATE TABLE statement is processed, the database manager considers a LONG VARCHAR column to be VARCHAR and a LONG VARGRAPHIC column to be VARGRAPHIC. The maximum length is calculated in a product-specific fashion that is not portable.

LONG VARCHAR ⁵⁴

For a varying length character string whose maximum length is determined by the amount of space available in the row.

LONG VARGRAPHIC ⁵⁴

For a varying length graphic string whose maximum length is determined by the amount of space available in the row.

The maximum length of a LONG column is determined as follows. Let:

- i be the sum of the row buffer byte counts of all columns in the table that are not LONG VARCHAR or LONG VARGRAPHIC
- j be the number of LONG VARCHAR and LONG VARGRAPHIC columns in the table
- k be the number of columns in the row that allow nulls.

The length of each LONG VARCHAR column is $\text{INTEGER}((32716 - i - ((k+7)/8))/j)$.

The length of each LONG VARGRAPHIC column is determined by taking the length calculated for a LONG VARCHAR column and dividing it by 2. The integer portion of the result is the length.

Using an Identity Column

When a table has an identity column, the database manager can automatically generate sequential numeric values for the column as rows are inserted into the table. Thus, identity columns are ideal for primary keys. Identity columns and ROWID columns are similar in that both types of columns contain values that the database manager generates. ROWID columns can be useful in direct-row access. ROWID columns contain values of the ROWID data type, which returns a 40-byte VARCHAR value that is not regularly ascending or descending. ROWID data values are therefore not well suited to many application uses, such as generating employee numbers or product numbers. For data that does not require direct-row access, identity columns are usually a better approach, because identity columns contain existing numeric data types and can be used in a wide variety of uses for which ROWID values would not be suitable.

When a table is recovered to a point-in-time (using RMVJRNCHG), it is possible that a large gap in the sequence of generated values for the identity column might result. For example, assume a table has an identity column that has an incremental value of 1 and that the last generated value at time T1 was 100 and the database manager subsequently generates values up to 1000. Now, assume that the table is recovered back to time T1. The generated value of the identity column for the next row that is inserted after the recovery completes will be 1001, leaving a gap from 100 to 1001 in the values of the identity column.

When CYCLE is specified duplicate values for a column may be generated even when the column is GENERATED ALWAYS, unless a unique constraint or unique index is defined on the column.

Rules for System Name Generation

There are specific instances when the system generates a system table, view, index, or column name. These instances and the name generation rules are described in the following sections.

Rules for Column Name Generation

A system-column-name is generated if the system-column-name is not specified when a table or view is created and the column-name is not a valid system-column-name.

If the column-name does not contain special characters and is longer than 10 characters, a 10-character system-column-name will be generated as:

- The first 5 characters of the name
- A 5 digit unique number

For example:

The system-column-name for LONGCOLUMNNAME would be LONGC00001

If the column name is delimited:

- The first 5 characters from within the delimiters will be used as the first 5 characters of the system-column-name. If there are fewer than 5 characters within the delimiters, the name will be padded on the right with underscore (_) characters. Lower case characters are folded to upper case characters. The only valid characters in a system-column-name are: A-Z, 0-9, @, #, \$, and _. Any other characters will be changed to the underscore (_) character. If the first character ends up as an underscore, it will be changed to the letter Q.
- A 5 digit unique number is appended to the 5 characters.

For example:

The system-column-name for "abc" would be ABC_00001
 The system-column-name for "COL2.NAME" would be COL2_00001
 The system-column-name for "C 3" would be C_3_00001
 The system-column-name for "??" would be Q_00001
 The system-column-name for "*column1" would be QC0LU00001

Rules for Table Name Generation

A system name will be generated if a table, view, alias, or index is created with either:

- A name longer than 10 characters
- A name that contains characters not valid in a system name

The SQL name or its corresponding system name may both be used in SQL statements to access the file once it is created. However, the SQL name is only recognized by DB2 UDB for iSeries and the system name must be used in other environments.

If the name does not contain special characters and is longer than 10 characters, a 10-character system name will be generated as:

- The first 5 characters of the name
- A 5 digit unique number

For example:

The system name for LONGTABLENAME would be LONGT00001

If the SQL name contains special characters, the system name is generated as:

CREATE TABLE

- The first 4 characters of the name
- A 4 digit unique number

In addition:

- All special characters are replaced by the underscore (_)
- Any trailing blanks are removed from the name
- The name is delimited by double quotes (") if the delimiters are required for the name to be a valid system name.

For example:

```
The system name for "???" would be "_0001"
The system name for "longtablename" would be "long0001"
The system name for "LONGTableName" would be LONG0001
The system name for "A b   " would be "A_b0001"
```

SQL ensures the system name is unique by searching the cross reference file. If the name already exists in the cross reference file, the number is incremented until the name is no longer a duplicate.

If a unique name cannot be determined using the above rules, an additional character is added to the counter in the name, and the number is incremented until a unique name can be found or the range is exhausted. For example, if creating "longtablename" and names "long0001" through "long9999" already exist, the name would become "lon00001".

Examples

Example 1: Given administrative authority, create a table named 'ROSSITER.INVENTORY' with the following columns:

Part number	Small integer, must not be null
Description	Character of length 0 to 24, allows nulls
Quantity on hand,	Integer allows nulls

```
CREATE TABLE ROSSITER.INVENTORY
(PARTNO      SMALLINT  NOT NULL,
DESCR       VARCHAR(24),
QONHAND     INT)
```

Example 2: Create a table named DEPARTMENT with the following columns:

Department number	Character of length 3, must not be null
Department name	Character of length 0 through 36, must not be null
Manager number	Character of length 6
Administrative dept.	Character of length 3, must not be null
Location name	Character of length 16, allows nulls

The primary key is column DEPTNO.

```
CREATE TABLE DEPARTMENT
(DEPTNO     CHAR(3)    NOT NULL,
DEPTNAME   VARCHAR(36) NOT NULL,
MGRNO      CHAR(6),
ADMRDEPT   CHAR(3)    NOT NULL,
LOCATION     CHAR(16),
PRIMARY KEY(DEPTNO))
```

CREATE TABLE

Example 3: Create a table named REORG_PROJECTS which has the same column definitions as the columns in the view PRJ_LEADER.

```
CREATE TABLE REORG_PROJECTS  
LIKE PRJ_LEADER
```

CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The authorities for the table identified in the statement, and the trigger name qualifier:
 - *EXECUTE on the library containing the subject table,
 - The ALTER (*OBJALTER), or WITH GRANT OPTION privilege (*OBJMGT) on the subject table,
 - The SELECT(*OBJOPR and *READ) on the subject table,
 - UPDATE (*UPD and *OBJOPR) on the subject table if the BEFORE UPDATE trigger contains a SET statement that modifies the NEW correlation variable,
 - SELECT(*OBJOPR and *READ) and INSERT(*OBJOPR and *ADD) privilege on the trigger name library,
 - *USE on the Add Physical File Trigger (ADDPFTRG) command,
 - If SQL naming is in effect, and if a user profile exists that matches the schema qualifier of the trigger name, and the name is different from the authorization ID of the statement, then *ALLOBJ and *SECADM special authority is required.
- Administrative authority

The authorization ID of the statement has the ALTER privilege on a table when:

- It is the owner of the table,
- It has been granted the ALTER privilege to the table, or
- It has been granted the system authorities of either *OBJALTER or *OBJMGT to the table.

The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table.

The authorization ID of the statement has the UPDATE privilege on a table when:

- It is the owner of the table,
- It has been granted the UPDATE privilege on the table or on the columns of the table, or
- It has been granted the system authorities of *OBJOPR and *UPD on the table.

In addition, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - *USE on the Create Program (CRTPGM) command

CREATE TRIGGER

- Administrative authority

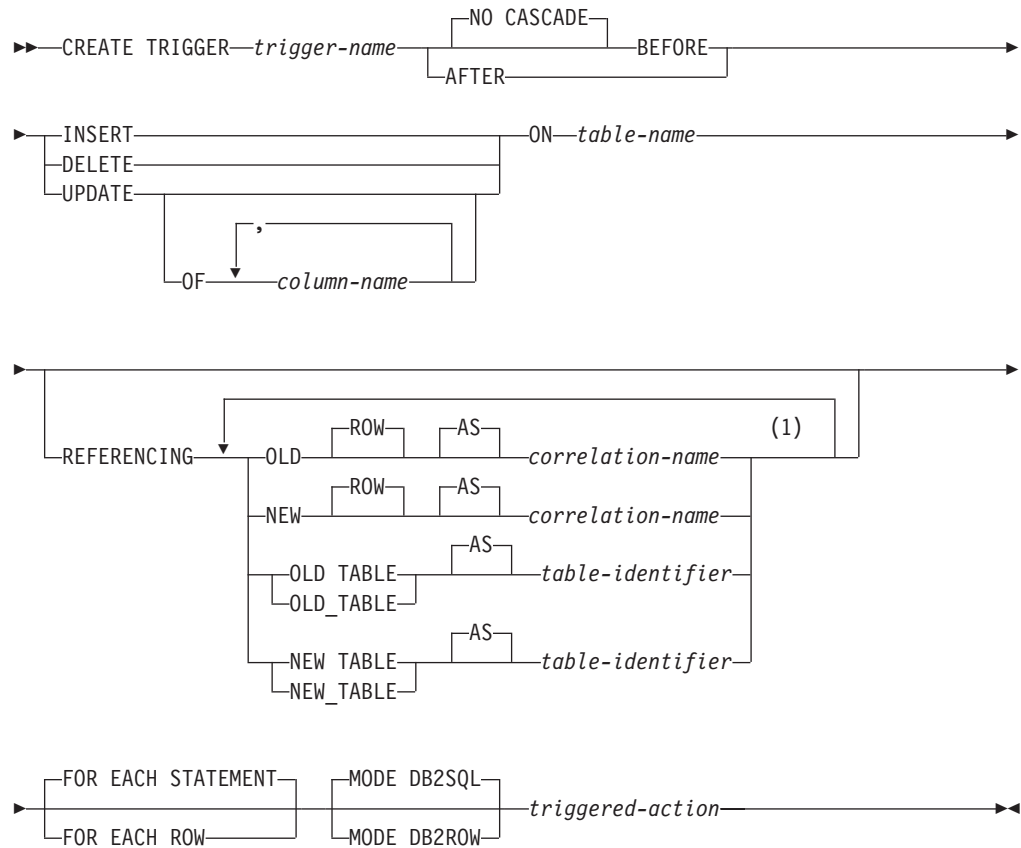
If SQL names are specified, and a user profile exists that has the same name as the library into which the trigger is created, and the name is different from the authorization ID of the statement, then the privileges held by the authorization ID of the statement must include at least one of the following:

- *ALLOBJ and *SECADM special authority
- Administrative authority

For each table or view identified in an SQL statement in the *SQL-trigger-body*, the privileges held by the authorization ID must include the privileges required to perform that SQL statement.

CREATE TRIGGER

Syntax



Notes:

- 1 The same clause must not be specified more than once.

triggered-action:

```

┌──────────────────────────┐ ┌──────────────────────────┐ ┌──────────────────────────┐
└── SET OPTION-statement ┘ └── WHEN (—search-condition—) ┘ └── SQL-trigger-body ┘

```

SQL-trigger-body:

```

┌──────────────────────────┐ ┌──────────────────────────┐
└── SQL-control-statement ┘ ┌──────────────────────────┐
└── fullselect ┘
└── ALTER-statement ┘
└── COMMENT statement ┘
└── CREATE ALIAS-statement ┘
└── CREATE DISTINCT TYPE-statement ┘
└── CREATE FUNCTION (External Scalar)-statement ┘
└── CREATE FUNCTION (External Table)-statement ┘
└── CREATE INDEX-statement ┘
└── CREATE PROCEDURE (External)-statement ┘
└── CREATE SCHEMA-statement ┘
└── CREATE TABLE-statement ┘
└── CREATE VIEW-statement ┘
└── DECLARE GLOBAL TEMPORARY TABLE-statement ┘
└── DELETE-statement ┘
└── DROP-statement ┘
└── EXECUTE IMMEDIATE-statement ┘
└── GRANT-statement ┘
└── INSERT-statement ┘
└── LABEL-statement ┘
└── LOCK TABLE-statement ┘
└── RELEASE-statement ┘
└── RELEASE SAVEPOINT-statement ┘
└── RENAME-statement ┘
└── REVOKE-statement ┘
└── SAVEPOINT-statement ┘
└── SELECT INTO-statement ┘
└── SET SCHEMA-statement ┘
└── SET PATH-statement ┘
└── SET TRANSACTION-statement ┘
└── UPDATE-statement ┘

```

Description

trigger-name

Names the trigger. The name, including the implicit or explicit qualifier, must not be the same as a trigger that already exists at the current server. QTEMP cannot be used as the *trigger-name* schema qualifier.

If SQL names were specified, the trigger will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the trigger will be created in the schema that is specified by the qualifier. If not qualified, the trigger will be created in the same schema as the subject table.

CREATE TRIGGER

If the trigger name is not a valid system name, or if a program with the same name already exists, the database manager will generate a system name. For information on the rules for generating a name, see “Rules for Table Name Generation” on page 553.

NO CASCADE

NO CASCADE is allowed for compatibility with other products and is not used by DB2 UDB for iSeries.

BEFORE

Specifies that the trigger is a before trigger. The database manager executes the *triggered-action* before it applies any changes caused by an insert, delete, or update operation on the subject table. It also specifies that the *triggered-action* does not activate other triggers because the *triggered-action* of a before trigger cannot contain any updates.

AFTER

Specifies that the trigger is an after trigger. The database manager executes the *triggered-action* after it applies any changes caused by an insert, delete, or update operation on the subject table. It also specifies that the *triggered-action* does not activate other triggers because the *triggered-action* of a BEFORE trigger cannot contain any updates.

INSERT

Specifies that the trigger is an insert trigger. The database manager executes the *triggered-action* whenever there is an insert operation on the subject table.

DELETE

Specifies that the trigger is a delete trigger. The database manager executes the *triggered-action* whenever there is a delete operation on the subject table.

A DELETE trigger cannot be added to a table with a referential constraint of ON DELETE CASCADE.

UPDATE

Specifies that the trigger is an update trigger. The database manager executes the *triggered-action* whenever there is an update operation on the subject table.

An UPDATE trigger event cannot be added to a table with a referential constraint of ON DELETE SET NULL.

If an explicit *column-name* list is not specified, an update operation on any column of the subject table, including columns that are subsequently added with the ALTER TABLE statement, activates the *triggered-action*.

OF *column-name*, ...

Each *column-name* specified must be a column of the subject table, and must appear in the list only once. An update operation on any of the listed columns activates the *triggered-action*.

ON *table-name*

Identifies the subject table of the trigger definition. The name must identify a base table that exists at the current server, but must not identify a catalog table, a table in QTEMP, or a global temporary table.

REFERENCING

Specifies the correlation names for the transition tables 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-identifiers* identify the complete set of affected rows.

CREATE TRIGGER

Each row affected by the triggering SQL operation is available to the *triggered-action* by qualifying columns with *correlation-names* specified as follows:

OLD ROW AS *correlation-name*

Specifies a correlation name that identifies the values in the row prior to the triggering SQL operation.

NEW ROW AS *correlation-name*

Specifies a correlation name which identifies the values in the row as modified by the triggering SQL operation and 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 *table-identifier*

Specifies the name of a temporary table that identifies the values in the complete set of affected rows prior to the triggering SQL operation. The OLD TABLE includes the rows that were affected by the trigger if the current activation of the trigger was caused by statements in the *SQL-trigger-body* of a trigger.

NEW TABLE AS *table-identifier*

Specifies the name of a temporary table that identifies the state of the complete set of affected rows as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already been executed.

Only one OLD and one NEW *correlation-name* may be specified for a trigger. Only one OLD_TABLE and one NEW_TABLE *table-identifier* may be specified for a trigger. All of the *correlation-names* and *table-identifiers* must be unique from one another.

The OLD *correlation-name* and the OLD_TABLE *table-identifier* are valid only if the triggering event is either a delete operation or an update operation. For a delete operation, the OLD *correlation-name* captures the values of the columns in the deleted row, and the OLD_TABLE *table-identifier* captures the values in the set of deleted rows. For an update operation, OLD *correlation-name* captures the values of the columns of a row before the update operation, and the OLD_TABLE *table-identifier* captures the values in the set of updated rows.

The NEW ROW *correlation-name* and the NEW TABLE *table-identifier* are valid only if the triggering event is either an INSERT operation or an UPDATE operation. For both operations, the NEW ROW *correlation-name* captures the values of the columns in the inserted or updated row, and the NEW TABLE *table-identifier* captures the values in the set of inserted or updated rows. For BEFORE triggers, the values of the updated rows include the changes from any SET statements in the *triggered-action* of BEFORE triggers.

The OLD ROW and NEW ROW *correlation-name* variables cannot be modified in an AFTER trigger.

The tables below summarize the allowable combinations of correlation variables and transition tables.

CREATE TRIGGER

Granularity: **FOR EACH ROW**

MODE	Activation Time	Triggering Operation	Correlation Variables Allowed	Transition Tables Allowed	
DB2ROW	BEFORE	DELETE	OLD	NONE	
		INSERT	NEW		
		UPDATE	OLD, NEW		
	AFTER	DELETE	OLD		
		INSERT	NEW		
		UPDATE	OLD, NEW		
DB2SQL	BEFORE	DELETE	OLD	NONE	
		INSERT	NEW		
		UPDATE	OLD, NEW		
	AFTER	DELETE	OLD		OLD TABLE
		INSERT	NEW		NEW TABLE
		UPDATE	OLD, NEW		OLD TABLE, NEW TABLE

Granularity: **FOR EACH STATEMENT**

MODE	Activation Time	Triggering Operation	Correlation Variables Allowed	Transition Tables Allowed
DB2SQL	AFTER	DELETE	NONE	OLD TABLE
		INSERT		NEW TABLE
		UPDATE		OLD TABLE, NEW TABLE

A transition variable that has a character data type inherits the CCSID of the column of the subject table. During the execution of the *triggered-action*, the transition variables are treated like host variables. Therefore, character conversion might occur.

The temporary transition tables are read-only. They cannot be modified.

The scope of each *correlation-name* and each *table-identifier* is the entire trigger definition.

FOR EACH ROW

Specifies that the database manager executes the *triggered-action* for each row of the subject table that the triggering operation modifies. If the triggering operation does not modify any rows, the *triggered-action* is not executed.

FOR EACH STATEMENT

Specifies that the database manager executes the *triggered-action* only once for the triggering operation. Even if the triggering operation does not modify or delete any rows, the triggered action is still executed once.

FOR EACH STATEMENT cannot be specified for a BEFORE trigger.

FOR EACH STATEMENT cannot be specified for a MODE DB2ROW trigger.

MODE DB2SQL

MODE DB2SQL triggers are activated after all of the row operations have occurred.

MODE DB2ROW

MODE DB2ROW triggers are activated on each row operation.

MODE DB2ROW is valid for both the BEFORE and AFTER activation time.

triggered-action

Specifies the action to be performed when a trigger is activated. The *triggered-action* is composed of one or more SQL statements and by an optional condition that controls whether the statements are executed.

SET OPTION-statement

Specifies the options that will be used to create the trigger. For example, to create a debuggable trigger, the following statement could be included:

```
SET OPTION DBGVIEW = *LIST
```

For more information, see “SET OPTION” on page 733.

The options CLOSQLCSR, CNULRQD, DFTRDBCOL, DYNDFTCOL, and NAMING are not allowed in the CREATE TRIGGER statement.

The options DATFMT, DATSEP, TIMFMT, and TIMSEP cannot be used if OLD ROW or NEW ROW is specified.

WHEN (*search-condition*)

Specifies a condition that evaluates to true, false, or unknown. The triggered SQL statements are executed only if the *search-condition* evaluates to true. If the WHEN clause is omitted, the associated SQL statements are always executed.

SQL-trigger-body

Specifies a single SQL statement, including a compound statement. See Chapter 6, “SQL Control Statements” on page 779 for more information about defining SQL triggers.

A call to a procedure that issues a CONNECT, SET CONNECTION, RELEASE, DISCONNECT, COMMIT, ROLLBACK, SET TRANSACTION, and SET RESULT SETS statement is not allowed in the *triggered-action* of a trigger.

If the trigger is a BEFORE trigger, then the *SQL-trigger-body* must not contain an INSERT, UPDATE, DELETE, ALTER TABLE, COMMENT, any CREATE statement, DROP, any GRANT statement, LABEL, RENAME, or any REVOKE statement. It must not contain a reference to a procedure or function that modifies SQL data.

An UNDO handler is not allowed in a trigger.

All tables, views, aliases, distinct types, user-defined functions, and procedures referenced in the *triggered-action* must exist at the current server when the trigger is created. The table or view that an alias refers to must also exist when the trigger is created. This includes objects in library QTEMP. While objects in QTEMP can be referenced in the *triggered-action*, dropping those objects in QTEMP will not cause the trigger to be dropped.

At the time the trigger is created, the *triggered-action* is modified as a result of the CREATE trigger statement:

- Naming mode is switched to SQL naming.

CREATE TRIGGER

- All unqualified object references are explicitly qualified
- All implicit column lists (e.g. SELECT *, INSERT with no column list, UPDATE SET ROW) are expanded to be the list of actual column names.

The modified *triggered-action* is stored in the catalog.

The statements in the *triggered-action* can invoke a procedure or a user-defined function that can access a server other than the current server if the procedure or user-defined function runs in a different activation group.

Notes

Trigger ownership: If SQL names were specified, the *owner* of the trigger is the user profile with the same name as the schema into which the trigger is created. Otherwise, the *owner* of the trigger is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the trigger is the user profile or group user profile of the job executing the statement.

Trigger authority: The trigger program object authorities are:

- When SQL naming is in effect, the trigger program will be created with the public authority of *EXCLUDE, and adopt authority from the schema qualifier of the trigger-name if a user profile with that name exists. If a user profile for the schema qualifier does exist, then the owner of the trigger program will be the user profile for the schema qualifier. Note that the special authorities *ALLOBJ and *SECADM are required to create the trigger program object in the schema qualifier library if a user profile exists that has the same name as the schema qualifier, and the name is different from the authorization ID of the statement. If a user profile for the schema qualifier does not exist, then the owner of the trigger program will be the user profile or group user profile of the job executing the SQL CREATE TRIGGER statement. The group user profile will be the owner of the trigger program object, only if OWNER(*GRPPRF) was specified on the user's profile who is executing the statement. If the owner of the trigger program is a member of a group profile, and if OWNER(*GRPPRF) was specified on the user's profile, the program will run with the adopted authority of the group profile.
- When System naming is in effect, the trigger program will be created with public authority of *EXCLUDE, and adopt authority from the user or group user profile of the job executing the SQL CREATE TRIGGER statement.

Activating a trigger: Only insert, delete, or update operations can activate a trigger. A delete operation that occurs as a result of a referential constraint will not activate a trigger. Hence,

- A trigger with a DELETE trigger event cannot be added to a table with a referential constraint of ON DELETE CASCADE.
- A trigger with an UPDATE trigger event cannot be added to a table with a referential constraint of ON DELETE SET NULL or ON DELETE SET DEFAULT.

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, which may result in the activation of additional triggers. With trigger cascading, a significant chain of triggers may be activated causing significant change to the database as a result of a single delete,

CREATE TRIGGER

insert or update statement. The number of levels of cascading is limited to 200 or the maximum amount of storage allowed in the job or process, whichever comes first.

Adding triggers to enforce constraints: Adding a trigger to a table that already has rows in it will not cause the triggered actions to be executed. Thus, if the trigger is designed to enforce constraints on the data in the table, the data in the existing rows might not satisfy those constraints.

Multiple triggers: Multiple triggers that have the same triggering SQL operation and activation time can be defined on a table. The triggers are activated in the order in which they were created. For example, the trigger that was created first is executed first, the trigger that was created second is executed second.

A maximum of 300 triggers can be added to any given source table.

Adding columns to a subject table or a table referenced in the triggered action: If a column is added to the subject table after triggers have been defined, the following rules apply:

- If the trigger is an UPDATE trigger that was defined without an explicit column list, then an update to the new column will cause the activation of the trigger.
- If the SQL statements in the *triggered-action* refer to the triggering table, the new column is not accessible to the SQL statements until the trigger is recreated.
- The OLD_TABLE and NEW_TABLE transition tables will contain the new column, but the column cannot be referenced unless the trigger is recreated.

If a column is added to any table referenced by the SQL statements in the triggered-action, the new column is not accessible to the SQL statements until the trigger is recreated.

Dropping or revoking privileges on a table referenced in the triggered action: If an object such as a table, view or alias, referenced in the *triggered-action* is dropped, the access plans of the statements that reference the object will be rebuilt when the trigger is fired. If the object does not exist at that time, the corresponding INSERT, UPDATE or DELETE operation on the subject table will fail.

If a privilege that the creator of the trigger is required to have for the trigger to execute is revoked, the access plans of the statements that reference the object will be rebuilt when the trigger is fired. If the appropriate privilege does not exist at that time, the corresponding INSERT, UPDATE or DELETE operation on the subject table will fail.

Errors executing triggers: If a SIGNAL statement is executed in the *SQL-trigger-body*, an SQLCODE -438 and the SQLSTATE specified in the SIGNAL statement will be returned.

Other errors that occur during the execution of *SQL-trigger-body* statements are returned using SQLSTATE 09000.

Special Registers: The values of the special registers are saved before a trigger is activated and are restored on return from the trigger. The values of the special registers are inherited from the triggering SQL operation.

Transaction isolation: All triggers, when they are activated, perform a SET TRANSACTION statement so that all of the operations by the trigger are

CREATE TRIGGER

performed with the same isolation level as the application program that caused the trigger to be run. The user may put their own SET TRANSACTION statements in an *SQL-control-statement* in the *SQL-trigger-body* of the trigger. If the user places a SET TRANSACTION statement within the *SQL-trigger-body* of the trigger, then the trigger will run with the isolation level specified in the SET TRANSACTION statement, instead of the isolation level of the application program that caused the trigger to be run.

If the application program that caused a trigger to be activated, is running with an isolation level other than No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger will be run under commitment control and will not be committed or rolled back until the application commits its current unit of work. If ATOMIC is specified in the *SQL-trigger-body* of the trigger, and the application program that caused the ATOMIC trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), the operations within the trigger will not be run under commitment control. If the application that caused the trigger to be activated is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), then the operations of a trigger are written to the database immediately, and cannot be rolled back.

If both system triggers defined by the Add Physical File Trigger (ADDPFTRG) CL command and SQL triggers defined by the CREATE TRIGGER statement are defined for a table, it is recommended that the system triggers perform a SET TRANSACTION statement so that they are run with the same isolation level as the original application that caused the triggers to be activated. It is also recommended that the system triggers run in the Activation Group of the calling application. If system triggers run in a separate Activation Group (ACTGRP(*NEW)), then those system triggers will not participate in the unit of the work for the calling application, nor in the unit of work for any SQL triggers. System triggers that run in a separate Activation Group are responsible for committing or rolling back any database operations they perform under commitment control. Note that SQL triggers defined by the CREATE TRIGGER statement always run in the caller's Activation Group.

If the triggering application is running with commitment control, the operations of an SQL trigger, and any cascaded SQL triggers, will be captured into a sub-unit of work. If the operations of the trigger and any cascaded triggers are successful, the operations captured in the sub-unit of work will be committed or rolled back when the triggering application commits or rolls back its current unit of work. Any system triggers that run in the same Activation Group as the caller, and perform a SET TRANSACTION to the isolation level of the caller, will also participate in the sub-unit of work. If the triggering application is running without commit control, then the operations of the SQL triggers will also be run without commitment control.

If an application that causes a trigger to be activated, is running with an isolation level of No Commit (COMMIT(*NONE) or COMMIT(*NC)), and it issues an INSERT, UPDATE, or DELETE statement that encounters an error during the execution of the statement, no other the system and SQL triggers will still be activated following the error for that operation. However, some number of changes will already have been performed. If the triggering application is running with commitment control, the operations of any triggers that are captured in a sub-unit of work will be rolled back when the first error is encountered, and no additional triggers will be activated for the current INSERT, UPDATE, or DELETE statement.

Renaming or moving a table referenced in the triggered action: Any table (including the subject table) referenced in a *triggered-action* can be moved or renamed. However, the *triggered-action* will continue to reference the old name or schema. An error will occur if the referenced table is not found when the *triggered-action* is executed. Hence, you should drop the trigger and then re-create the trigger so that it refers to the renamed or moved table.

Datetime Considerations: If OLD ROW or NEW ROW is specified, the date or time constants and the string representation of dates and times in variables that are used in SQL statements in the *triggered-action* must have a format of ISO, EUR, JIS, USA, or must match the date and time formats specified when the table was created if it was created using DDS and the CRTPF CL command. If the DDS specifications contain multiple different date or time formats, the trigger cannot be created.

Operations that invalidate triggers: An *invalid trigger* is a trigger that is no longer available to be activated. If a trigger becomes invalid, no INSERT, UPDATE or DELETE operations will be allowed on the subject table. A trigger becomes invalid if:

- The SQL statements in the *triggered-action* reference the subject table, the trigger is a self-referencing trigger, and the table is duplicated using the system CRTDUPOBJ CL command, or
- If the SQL statements in the *triggered-action* reference tables or views in the from library and the objects are not found in the new library when the table is duplicated using the system CRTDUPOBJ CL command, or
- If the table is restored to a new library using the system RSTOBJ or RSTLIB CL commands, and the *triggered-action* references the subject-table, the trigger is a self-referencing trigger.

An invalid trigger must first be dropped before it can be recreated by issuing a CREATE TRIGGER statement. Note that dropping and recreating a trigger will affect the activation order of a trigger if multiple triggers for the same triggering operation and activation time are defined for the subject table.

Trigger program object: When a trigger is created, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program object is then created using the CRTPGM command. The SQL options used to create the program are the options that are in effect at the time the CREATE TRIGGER statement is executed. The program is created with ACTGRP(*CALLER).

The trigger will execute with the adopted authority of the *owner* of the trigger.

Examples

Example 1: Create two triggers that track the number of employees that a company manages. The triggering table is the EMPLOYEE table, and the triggers increment and decrement a column with the total number of employees in the COMPANY_STATS table. The COMPANY_STATS table has the following properties:

```
CREATE TABLE COMPANY_STATS
(NBEMP INTEGER,
 NBPRODUCT INTEGER,
 REVENUE DECIMAL(15,0))
```

This example uses row triggers to maintain summary data in another table.

CREATE TRIGGER

Create the first trigger, NEW_HIRE, so that it increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table, increase the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

Create the second trigger, FORM_EMP, so that it decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE, decrease the value of column NBEMP in table COMPANY_STATS by 1.

```
CREATE TRIGGER FORM_EMP
AFTER DELETE ON EMPLOYEE
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1;
END
```

Example 2: Create a trigger, REORDER, that invokes user-defined function ISSUE_SHIP_REQUEST to issue a shipping request whenever a parts record is updated and the on-hand quantity for the affected part is less than 10% of its maximum stocked quantity. User-defined function ISSUE_SHIP_REQUEST orders a quantity of the part that is equal to the part's maximum stocked quantity minus its on-hand quantity. The function eliminates any duplicate requests to order the same PARTNO and sends the unique order to the appropriate supplier.

This example also shows how to define the trigger as a statement trigger instead of a row trigger. For each row in the transition table that evaluates to true for the WHERE clause, a shipping request is issued for the part.

```
CREATE TRIGGER REORDER
AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
REFERENCING NEW_TABLE AS NTABLE
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
SELECT ISSUE_SHIP_REQUEST(MAX_STOCKED - ON_HAND, PARTNO)
FROM NTABLE
WHERE ON_HAND < 0.10 * MAX_STOCKED;
END
```

Example 3: Assume that table EMPLOYEE contains column SALARY. Create a trigger, SAL_ADJ, that prevents an update to an employee's salary that exceeds 20% and signals such an error. Have the error that is returned with an SQLSTATE of 75001 and a description. This example shows that the SIGNAL SQLSTATE statement is useful for restricting changes that violate business rules.

```
CREATE TRIGGER SAL_ADJ
AFTER UPDATE OF SALARY ON EMPLOYEE
REFERENCING OLD AS OLD_EMP
NEW AS NEW_EMP
FOR EACH ROW MODE DB2SQL
WHEN (NEW_EMP.SALARY > (OLD_EMP.SALARY *1.20))
BEGIN ATOMIC
SIGNAL SQLSTATE '75001'('Invalid Salary Increase - Exceeds 20%');
END
```

CREATE VIEW

The CREATE VIEW statement creates a view on one or more tables or views at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - *USE to the Create Logical File (CRTLF) CL command
 - *EXECUTE and *ADD to the library into which the view is created
 - *CHANGE to the data dictionary if the library into which the view is created is an SQL schema with a data dictionary
- Administrative authority

The privileges held by the authorization ID of the statement must also include at least one of the following:

- For each table and view referenced directly through the fullselect, or indirectly through views referenced in the fullselect:
 - The SELECT privilege on the table or view, and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The authorization ID of the statement has the SELECT privilege on a table when:

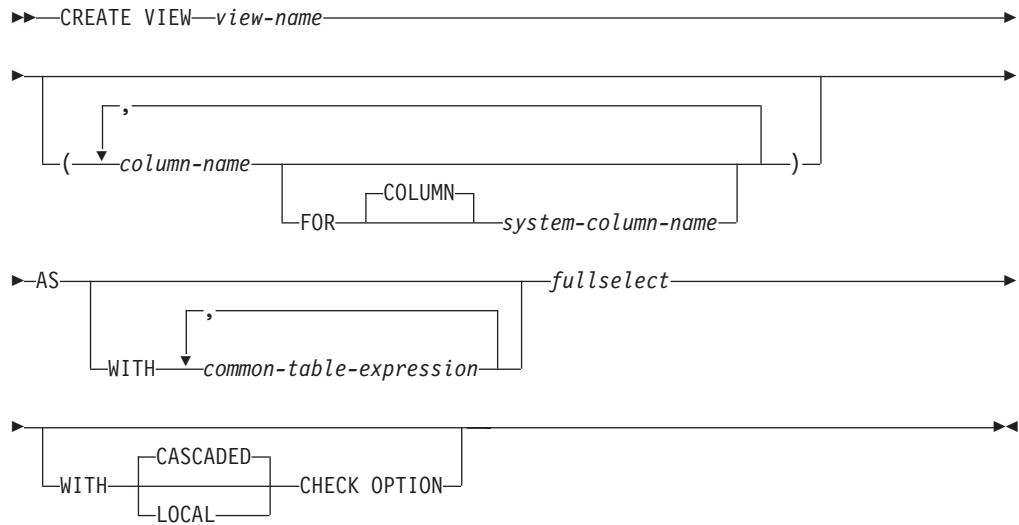
- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table.

The authorization ID of the statement has the SELECT privilege on a view when:

- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views that this view is directly or indirectly dependent on. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

CREATE VIEW

Syntax



Description

view-name

Names the view. The name, including the implicit or explicit qualifier, must not be the same as an alias, file, index, table, or view that already exists at the current server.

If SQL names were specified, the view will be created in the schema specified by the implicit or explicit qualifier.

If system names were specified, the view will be created in the schema that is specified by the qualifier. If not qualified, the view name will be created in the same schema as the first table specified on the first FROM clause (including FROM clauses in any common table expressions or nested table expression).

If a view name is not a valid system name, DB2 UDB for iSeries SQL will generate a system name. For information on the rules for generating the name, see "Rules for Table Name Generation" on page 553.

(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* and *system-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 and system names of the columns of the result table of the fullselect.

A list of column names (and system column names) must be specified if the result table of the subselect has duplicate column names, duplicate system column names, or an unnamed column. For more information about unnamed columns, see "Names of result columns" on page 333.

FOR COLUMN *system-column-name*

Provides an OS/400 name for the column. Do not use the same name for more than one column of the view or for a *column-name* of the view.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see “Rules for Column Name Generation” on page 553.

AS fullselect

Defines the view. At any time, the view consists of the rows that would result if the fullselect were executed.

fullselect must not reference host variables.

The maximum number of columns allowed in a view is 8000. The column name lengths and the length of the WHERE clause also reduce this number. The maximum number of base tables allowed in the view is 32.

For an explanation of *fullselect*, see “fullselect” on page 346.

common-table-expression defines a common table expression for use with the *fullselect* that follows. For more information see “common-table-expression” on page 350.

WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTION

Specifies that every row that is inserted or updated through the view must conform to the definition of the view. A row that does not conform to the definition of the view is a row that cannot be retrieved using that view.

CHECK OPTION must not be specified if:

- the view is read-only
- the definition of the view includes a subquery
- the definition of the view contains a non-deterministic function

If CHECK OPTION is specified for an updateable view that does not allow inserts, then the check option applies to updates only.

If 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 a CHECK OPTION. Because the definition of the view is not used, rows that do not conform to the definition of the view might be inserted or updated through the view.

CASCADED

The WITH CASCADED CHECK OPTION on a view V is inherited by any updateable view that is directly or indirectly dependent on V. Thus, if an updateable view is defined on V, the check option on V also applies to that view, even if WITH CHECK OPTION is not specified on that view. For example, consider the following updateable views:

```
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
```

SQL statement	Description of result
INSERT INTO V1 VALUES(5)	Succeeds because V1 does not have a CHECK OPTION clause and it is not dependent on any other view that has a CHECK OPTION clause.
INSERT INTO V2 VALUES(5)	Results in an error because the inserted row does not conform to the search condition of V1 which is implicitly part of the definition of V2.

CREATE VIEW

SQL statement	Description of result
INSERT INTO V3 VALUES(5)	Results in an error because V3 is dependent on V2 which has a CHECK OPTION clause and the inserted row does not conform to the definition of V2.
INSERT INTO V3 VALUES(200)	Succeeds even though it does not conform to the definition of V3 (V3 does not have the view CHECK OPTION clause specified); it does conform to the definition of V2 (which does have the view CHECK OPTION clause specified).

LOCAL

WITH LOCAL CHECK OPTION is identical to WITH CASCADED CHECK OPTION except that it is still possible to update a row so that it no longer conforms to the definition of the view when the view is defined with the WITH LOCAL CHECK OPTION. This can only happen when the view is directly or indirectly dependent on a view that was defined without either WITH CASCADED CHECK OPTION or WITH LOCAL CHECK OPTION clauses.

WITH LOCAL CHECK OPTION specifies that the search conditions of the following underlying views are checked when a row is inserted or updated:

- views that specify WITH LOCAL CHECK OPTION
- views that specify WITH CASCADED CHECK OPTION
- all underlying views of a view that specifies WITH CASCADED CHECK OPTION

In contrast, WITH CASCADED CHECK OPTION specifies that the search conditions of all underlying views are checked when a row is inserted or updated.

The difference between CASCADED and LOCAL is best shown by example. Consider the following updateable views where x and y represent either LOCAL or CASCADED:

```
V1 defined on table T0
V2 defined on V1 WITH x CHECK OPTION
V3 defined on V2
V4 defined on V3 WITH y CHECK OPTION
V5 defined on V4
```

The following table describes which views search conditions are checked during an INSERT or UPDATE operation:

Table 49. Views whose search conditions are checked during INSERT and UPDATE

View used in INSERT or UPDATE	x = LOCAL	x = CASCADED	x = LOCAL	x = CASCADED
	y = LOCAL	y = CASCADED	y = CASCADED	y = LOCAL
V1	none	none	none	none
V2	V2	V2 V1	V2	V2 V1
V3	V2	V2 V1	V2	V2 V1
V4	V4 V2	V4 V3 V2 V1	V4 V3 V2 V1	V4 V2 V1
V5	V4 V2	V4 V3 V2 V1	V4 V3 V2 V1	V4 V2 V1

Notes

View ownership: If SQL names were specified, the *owner* of the view is the user profile with the same name as the schema into which the view is created. Otherwise, the *owner* of the view is the user profile or group user profile of the job executing the statement.

If system names were specified, the *owner* of the view is the user profile or group user profile of the job executing the statement.

View authority: If SQL names are used, views are created with the system authority of *EXCLUDE on *PUBLIC. If system names are used, views are created with the authority to *PUBLIC as determined by the create authority (CRTAUT) parameter of the schema.

If the owner of the view is a member of a group profile (GRPPRF keyword) and group authority is specified (GRPAUT keyword), that group profile will also have authority to the view.

The owner always acquires the SELECT privilege on the view and the authorization to drop the view. The SELECT privilege can be granted to others only if the owner also has the authority to grant the SELECT privilege on every table or view identified in the fullselect.

The owner can also acquire the INSERT, UPDATE, and DELETE privileges on the view. If the view is not read-only, then the same privileges will be acquired on the new view as the owner has on the table or view identified in the first FROM clause of the fullselect. These privileges can be granted only if the privileges from which they are derived can also be granted.

Deletable views: A cursor is *deletable* if all of the following are true:

- the outer fullselect identifies only one base table or deletable view.
- 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 a UNION or UNION ALL operator.
- the outer fullselect does not include the DISTINCT clause.

Updatable views: A column of a view is *updatable* if all of the following are true:

- the view is deletable.
- at least one column of the view is updateable.

A column of a view is *updatable* if the corresponding result column of the *subselect* is derived solely from a column of a table or an updatable column of another view (that is, it is not derived from an expression that contains an operator, scalar function, constant, or a column that itself is derived from such expressions).

Insertable views: A view is *insertable* if at least one column of the view is updateable.

Read-only views: A view is *read-only* if it is not deletable.

A read-only view cannot be the object of an INSERT, UPDATE, or DELETE statement.

CREATE VIEW

Sort sequence: The view is created with the sort sequence in effect at the time the CREATE VIEW statement is executed. The sort sequence of the view applies to all comparisons involving SBCS data and mixed data in the view fullselect. When the view is included in a query, an intermediate result table is generated from the view fullselect. The sort sequence in effect when the query is executed applies to any selection specified in the query.

View attributes: Views are created as nonkeyed logical files. When a view is created, the file wait time and record wait time attributes are set to the default that is specified on the WAITFILE and WAITRCD keywords of the Create Logical File (CRTLF) command.

A view created over a distributed table is created on all of the systems across which the table is distributed. If a view is created over more than one distributed table, and those tables are not distributed using the same nodegroup, then the view is created only on the system that performs the CREATE VIEW statement. For more information about distributed tables, see the DB2 Multisystem book.

Identity columns: A column of a view is considered an identity column if the element of the corresponding column in the fullselect of the view definition is the name of an identity column of a table, or the name of a column of a view which directly or indirectly maps to the name of an identity column of a base table. In all other cases, the columns of a view will not get the identity property. For example:

- the select-list of the view definition includes multiple instances of the name of an identity column (that is, selecting the same column more than once)
- the view definition involves a join
- a column in the view definition includes an expression that refers to an identity column
- the view definition includes a UNION

Examples

Example 1: Create a view named MA_PROJ over 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_PROJ2
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 you are changing only one of the column names, the names of all three columns in the view must be listed in the parentheses that follow MA_PROJ.

CREATE VIEW

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 RESEMP 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, RESEMP 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
```

DECLARE CURSOR

DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java.

Authorization

No authorization is required to use this statement. However to use OPEN or FETCH for the cursor, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the SELECT statement of the cursor:
 - The SELECT privilege on the table or view, and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table

The authorization ID of the statement has the SELECT privilege on a view when:

- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views that this view is directly or indirectly dependent on. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

The SELECT statement of the cursor is one of the following:

- The prepared select-statement identified by the *statement-name*.
- The specified *select-statement*.

If *statement-name* is specified:

- The authorization ID of the statement is the run-time authorization ID unless DYNUSRPF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see “Authorization IDs and Authorization-Names” on page 57.
- The authorization check is performed when the select-statement is prepared unless DLYPRP(*YES) is specified on the CRTSQLxxx command.
- The authorization check is performed when the cursor is opened for programs compiled with the DLYPRP(*YES) parameter.

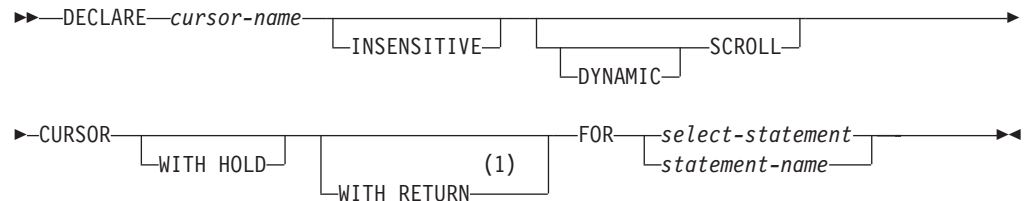
If the *select-statement* is specified:

- If USRPRF(*OWNER) or USRPRF(*NAMING) with SQL naming was specified on the CRTSQLxxx command, the authorization ID of the statement is the owner of the SQL program or package.

DECLARE CURSOR

- If USRPRF(*USER) or USRPRF(*NAMING) with system naming was specified on the CRTSQLxxx command, the authorization ID of the statement is the run-time authorization ID.
- In REXX, the authorization ID of the statement is the run-time authorization ID.
- The authorization check is performed when the cursor is opened.

Syntax



Notes:

- 1 `WITH HOLD` and `WITH RETURN` can be specified in any order.

Description

cursor-name

Names a cursor. The name must not be the same as the name of another cursor declared in your source program.

INSENSITIVE

Specifies that once the cursor is opened, it does not have sensitivity to inserts, updates, or deletes performed by this or any other activation group. If `INSENSITIVE` is specified, the cursor is read-only and a temporary result is created when the cursor is opened. In addition, the `SELECT` statement cannot contain a `FOR UPDATE` clause and the application must allow a copy of the data (`ALWCPYDTA(*OPTIMIZE)` or `ALWCPYDTA(*YES)`).

SCROLL

Specifies that the cursor is scrollable. The cursor may or may not have immediate sensitivity to inserts, updates, and deletes done by other activation groups. If `DYNAMIC` is not specified, the cursor is read-only. In addition, the `SELECT` statement cannot contain a `FOR UPDATE` clause.

DYNAMIC SCROLL

Specifies that the cursor is updateable if the result table is updateable, and that the cursor will usually have immediate sensitivity to inserts, updates, and deletes done by other application processes. However, in the following cases, the keyword `DYNAMIC` is ignored and the cursor will not have immediate sensitivity to the inserts, updates, and deletes:

- Queries that are implemented as temporary result tables. A temporary result table is created when:
 - `INSENSITIVE` was specified
 - The total length in bytes of storage for the columns specified in an `ORDER BY` clause exceeds 2000 bytes.
 - The `ORDER BY` and `GROUP BY` clauses specify different columns or columns in a different order.
 - The `ORDER BY` and `GROUP BY` clauses include a user-defined function or one of the following scalar functions: `DLVALUE`, `DLURLPATH`,

DECLARE CURSOR

DLURLPATHONLY, DLURLSERVER, DLURLSCHEME, or DLURLCOMPLETE for DataLinks with an attribute of FILE LINK CONTROL and READ PERMISSION DB.

- The UNION or DISTINCT clauses are specified.
- The ORDER BY or GROUP BY clauses specify columns which are not all from the same table.
- A logical file defined by the JOINDFT data definition specifications (DDS) keyword is joined to another file.
- A logical file that is based on multiple database file members is specified.
- The CURRENT or RELATIVE scroll options are specified on the FETCH statement when the select statement of the DECLARE CURSOR contains a GROUP BY clause.
- Queries that include a subquery where:
 - The outermost query does not provide correlated values to any inner subselects.
 - No IN, = ANY, = SOME, or <> ALL subqueries are referenced by the outermost query.

WITH HOLD

Prevents the cursor from being closed as a consequence of a commit operation. A cursor declared using the WITH HOLD clause is implicitly closed at commit time only if the connection associated with the cursor is ended during the commit operation.

When WITH HOLD is specified, a commit operation commits all the changes in the current unit of work, and releases all locks except those that are required to maintain the cursor position. Afterwards, a FETCH statement is required before a Positioned UPDATE or DELETE statement can be executed.

All cursors are implicitly closed by a CONNECT (Type 1) or rollback operation. All cursors associated with a connection are implicitly closed by a disconnect of the connection. A cursor is also implicitly closed by a commit operation if WITH HOLD is not specified, or if the connection associated with the cursor is in the release-pending state.

If a cursor is closed before the commit operation, the effect is the same as if the cursor was declared without the WITH HOLD option.

WITH RETURN

This clause indicates that the cursor is intended for use as a result set from a procedure. WITH RETURN is relevant only if the DECLARE CURSOR statement is contained with the source code for a procedure. In other cases, the precompiler may accept the clause, but it has no effect.

Within a procedure, cursors declared using the WITH RETURN clause that are still open when the SQL procedure ends define the result sets from the SQL procedure. All other open cursors in an SQL procedure are closed when the SQL procedure ends. Otherwise, any cursors open at the end of an external procedure are considered the result sets.

The result set consists of all rows from the current cursor position to the end of the result set when the procedure returns to the caller.

select-statement

Specifies the SELECT statement of the cursor. See “select-statement” on page 349 for more information.

DECLARE CURSOR

The *select-statement* must not include parameter markers (except for REXX), but can include references to host variables. In host languages, other than RPG, PL/I, and REXX, the declarations of the host variables must precede the DECLARE CURSOR statement in the source program. Host variable declarations can follow the DECLARE CURSOR statement in RPG and PL/I. In REXX, parameter markers must be used in place of host variables and the statement must be prepared.

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-name* must not be identical to a *statement-name* specified in another DECLARE CURSOR statement of the source program. See “PREPARE” on page 691 for an explanation of prepared statements.

Notes

Placement of DECLARE CURSOR: The DECLARE CURSOR statement must precede all statements that explicitly reference the cursor by name.

Result table of a cursor: 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:

- The outer fullselect identifies only one base table or deletable view.
- 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 a UNION or UNION ALL operator.
- The outer fullselect does not include the DISTINCT clause.
- The select-statement contains an ORDER BY clause, and the FOR UPDATE OF clause or DYNAMIC SCROLL are specified.
- The select-statement does not include a FOR READ ONLY clause.
- The select-statement does not include a FETCH FIRST n ROWS ONLY clause.
- The result of the outer fullselect does not make use of a temporary table.
- The select-statement does not include the SCROLL keyword unless the DYNAMIC keyword is also specified.
- The select list does not include a DATALINK column unless a FOR UPDATE OF clause is specified.

A result 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 result column is derived solely from a column of a table or an updatable column of a view. That is, at least one result column must not be derived from an expression that contains an operator, scalar function, constant, or a column that itself is derived from such expressions.

A cursor is *read-only* if it is not deletable.

If ORDER BY is specified and FOR UPDATE OF is specified, the columns in the FOR UPDATE OF clause cannot be the same as any columns specified in the ORDER BY clause.

DECLARE CURSOR

If the FOR UPDATE OF clause is omitted, only the columns in the SELECT clause of the subselect that can be updated can be changed.

Scope of a cursor: The scope of *cursor-name* is the source program in which it is defined; that is, the program submitted to the precompiler. Thus, a cursor can only be referenced by statements that are precompiled with the cursor declaration. For example, a program called from another separately compiled program cannot use a cursor that was opened by the calling program.

The scope of *cursor-name* is also limited to the thread in which the program that contains the cursor is running. For example, if the same program is running in two separate threads in the same job, the second thread cannot use a cursor that was opened by the first thread.

A cursor can only be referred to in the same instance of the program in the program stack unless CLOSQLCSR(*ENDJOB), CLOSQLCSR(*ENDSQL), or CLOSQLCSR(*ENDACTGRP) is specified on the CRTSQLxxx commands.

- If CLOSQLCSR(*ENDJOB) is specified, the cursor can be referred to by any instance of the program on the program stack.
- If CLOSQLCSR(*ENDSQL) is specified, the cursor can be referred to by any instance of the program on the program stack until the last SQL program on the program stack ends.
- If CLOSQLCSR(*ENDACTGRP) is specified, the cursor can be referred to by all instances of the module in the activation group until the activation group ends.

Although the scope of a cursor is the program in which it is declared, each package created from the program includes a separate instance of the cursor and more than one cursor can exist at run time. For example, assume a program using CONNECT (Type 2) statements connects to location X and location Y in the following sequence:

```
EXEC SQL DECLARE C CURSOR FOR ...
EXEC SQL CONNECT TO X;
EXEC SQL OPEN C;
EXEC SQL FETCH C INTO ...
EXEC SQL CONNECT TO Y;
EXEC SQL OPEN C;
EXEC SQL FETCH C INTO ...
```

The second OPEN C statement does not cause an error because it refers to a different instance of cursor C.

A SELECT statement is evaluated at the time the cursor is opened. If the same cursor is opened, closed, and then opened again, the results may be different. Multiple cursors using the same SELECT statement can be opened concurrently. They are each considered independent activities.

Blocking of data: For more efficient processing of data, the database manager can block data for read-only cursors. If a cursor is not going to be used in a Positioned UPDATE or DELETE statement, it should be declared as FOR READ ONLY.

Usage in REXX: If host variables are used on the DECLARE CURSOR statement within a REXX procedure, then the DECLARE CURSOR must be the object of a PREPARE and EXECUTE.

Cursor sensitivity: The ALWCPYDTA precompile option is ignored for DYNAMIC SCROLL cursors. If sensitivity to inserts, updates, and deletes must be maintained, a temporary copy of the data is never made unless a temporary result is required to implement the query.

Examples

Example 1: Declare C1 as the cursor of a query to retrieve data from the table DEPARTMENT. The query itself appears in the DECLARE CURSOR statement.

```
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT DEPTNO, DEPTNAME, MGRNO
        FROM DEPARTMENT
        WHERE ADMRDEPT = 'A00';
```

Example 2: Declare C2 as the cursor for a statement named STMT2.

```
EXEC SQL DECLARE C2 CURSOR FOR STMT2;
```

Example 3: Declare C3 as the cursor for a query to be used in positioned updates of the table EMPLOYEE. Allow the completed updates to be committed from time to time without closing the cursor.

```
EXEC SQL DECLARE C3 CURSOR WITH HOLD FOR
        SELECT *
        FROM EMPLOYEE
        FOR UPDATE OF WORKDEPT, PHONENO, JOB, EDLEVEL, SALARY;
```

Instead of explicitly specifying the columns to be updated, an UPDATE clause could have been used without naming the columns. This would allow all the updatable columns of the table to be updated. Since this cursor is updatable, it can also be used to delete rows from the table.

Example 4: In a C program, use the cursor C1 to fetch the values for a given project (PROJNO) from the first four columns of the EMPPROJECT table a row at a time and put them into the following host variables: EMP(CHAR(6)), PRJ(CHAR(6)), ACT(SMALLINT) and TIM(DECIMAL(5,2)). Obtain the value of the project to search for from the host variable SEARCH_PRJ (CHAR(6)). Dynamically prepare the *select-statement* to allow the project to search by to be specified when the program is executed.

```
void main ()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char      EMP[7];
    char      PRJ[7];
    char      SEARCH_PRJ[7];
    short     ACT;
    double    TIM;
    char      SELECT_STMT[201];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL INCLUDE SQLCA;

    strcpy(SELECT_STMT, "SELECT EMPNO, PROJNO, ACTNO, EMPTIME \
        FROM EMPPROJECT \
        WHERE PROJNO = ?");

    .
    .
    .
    EXEC SQL PREPARE SELECT_PRJ FROM :SELECT_STMT;

    EXEC SQL DECLARE C1 CURSOR FOR SELECT_PRJ;

    /* Obtain the value for SEARCH_PRJ from the user.    */
    .
}
```

DECLARE CURSOR

```
.
.
EXEC SQL OPEN C1 USING :SEARCH_PRJ;

EXEC SQL FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM;

if (strcmp(SQLSTATE, "02000", 5) )
{
    data_not_found();
}
else
{
    while (strcmp(SQLSTATE, "00", 2) || strcmp(SQLSTATE, "01", 2) )
    {
        EXEC SQL FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM;
    }
}

EXEC SQL CLOSE C1;
.
.
}
```

Example 5: The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT. C1 is an updateable, scrollable cursor.

```
EXEC SQL DECLARE C1 DYNAMIC SCROLL CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO
FROM TDEPT
WHERE ADMRDEPT = 'A00';
```

Example 6: Declare a cursor in order to fetch values from four columns and assign the values to host variables using the Serializable (RR) isolation level:

```
DECLARE CURSOR1 CURSOR FOR
SELECT COL1, COL2, COL3, COL4
FROM TBLNAME WHERE COL1 = :varname
WITH RR
```

DECLARE GLOBAL TEMPORARY TABLE

The DECLARE GLOBAL TEMPORARY TABLE statement defines a declared temporary table for the current application process. The declared temporary table description does not appear in the system catalog. It is not persistent and cannot be shared with other sessions. Each session that defines a declared global temporary table of the same name has its own unique description of the temporary table. When the application process ends, the temporary table is dropped.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

If the LIKE or AS select-statement clause is specified, the privileges held by the authorization ID of the statement must include at least one of the following on any table or view specified in the LIKE clause or as-subquery clause:

- The SELECT privilege for the table or view
- Ownership of the table or view
- Administrative authority

If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
 - The USAGE privilege on the distinct type, and
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

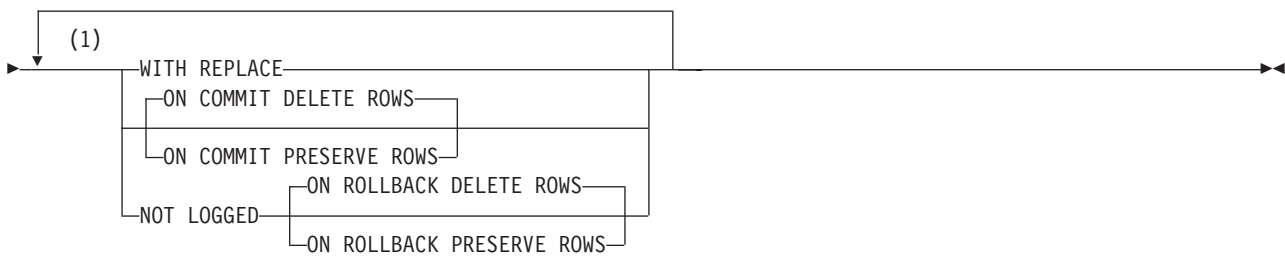
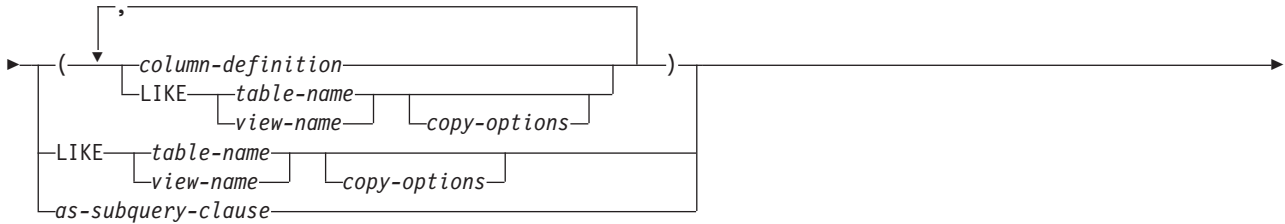
The authorization ID of the statement has the USAGE privilege on a distinct type when one of the following is true:

- It is the owner of the distinct type.
- It was granted the USAGE privilege to the distinct type.
- It was granted the system authorities of *OBJOPR and *EXECUTE to the distinct type.

DECLARE GLOBAL TEMPORARY TABLE

Syntax

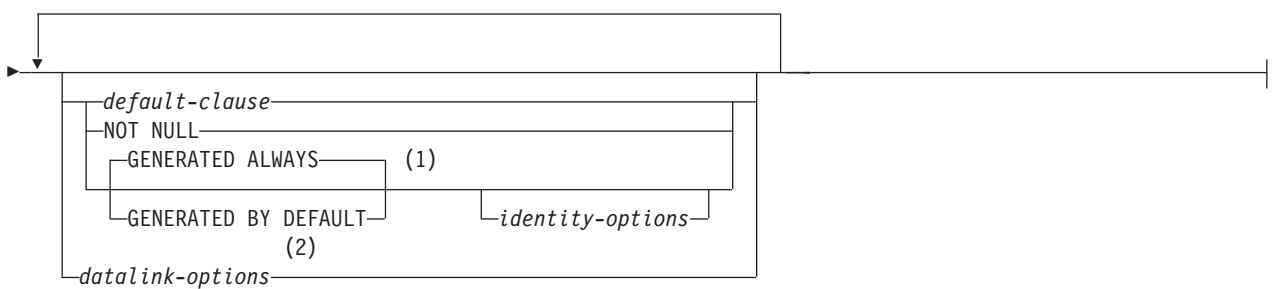
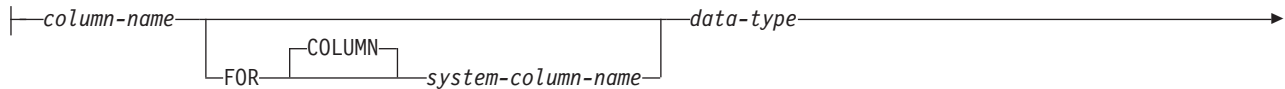
▶▶ DECLARE GLOBAL TEMPORARY TABLE *table-name* ▶▶



Notes:

- 1 Each clause may be specified only once.

column-definition:



Notes:

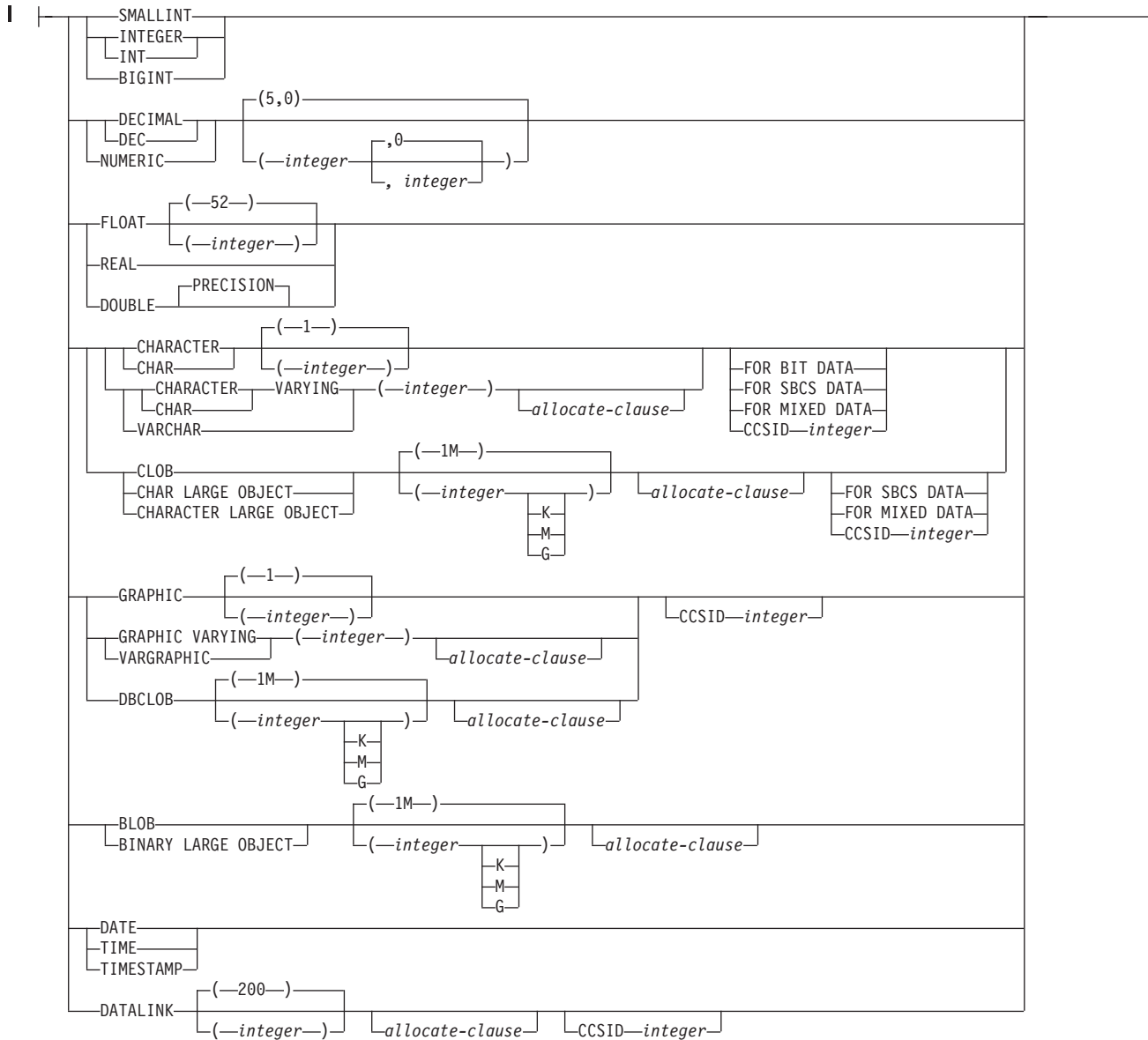
- 1 GENERATED can be specified only if the column is an identity column.
- 2 The datalink-options can only be specified for DATALINKs and distinct-types sourced on DATALINKs.

DECLARE GLOBAL TEMPORARY TABLE

data-type:

| `built-in-type`
| `distinct-type-name`

built-in-type:

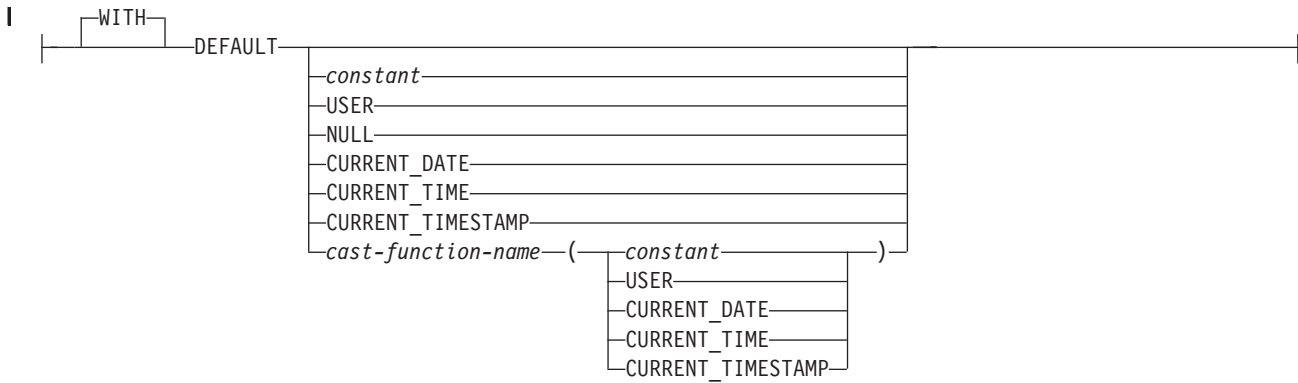


allocate-clause:

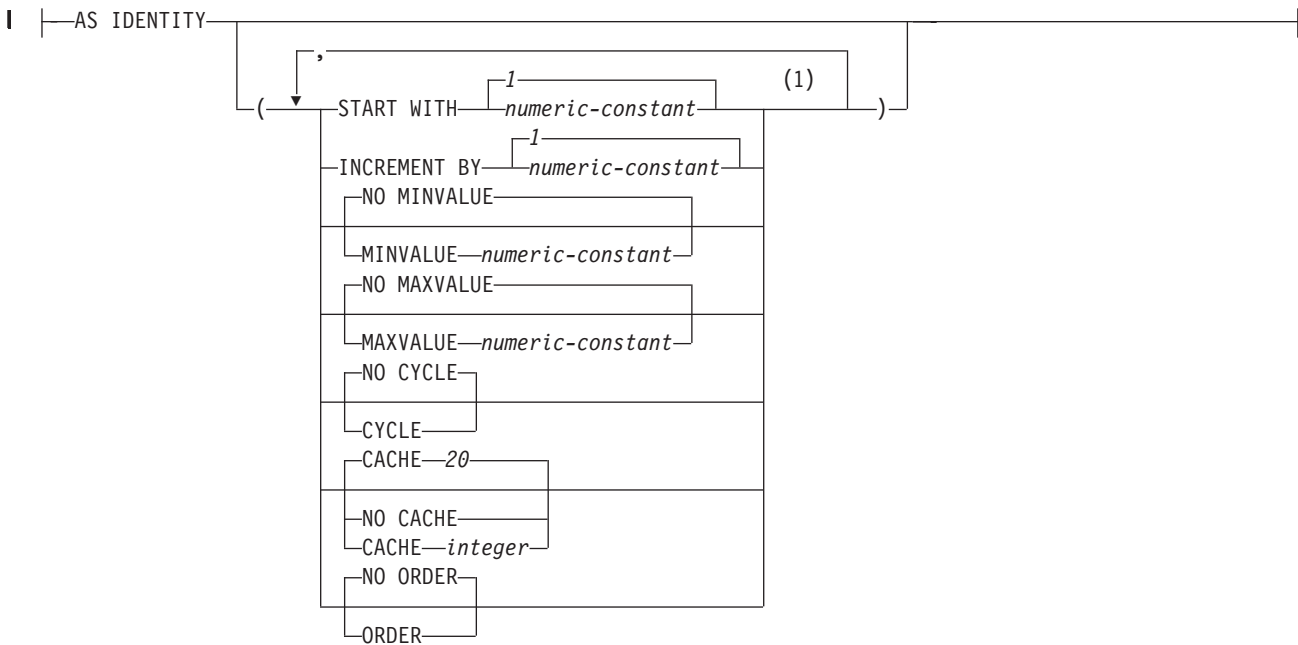
| `ALLOCATE` (*integer*)

DECLARE GLOBAL TEMPORARY TABLE

default-clause:



identity-options:

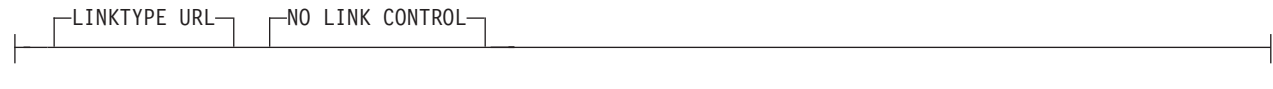


Notes:

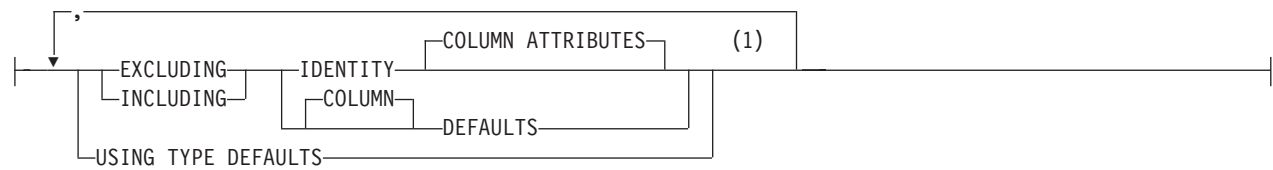
- 1 Each clause may be specified only once.

DECLARE GLOBAL TEMPORARY TABLE

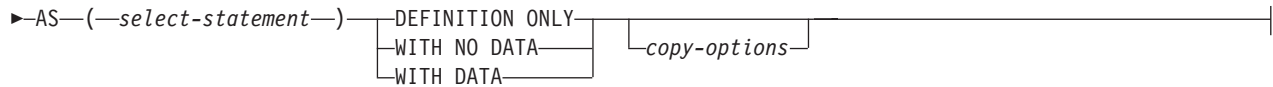
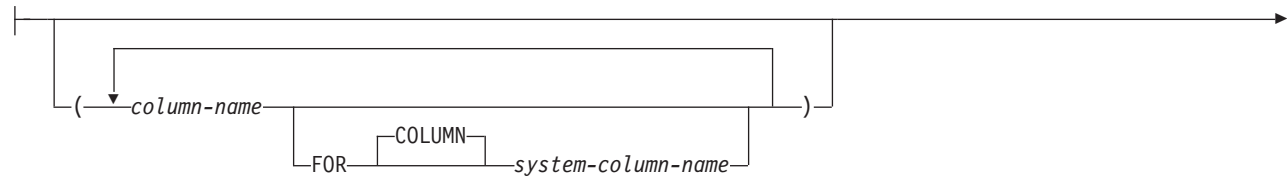
datalink-options:



copy-options:



as-subquery-clause:



Notes:

- 1 Each clause may be specified only once.

Description

table-name

Names the temporary table. The qualifier, if specified explicitly, must be SESSION. If the qualifier is not specified, it is implicitly defined to be SESSION. If a table, view, index, or alias already exists with the same name in a permanent library called SESSION:

- The declared temporary table is still defined with SESSION.table-name. An error is not issued because the resolution of a declared temporary table name does not include a permanent library.
- Any references to SESSION.table-name will resolve to the declared temporary table rather than to a permanent table, view, index, or alias with a name of SESSION.table-name.

The table will be created in library QTEMP.

column-definition

Defines the attributes of a column. There must be at least one column definition and no more than 8000 column definitions.

The sum of the row buffer byte counts of the columns must not be greater than 32766 or, if a VARCHAR or VARGRAPHIC column is specified, 32740.

DECLARE GLOBAL TEMPORARY TABLE

Additionally, if a LOB is specified, the sum of the row data byte counts of the columns must not be greater than 3.5 gigabytes. For information on the byte counts of columns according to data type, see “Notes” on page 550.

column-name

Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one column of the table or for a system-column-name of the table.

FOR COLUMN *system-column-name*

Provides an OS/400 name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see “Rules for Column Name Generation” on page 553.

data-type

Specifies the data type of the column.

built-in-type

Specifies a built-in data type. See “CREATE TABLE” on page 525 for a description of *built-in-type*.

A ROWID column or a DATALINK column with FILE LINK CONTROL cannot be specified for a global temporary table.

NOT NULL

Prevents the column from containing null values. Omission of NOT NULL implies that the column can be null.

DEFAULT

Specifies a default value for the column. This clause cannot be specified more than once in a *column-definition*. DEFAULT cannot be specified on an identity column (a column that is defined AS IDENTITY). The database manager generates default values for identity columns. If a value is not specified following the DEFAULT keyword, then:

- if the column is nullable, the default value is the null value.
- if the column is not nullable, the default depends on the data type of the column:

Data type	Default value
Numeric	0
Fixed-length string	Blanks
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
<i>distinct-type</i>	The default value of the corresponding source type of the distinct type.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

DECLARE GLOBAL TEMPORARY TABLE

constant

Specifies the constant as the default 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 “Assignments and Comparisons” on page 80. A floating-point constant must not be used for a SMALLINT, INTEGER, DECIMAL, or NUMERIC column. A decimal constant must not contain more digits to the right of the decimal point than the specified scale of the column.

USER

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value of the column. The data type of the column must be CHAR or VARCHAR with a length attribute greater than or equal to the length attribute of the USER special register.

NULL

Specifies null as the default for the column. If NOT NULL is specified, DEFAULT NULL must not be specified within the same column definition.

CURRENT_DATE

Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the column must be DATE or a distinct type based on a DATE.

CURRENT_TIME

Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the column must be TIME or a distinct type based on a TIME..

CURRENT_TIMESTAMP

Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the column must be TIMESTAMP or a distinct type based on a TIMESTAMP.

cast-function-name

This form of a default value can only be used with columns defined as a distinct type, BLOB, CLOB, DBCLOB, DATE, TIME or TIMESTAMP data types. The following table describes the allowed uses of these *cast-functions*.

Data Type	Cast Function Name
Distinct type N based on a BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
Distinct type N based on a DATE, TIME, or TIMESTAMP	N (the user-defined cast function that was generated when N was created) ** or DATE, TIME, or TIMESTAMP *
Distinct type N based on other data types	N (the user-defined cast function that was generated when N was created) **
BLOB, CLOB, or DBCLOB	BLOB, CLOB, or DBCLOB *
DATE, TIME, or TIMESTAMP	DATE, TIME, or TIMESTAMP *
Notes:	
* The name of the function must match the name of the data type (or the source type of the distinct type) with an implicit or explicit schema name of QSYS2.	
** 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.	

DECLARE GLOBAL TEMPORARY TABLE

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. For BLOB, CLOB, DBCLOB, DATE, TIME, and TIMESTAMP functions, the constant must be a string constant.

USER

Specifies the value of the USER special register at the time of INSERT or UPDATE as the default value for the column. The data type of the source type of the distinct type of the column must be CHAR or VARCHAR with a length attribute greater than or equal to the length attribute of the USER special register.

CURRENT_DATE

Specifies the current date as the default for the column. If CURRENT_DATE is specified, the data type of the source type of the distinct type of the column must be DATE.

CURRENT_TIME

Specifies the current time as the default for the column. If CURRENT_TIME is specified, the data type of the source type of the distinct type of the column must be TIME.

CURRENT_TIMESTAMP

Specifies the current timestamp as the default for the column. If CURRENT_TIMESTAMP is specified, the data type of the source type of the distinct type of the column must be TIMESTAMP.

If the value specified is not valid, an error is returned.

GENERATED

Specifies that the database manager generates values for the column. GENERATED must be specified if the column is to be considered an identity column (defined with the AS IDENTITY clause).

ALWAYS

Specifies that the database manager will always generate a value for the column when a row is inserted into the table. ALWAYS is the recommended value.

BY DEFAULT

Specifies that the database manager will generate a value for the column when a row is inserted only if a value is not specified for the column. If a value is specified, the database manager uses that value.

For an identity column, the database manager inserts a specified value but does not verify that it is a unique value for the column unless the identity column has a unique constraint or a unique index that solely specifies the identity column.

AS IDENTITY

Specifies that the column is an identity column for the table. A table can have only one identity column. AS IDENTITY can be specified only if the data type for the column is an exact numeric type with a scale of zero (SMALLINT, INTEGER, BIGINT, DECIMAL or NUMERIC with a scale of zero, or a distinct type based on one of these types).

An identity column is implicitly NOT NULL.

DECLARE GLOBAL TEMPORARY TABLE

START WITH *numeric-constant*

Specifies the first value that is generated for the identity column. The value can be any positive or negative value that could be assigned to the column without non-zero digits existing to the right of the decimal point.

If a value is not explicitly specified when the identity column is defined, the default is the MINVALUE for an ascending sequence and the MAXVALUE for a descending sequence. This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*

Specifies the interval between consecutive values of the identity column. The value can be any positive or negative value that is not 0, does not exceed the value of a large integer constant, and could be assigned to the column without any non-zero digits existing to the right of the decimal point. The default is 1.

If the value is positive, the sequence of values for the identity column ascends. If the value is negative, the sequence of values descends.

MAXVALUE *numeric-constant*

Specifies the numeric constant that is the maximum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be greater than the minimum value.

If a value is not explicitly specified when the identity column is defined, this is the maximum value of the data type (and precision, if DECIMAL) for an ascending sequence; or the START WITH value, or -1 if START WITH was not specified, for a descending sequence.

MINVALUE *numeric-constant*

Specifies the numeric constant that is the minimum value that is generated for this identity column. This value can be any positive or negative value that could be assigned to this column, but the value must be less than the maximum value.

If a value is not explicitly specified when the identity column is defined, this is the START WITH value, or 1 if START WITH was not specified, for an ascending sequence; or the minimum value of the data type (and precision, if DECIMAL) for a descending sequence.

CACHE or NO CACHE

Specifies whether to keep some preallocated values in memory. Preallocating and storing values in the cache improves the performance of inserting rows into a table.

CACHE *integer*

Specifies the number of values of the identity column sequence that the database manager preallocates and keeps in memory. The minimum value that can be specified is 2, and the maximum is the largest value that can be represented as an integer. The default is 20.

During a system failure, all cached identity column values that are yet to be assigned are lost, and thus, will never be used. Therefore, the value specified for CACHE also represents the maximum number of values for the identity column that could be lost during a system failure.

DECLARE GLOBAL TEMPORARY TABLE

NO CACHE

Specifies that values for the identity column are not preallocated.

CYCLE or NO CYCLE

Specifies whether this identity column should continue to generate values after reaching either the maximum or minimum value of the sequence.

CYCLE

Specifies that values continue to be generated for this column after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches the maximum value of the sequence, it generates its minimum value. After a descending sequence reaches its minimum value of the sequence, it generates its maximum value. The maximum and minimum values for the column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated by the database manager for an identity column. If a unique constraint or unique index exists on the identity column, and a non-unique value is generated for it, an error occurs.

NO CYCLE

Specifies that values will not be generated for the identity column once the maximum or minimum value for the sequence has been reached. This is the default.

ORDER or NO ORDER

Specifies whether the identity values must be generated in order of request.

ORDER

Specifies that the values are generated in order of request.

NO ORDER

Specifies that the values do not need to be generated in order of request. This is the default.

datalink-options

Specifies the options associated with a DATALINK data type.

LINKTYPE URL

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 linked files exist. Only the syntax of the URL will be checked. There is no database manager control over the linked files.

LIKE

table-name or view-name

Specifies that the columns defined in the specified table or view are included in this table. The *table-name* or *view-name* specified in a LIKE clause must identify the table or view that already exists at the server.

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table or view. The implicit definition includes the following attributes of the *n* columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID

DECLARE GLOBAL TEMPORARY TABLE

If the LIKE clause is specified immediately following the *table-name* and not enclosed in parenthesis, the following column attributes are also included, otherwise they are not included (the default value and identity attributes can also be controlled by using the *copy-options*):

- Default value, if a *table-name* is specified (*view-name* is not specified)
- Identity attributes
- Nullability
- Column heading and text (see “LABEL” on page 681)

If the specified table or view is a non-SQL created physical file or logical file, any non-SQL attributes are removed. For example, the date and time format will be changed to ISO.

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically include a primary key or foreign key from a table. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

as-subquery-clause

column-name

Names a column of the table. Do not qualify *column-name* and do not use the same name for more than one column of the table or for a system-column-name of the table.

FOR COLUMN *system-column-name*

Provides an OS/400 name for the column. Do not use the same name for more than one column of the table or for a column-name of the table.

If the system-column-name is not specified, and the column-name is not a valid system-column-name, a system column name is generated. For more information about how system column names are generated, see “Rules for Column Name Generation” on page 553.

select-statement

Specifies that the columns of the table are to have the same name and description as the columns that would appear in the derived result table of the select-statement if the select-statement were to be executed. The use of AS select-statement is an implicit definition of n columns for the table, where n is the number of columns that would result from the select-statement. The implicit definition includes the following attributes of the n columns (if applicable to the data type):

- Column name (and system column name)
- Data type, length, precision, and scale
- CCSID
- Nullability
- Column heading and text (see “LABEL” on page 681)

The following attributes are not included (the default value and identity attributes may be included by using the *copy-options*):

- Default value
- Identity attributes

The implicit definition does not include any other optional attributes of the identified table or view. For example, the new table does not automatically

DECLARE GLOBAL TEMPORARY TABLE

include a primary key or foreign key from a table. The new table has these and other optional attributes only if the optional clauses are explicitly specified.

The implicitly defined columns of the table inherit the names of the columns from the result table of the *select-statement*. Therefore, a column name must be specified in the *select-statement* or in the column name list for all result columns. For result columns that are derived from expressions, constants, and functions, the *select-statement* must include the AS column-name clause immediately after the result column or a name must be specified in the column list preceding the *select-statement*.

The *select-statement* must not refer to host variables or include parameter markers (question marks).

WITH DATA

Specifies that the *select-statement* is executed. After the table is created, the result table rows of the *select-statement* are automatically inserted into the table.

WITH NO DATA or DEFINITION ONLY

Specifies that the *select-statement* is not executed. Therefore, there is no result table with a set of rows with which to automatically populate the table.

copy-options

INCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table inherits the identity attribute, if any, of the columns resulting from *select-statement*, *table-name* or *view-name*. In general, the identity attribute is copied if the element of the corresponding column in the table, view, or select-statement is the name of a table column or the name of a view column that directly or indirectly maps to the name of a base table column with the identity attribute.

If the INCLUDING IDENTITY COLUMN ATTRIBUTES clause is specified with the AS *select-statement* clause, the columns of the new table do not inherit the identity attribute in the following cases:

- The select list of the *select-statement* includes multiple instances of an identity column name (that is, selecting the same column more than once).
- The select list of the *select-statement* includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list.
- The *select-statement* includes a set operation (union).

If INCLUDING IDENTITY is not specified, the table will not have an identity column.

EXCLUDING IDENTITY COLUMN ATTRIBUTES

Specifies that the table does not inherit the identity attribute, if any, of the columns resulting from the *select-statement*, *table-name*, or *view-name*.

INCLUDING COLUMN DEFAULTS

Specifies that the table inherits the default values of the columns resulting from the *select-statement*, *table-name*, or *view-name*. A default value is the value assigned to a column when a value is not specified on an INSERT.

Do not specify INCLUDING COLUMN DEFAULTS, if you specify USING TYPE DEFAULTS.

If INCLUDING COLUMN DEFAULTS is not specified, the table will not inherit the default values.

DECLARE GLOBAL TEMPORARY TABLE

EXCLUDING COLUMN DEFAULTS

Specifies that the table does not inherit the default values of the columns resulting from the *select-statement*, *table-name*, or *view-name*.

USING TYPE DEFAULTS

Specifies that the default values for the table depend on the data type of the columns that result from the *select-statement*, *table-name*, or *view-name*. If the column is nullable, then the default value is the null value. Otherwise, the default value is as follows:

Data type	Default value
Numeric	0
Fixed-length string	Blanks
Varying-length string	A string length of 0
Date	The current date at the time of INSERT
Time	The current time at the time of INSERT
Timestamp	The current timestamp at the time of INSERT
Datalink	A value corresponding to DLVALUE('','URL','')
<i>distinct-type</i>	The default value of the corresponding source type of the distinct type.

Do not specify USING TYPE DEFAULTS, if INCLUDING COLUMN DEFAULTS is specified.

WITH REPLACE

Specifies that, in the case that a declared global temporary table already exists with the specified name, the existing table is replaced with the temporary table defined by this statement (and all rows of the existing table are deleted).

When WITH REPLACE is not specified, then the name specified must not identify a declared global temporary table that already exists in the current session.

ON COMMIT

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

The ON COMMIT clause does not apply if the declared global temporary table is opened under isolation level No Commit (NC) or if a COMMIT HOLD operation is performed.

DELETE ROWS

All rows of the table will be deleted if no WITH HOLD cursor is open on the table. This is the default.

PRESERVE ROWS

Rows of the table will be preserved.

NOT LOGGED

Changes to the table are not logged, including creation of the table. When a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed and the table was changed in the unit of work (or savepoint), the changes are not rolled back. If the table was created in the unit of work (or savepoint), then that table will be dropped. If the table was dropped in the unit of work (or savepoint) then the table will be restored, but with no rows.

DECLARE GLOBAL TEMPORARY TABLE

ON ROLLBACK

Specifies the action taken on the global temporary table when a ROLLBACK operation is performed.

The ON ROLLBACK clause does not apply if the declared global temporary table was opened under isolation level No Commit (NC) or if a ROLLBACK HOLD operation is performed.

DELETE ROWS

All rows of the table will be deleted. This is the default.

PRESERVE ROWS

Rows of the table will be preserved.

Notes

Instantiation, scope, and termination: Let P denote an application process and let T be a declared temporary table in an application program in P:

- When a program in P issues a DECLARE GLOBAL TEMPORARY TABLE statement, an empty instance of T is created.
- Any program in P can reference T, and any of those references is a reference to that same instance of T. (If a DECLARE GLOBAL TEMPORARY statement is specified within a compound statement of an SQL function, SQL procedure, or trigger; the scope of the declared temporary table is the application process and not the compound statement.)

If T was declared at a remote server, the reference to T must use the same connection that was used to declare T and that connection must not have been terminated after T was declared. When the connection to the database server at which T was declared terminates, T is dropped, and its instantiated rows are destroyed.

- If T is defined with the ON COMMIT DELETE ROWS clause, when a commit operation terminates a unit of work in P and no program in P has a WITH HOLD cursor open that is dependent on T, all rows are deleted.
- If T is defined with the ON ROLLBACK DELETE ROWS clause, when a rollback operation terminates a unit of work in P, all rows are deleted.
- When the application process that declared T terminates, T is dropped, and its rows are destroyed.

Temporary table ownership: The *owner* of the table is the user profile of the job executing the statement.

Temporary table authority: When a declared temporary table is defined, PUBLIC implicitly is granted all table privileges on the table and authority to drop the table.

Referring to a declared temporary table in other SQL statements: Many SQL statements support declared temporary tables. To refer to a declared temporary table in an SQL statement other than DECLARE GLOBAL TEMPORARY TABLE, the table must be implicitly or explicitly qualified with SESSION.

If you use SESSION as the qualifier for a table name but the application process does not include a DECLARE GLOBAL TEMPORARY TABLE statement for the table name, the database manager assumes that you are not referring to a declared temporary table. The database manager resolves such table references to a permanent table.

DECLARE GLOBAL TEMPORARY TABLE

Restrictions on the use of declared temporary tables:

- Declared temporary tables cannot be specified in an ALTER TABLE, COMMENT, CREATE TRIGGER, GRANT, LABEL, LOCK, RENAME or REVOKE statement.
- Declared temporary tables cannot be specified as the parent table in referential constraints
- If a declared temporary table is referenced in a CREATE INDEX or CREATE VIEW statement, the index or view must be created in SESSION (or library QTEMP).

Examples

Example 1: Define a declared temporary table with column definitions for an employee number, salary, commission, and bonus.

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
  (EMPNO CHAR(6) NOT NULL,
   SALARY DECIMAL(9, 2),
   BONUS DECIMAL(9, 2),
   COMM DECIMAL(9, 2))
ON COMMIT PRESERVE ROWS
```

Example 2: Assume that base table USER1.EMPTAB exists and that it contains three columns, one of which is an identity column. Declare a temporary table that has the same column names and attributes (including identity attributes) as the base table.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPTAB1
  LIKE USER1.EMPTAB
  INCLUDING IDENTITY
  ON COMMIT PRESERVE ROWS
```

In the above example, the database manager uses SESSION as the implicit qualifier for TEMPTAB1.

DECLARE PROCEDURE

DECLARE PROCEDURE

The DECLARE PROCEDURE statement defines an external procedure.

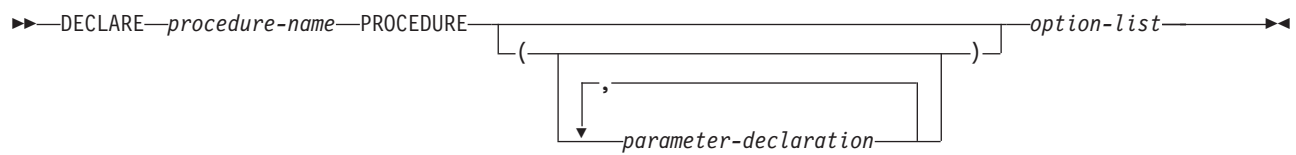
Invocation

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.

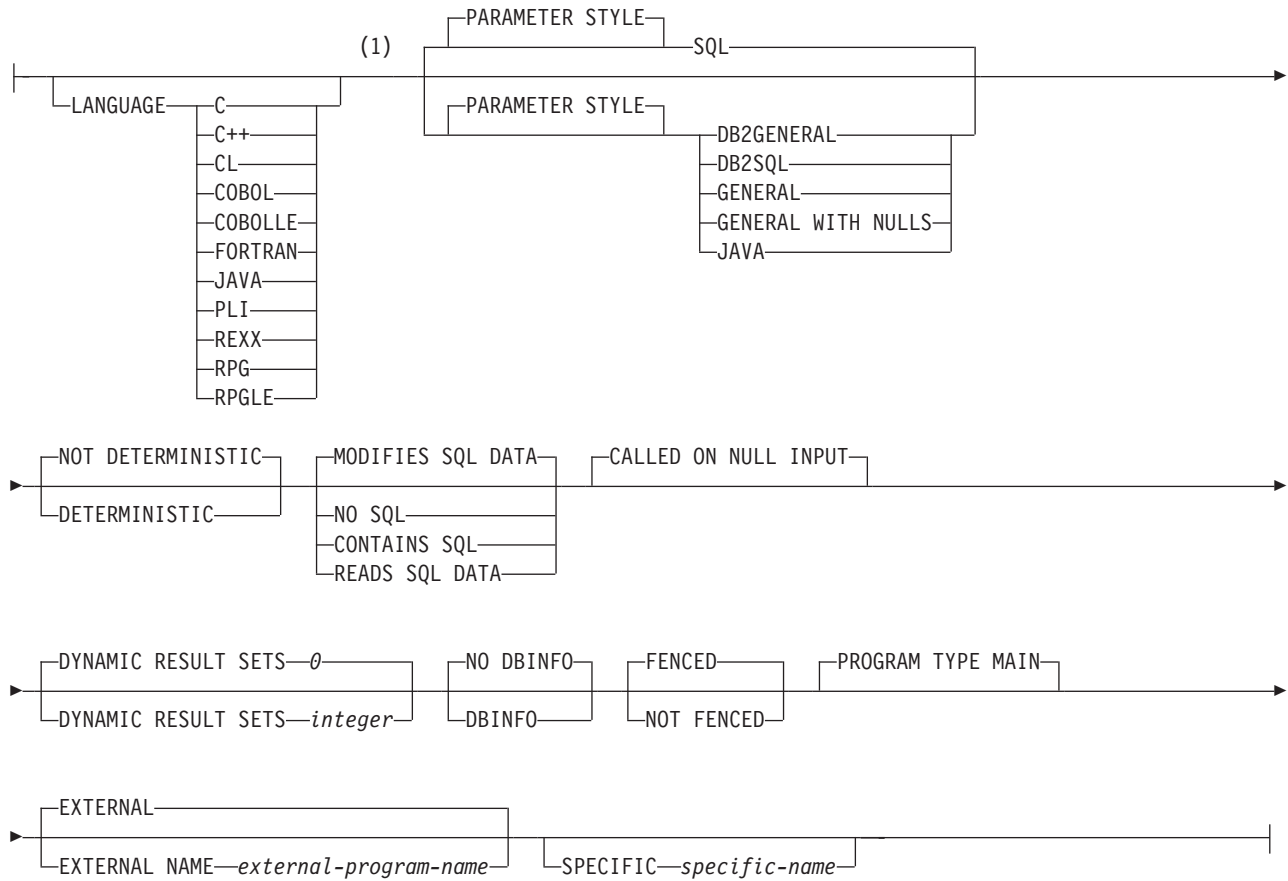
Syntax



parameter-declaration:



option-list:



Notes:

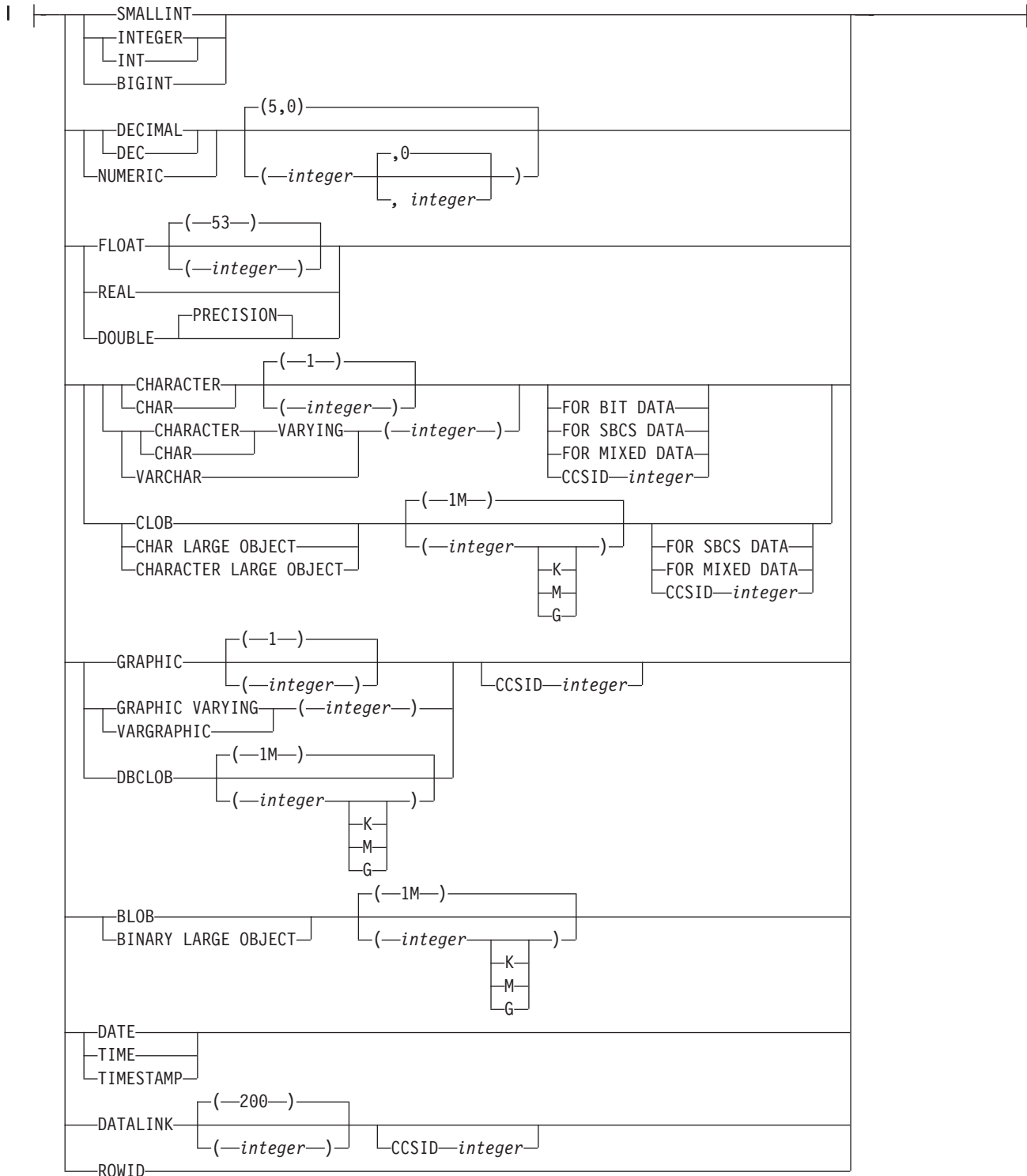
- 1 The optional clauses can be specified in a different order.

DECLARE PROCEDURE

data-type:



built-in-type:



Description

procedure-name

Names the procedure. The name must not be the same as the name of another procedure declared in your source program.

(parameter-declaration,...)

Specifies the number of parameters of the procedure and the data type of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. Although not required, you can give each parameter a name.

The maximum number of parameters allowed in an SQL procedure is 255.

IN

Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned.⁵⁵

OUT

Identifies the parameter as an output parameter that is returned by the procedure.

A DataLink or a distinct type based on a DataLink may not be specified as an output parameter.

INOUT

Identifies the parameter as both an input and output parameter for the procedure.

A DataLink or a distinct type based on a DataLink may not be specified as an input and output parameter.

parameter-name

Names the parameter. The name cannot be the same as any other *parameter-name* for the procedure.

data-type

Specifies the data type of the parameter.

The data type must be valid for the language specified in the language clause. All data types are valid for SQL procedures. DataLinks are not valid for external procedures. For more information about data types, see "CREATE TABLE" on page 525, and the SQL Programming Concepts book.

If a CCSID is specified, the parameter will be converted to that CCSID prior to passing it to the procedure. If a CCSID is not specified, the CCSID is determined by the default CCSID at the current server at the time the procedure is called.

AS LOCATOR

Specifies that the input parameter is a locator to the value rather than the actual value. You can specify AS LOCATOR only if the input parameter has a LOB data type or a distinct type based on a LOB data type.

DYNAMIC RESULT SETS *integer*

Specifies the maximum number of result sets that can be returned from the procedure. *integer* must be greater than or equal to zero. If zero is specified, no result sets are returned. A procedure can have any number of result sets, but at any time, only 100 procedures can have result sets that are waiting to be

55. When the language type is REXX, all parameters must be input parameters.

DECLARE PROCEDURE

fetched. If the SET RESULT SETS statement is issued, the number of results returned is the minimum of the number of result sets specified on this keyword and the SET RESULTS SET statement.

Result sets are only returned if the procedure is called from a iSeries Access client or the SQL Call Level Interface. For more information about result sets see "SET RESULT SETS" on page 750.

LANGUAGE

Specifies the language that the external program is written in. The language clause is required if the external program is a REXX procedure.

If LANGUAGE is not specified, the LANGUAGE is determined from the program attribute information associated with the external program. If the program attribute information associated with the program does not identify a recognizable language, then the language is assumed to be C.

C

The external program is written in C.

C++

The external program is written in C++.

CL

The external program is written in CL.

COBOL

The external program is written in COBOL.

COBOLLE

The external program is written in ILE COBOL.

FORTRAN

The external program is written in FORTRAN.

JAVA

The external program is written in JAVA.

PLI

The external program is written in PL/I.

REXX

The external program is a REXX procedure.

RPG

The external program is written in RPG.

RPGLE

The external program is written in ILE RPG.

SPECIFIC *specific-name*

Specifies a qualified or unqualified name that uniquely identifies the procedure. The *specific-name*, including the implicit or explicit qualifier, must be the same as the *procedure-name*.

If no qualifier is specified, the implicit or explicit qualifier of the *procedure-name* is used. If a qualifier is specified, the qualifier must be the same as the explicit or implicit qualifier of the *procedure-name*.

If *specific-name* is not specified, it is the same as the procedure name.

DETERMINISTIC or NOT DETERMINISTIC

Specifies whether the procedure returns the same results each time the procedure is called with the same IN and INOUT arguments.

DECLARE PROCEDURE

NOT DETERMINISTIC

The procedure always returns the same results each time the procedure is called with the same IN and INOUT arguments, provided the referenced data in the database has not changed.

DETERMINISTIC

The procedure may not return the same result each time the procedure is called with the same IN and INOUT arguments, even when the referenced data in the database has not changed.

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL

Specifies which SQL statements, if any, may be executed in the procedure or any routine called from this procedure. See Appendix C, "Characteristics of SQL Statements" on page 835 for a detailed list of the SQL statements that can be executed under each data access indication.

CONTAINS SQL

Specifies that SQL statements that neither read nor modify SQL data can be executed by the procedure.

NO SQL

Specifies that the procedure cannot execute any SQL statements.

READS SQL DATA

Specifies that SQL statements that do not modify SQL data can be included in the procedure.

MODIFIES SQL DATA

Specifies that the procedure can execute any SQL statement except statements that are not supported in procedures.

CALLED ON NULL INPUT

Specifies that the function is to be invoked, if any, or all, argument values are null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

FENCED or NOT FENCED

This parameter is allowed for compatibility with other products and is not used by DB2 UDB for iSeries.

PROGRAM TYPE MAIN

Specifies that the procedure executes as a main routine.

DBINFO

Specifies that the database manager should pass a structure containing status information to the procedure. Table 50 contains a description of the DBINFO structure. Detailed information about the DBINFO structure can be found in include file SQLUDF in QSYSINC.H.

DBINFO is only allowed with PARAMETER STYLE DB2SQL.

Table 50. DBINFO fields

Field	Data Type	Description
Relational database	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The run-time authorization ID.

DECLARE PROCEDURE

Table 50. DBINFO fields (continued)

Field	Data Type	Description
CCSID Information	INTEGER INTEGER INTEGER INTEGER CHAR(8)	The CCSID information of the job. The following information identifies the CCSID: <ul style="list-style-type: none"> • SBCS CCSID • DBCS CCSID • Mixed CCSID • Indication of which of the first three CCSIDs is appropriate. • Reserved If a CCSID is not explicitly specified for a parameter on the DECLARE PROCEDURE statement, the input string is assumed to be encoded in the CCSID of the job at the time the function is executed. If the CCSID of the input string is not the same as the CCSID of the parameter, the input string passed to the external function will be converted before calling the external program.
Target Column	VARCHAR(128) VARCHAR(128) VARCHAR(128)	Not applicable for a call to a procedure.
Version and release	CHAR(8)	The version, release, and modification level of the database manager.
Platform	INTEGER	The server's platform type.

EXTERNAL NAME *external-program-name*

Specifies the program that will be executed when the procedure is called by the CALL statement. The program name must identify a program that exists at the server. The program cannot be an ILE service program.

The validity of the name is checked at the server. If the format of the name is not correct, an error is returned.

If *external-program-name* is not specified, the external program name is assumed to be the same as the procedure name.

PARAMETER STYLE

Specifies the conventions used for passing parameters to and returning the values from procedures:

SQL

Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. The parameters are defined to be in the following order:

- The first N parameters are the parameters that are specified on the DECLARE PROCEDURE statement.
- N parameters for indicator variables for the parameters.
- A CHAR(5) output parameter for SQLSTATE. The SQLSTATE returned indicates the success or failure of the procedure. The SQLSTATE returned is assigned by the external program.
The user may set the SQLSTATE to any valid value in the external program to return an error or warning from the function.
- A VARCHAR(517) input parameter for the fully qualified procedure name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(70) output parameter for the message text.

DECLARE PROCEDURE

For more information about the parameters passed, see the include `sqludf` in the appropriate source file. For example, for C, `sqludf` can be found in `QSYSINC/H`.

PARAMETER STYLE SQL cannot be used with LANGUAGE JAVA.

DB2GENERAL

Specifies that the procedure will use a parameter passing convention that is defined for use with Java methods.

PARAMETER STYLE DB2GENERAL can only be specified with LANGUAGE JAVA. For details on passing parameters in JAVA, see the Developer Kit for Java book.

DB2SQL

Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. DB2SQL is identical to the SQL parameter style, except that the following additional parameter may be passed as the last parameter:

- A parameter for the `dbinfo` structure, if `DBINFO` was specified on the DECLARE PROCEDURE statement.

For more information about the parameters passed, see the include `sqludf` in the appropriate source file. For example, for C, `sqludf` can be found in `QSYSINC/H`.

PARAMETER STYLE DB2SQL cannot be used with LANGUAGE JAVA.

GENERAL

Specifies that the procedure will use a parameter passing mechanism where the procedure receives the parameters specified on the CALL. Additional arguments are not passed for indicator variables.

PARAMETER STYLE GENERAL cannot be used with LANGUAGE JAVA.

GENERAL WITH NULLS

Specifies that in addition to the parameters on the CALL statement as specified in GENERAL, another argument is passed to the procedure. This additional argument contains an indicator array with an element for each of the parameters of the CALL statement. In C, this would be an array of short ints. For more information about how the indicators are handled, see the SQL Programming Concepts book.

PARAMETER STYLE GENERAL WITH NULLS cannot be used with LANGUAGE JAVA.

JAVA

Specifies that the procedure will use a parameter passing convention that conforms to the Java language and SQLJ Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. For increased portability, you should write Java procedures that use the PARAMETER STYLE JAVA conventions.

PARAMETER STYLE JAVA can only be specified with LANGUAGE JAVA. For details on passing parameters in JAVA, see the Developer Kit for Java book.

Note that language of the external procedure determines how the parameters are passed. For example, in C, any VARCHAR or CHAR parameters are passed as NUL-terminated strings. For more information, see the SQL Programming Concepts book.

DECLARE PROCEDURE

Notes

DECLARE PROCEDURE scope: The scope of the *procedure-name* is the source program in which it is defined; that is, the program submitted to the precompiler. Thus, a program called from another separately compiled program or module will not use the attributes from a DECLARE PROCEDURE statement in the calling program.

DECLARE PROCEDURE rules: The DECLARE PROCEDURE statement should precede all CALL statements that reference that procedure.

The maximum number of parameters allowed in DECLARE PROCEDURE is 255. If GENERAL WITH NULLS is specified, the maximum is 254. If parameter style SQL is specified, only 90 parameters are allowed. The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program used to compile the external program.

The DECLARE PROCEDURE statement only applies to static CALL statements. It does not apply to any dynamically prepared CALL statements or a CALL statement where the procedure name is identified by a host variable.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords VARIANT and NOT VARIANT can be used as synonyms for NOT DETERMINISTIC and DETERMINISTIC.
- The keywords NULL CALL and NOT NULL CALL can be used as synonyms for CALLED ON NULL INPUT and RETURNS NULL ON NULL INPUT.
- The keywords SIMPLE CALL can be used as a synonym for GENERAL.
- The value DB2GENRL may be used as a synonym for DB2GENERAL.

Example

Declare an external procedure PROC1 in a C program. When the procedure is called using the CALL statement, a COBOL program named PGM1 in library LIB1 will be called.

```
EXEC SQL
  DECLARE PROC1 PROCEDURE
    (CHAR(10), CHAR(10))
    EXTERNAL NAME LIB1.PGM1
    LANGUAGE COBOL GENERAL;

EXEC SQL
  CALL PROC1 ('FIRSTNAME ', 'LASTNAME ');
```

DECLARE STATEMENT

The DECLARE STATEMENT statement is used for program documentation. It declares names that are used to identify prepared SQL statements.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. This statement is not allowed in Java or REXX.

Authorization

None required.

Syntax



Description

statement-name

Lists one or more names that are used in your program to identify prepared SQL statements.

Example

This example shows the use of the DECLARE STATEMENT statement in a C program.

```

EXEC SQL INCLUDE SQLDA;
void main ()
{
    EXEC SQL BEGIN DECLARE SECTION ;
    char src_stmt[32000];
    char sqlda[32000]
    EXEC SQL END DECLARE SECTION ;
    EXEC SQL INCLUDE SQLCA ;

    strcpy(src_stmt,"SELECT DEPTNO, DEPTNAME, MGRNO \
        FROM DEPARTMENT \
        WHERE ADMRDEPT = 'A00'");

    EXEC SQL DECLARE OBJ_STMT STATEMENT;

    (Allocate storage from SQLDA)

    EXEC SQL DECLARE C1 CURSOR FOR OBJ_STMT;

    EXEC SQL PREPARE OBJ_STMT FROM :src_stmt;
    EXEC SQL DESCRIBE OBJ_STMT INTO :sqlda;

    (Examine SQLDA) (Set SQLDATA pointer addresses)

    EXEC SQL OPEN C1;

    while (strncmp(SQLSTATE, "00000", 5) )
    {
        EXEC SQL FETCH C1 USING DESCRIPTOR :sqlda;
    }
}
  
```

DECLARE STATEMENT

```
        (Print results)
    }
EXEC SQL CLOSE C1;
return;
}
```

DECLARE VARIABLE

The DECLARE VARIABLE statement is used to assign a subtype or CCSID other than the default to a host variable.

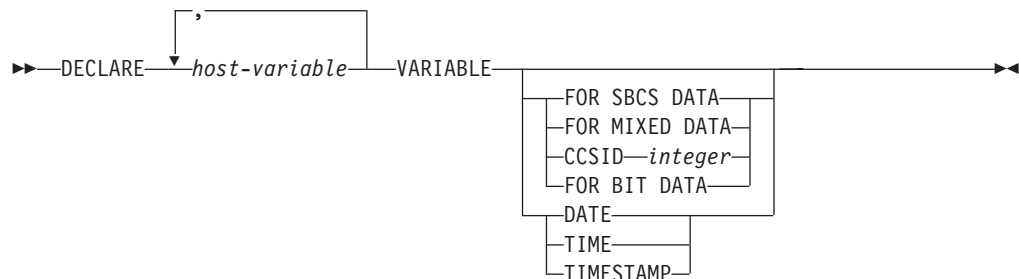
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

host-variable

Names a character or graphic-string host variable defined in the program. An indicator variable cannot be specified for the host-variable. The host-variable definition may either precede or follow a DECLARE VARIABLE statement that refers to that variable.

FOR BIT DATA

Specifies that the values of the host-variable are not associated with a coded character set and, therefore, are never converted. The CCSID of a FOR BIT DATA host variable is 65535. FOR BIT DATA cannot be specified for graphic host-variables.

FOR SBCS DATA

Specifies that the values of the host variable contain SBCS (single-byte character set) data. FOR SBCS DATA is the default if the CCSID attribute of the job at the application requester is not DBCS-capable or if the length of the host variable is less than 4. The CCSID of FOR SBCS DATA is determined by the CCSID attribute of the job at the application requester. FOR SBCS DATA cannot be specified for graphic host-variables.

FOR MIXED DATA

Specifies that the values of the host variable contain both SBCS data and DBCS data. FOR MIXED DATA is the default if the CCSID attribute of the job at the application requester is DBCS-capable and the length of the host variable is greater than 3. The CCSID of FOR DBCS DATA is determined by the CCSID attribute of the job at the application requester. FOR MIXED DATA cannot be specified for graphic host-variables.

DECLARE VARIABLE

CCSID *integer*

Specifies that the values of the host variable contain data of CCSID integer. If the integer is an SBCS CCSID, the host variable is SBCS data. If the integer is a mixed data CCSID, the host variable is mixed data. For character host variables, the CCSID specified must be an SBCS or mixed CCSID.

If the variable has a graphic string data type, the CCSID specified must be a DBCS or UCS-2 CCSID. For a list of valid CCSIDs, see Appendix E, "CCSID Values" on page 863. Consider specifying CCSID 13488 to indicate UCS-2 data. If a CCSID is not specified, the CCSID of the graphic string variable will be the associated DBCS CCSID for the job.

DATE

Specifies that the values of the host variable contain data that is a date.

TIME

Specifies that the values of the host variable contain data that is a time.

TIMESTAMP

Specifies that the values of the host variable contain data that is a timestamp.

Notes

Placement Restrictions: The DECLARE VARIABLE statement can be specified anywhere in an application program that SQL statements are valid with the following exceptions:

- If the host language is COBOL or RPG, the DECLARE VARIABLE statement must occur before an SQL statement that refers to a host variable specified in the DECLARE VARIABLE statement.
- If DATE, TIME, or TIMESTAMP is specified for a NUL-terminated character string in C, the length of the C declaration will be reduced by one.

Precompiler Rules: The following situations result in an error message during precompile:

- A reference is made to a variable that does not exist.
- A reference is made to a numeric variable.
- A reference is made to a variable that has been referred to already.
- A reference is made to a variable that is not unique.
- The DECLARE VARIABLE statement occurs after an SQL statement where the SQL statement and the DECLARE VARIABLE statement refer to the same variable.
- The FOR BIT DATA, FOR SBCS DATA, or FOR MIXED DATA clause is specified for a graphic host variable.
- A SBCS or mixed CCSID is specified for a graphic host variable.
- A DBCS or UCS-2 CCSID is specified for a character host variable.
- DATE, TIME, or TIMESTAMP is specified for a host variable that is not character.
- The length of a host variable used for DATE, TIME, or TIMESTAMP is not long enough for the minimum date, time, or timestamp value.

Example

In this example, declare C program variables *fred* and *pete* as mixed data, and *jean* and *dave* as SBCS data with CCSID 37.

DECLARE VARIABLE

```
void main ()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char fred[10];
    EXEC SQL DECLARE :fred VARIABLE FOR MIXED DATA;

    decimal(6,0) mary;
    char pete[4];
    EXEC SQL DECLARE :pete VARIABLE FOR MIXED DATA;

    char jean[30];
    char dave[9];
    EXEC SQL DECLARE :jean, :dave VARIABLE CCSID 37;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL INCLUDE SQLCA;
    ...
}
```

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 Searched DELETE statement can be embedded in an application program or issued interactively. A positioned DELETE must be embedded in an application program. Both Searched DELETE and Positioned DELETE are executable statements that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table or view identified in the statement:
 - The DELETE privilege on the table or view, and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The authorization ID of the statement has the DELETE privilege on a table when:

- It is the owner of the table,
- It has been granted the DELETE privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *DLT on the table.

The authorization ID of the statement has the DELETE privilege on a view when:⁵⁶

- It has been granted the DELETE privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *DLT on the view, and the system authority *DLT on the first table or view that this view is directly or indirectly dependent on. That is, the first table or view referenced in the view definition, and if a view is referenced, the first table or view referenced in its definition, and so forth.

If the *search-condition* in a Searched DELETE contains a reference to a column of the table or view, then the privileges held by the authorization ID of the statement must also include one of the following:

- The SELECT privilege on the table or view
- Administrative authority

If *search-condition* includes a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following:

- For each table or view identified in the subquery:

56. When a view is created, the owner does not necessarily acquire the DELETE privilege on the view. The owner only acquires the DELETE privilege if the view allows deletes and the owner also has the DELETE privilege on the first table referenced in the subselect.

- The SELECT privilege on the table or view, and
- The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table.

The authorization ID of the statement has the SELECT privilege on a view when:

- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views that this view is directly or indirectly dependent on. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

Syntax

Searched DELETE:

```

▶▶—DELETE FROM—table-name
                   |
                   |—view-name
                   |
                   |—correlation-clause
                   |
                   |—————▶
▶—WHERE—search-condition | isolation-clause

```

Positioned DELETE:

```

▶▶—DELETE FROM—table-name
                   |
                   |—view-name
                   |
                   |—correlation-clause
                   |
                   |—————▶
▶—WHERE CURRENT OF—cursor-name

```

isolation-clause:

```

|—WITH—
      |
      |—NC
      |—UR
      |—CS
      |—RS
      |—RR

```

Description

FROM *table-name* or *view-name*

Identifies the table or view from which rows are to be deleted. The name must identify a table or view that exists at the current server, but it must not identify a catalog table, a view of a catalog table, or a view that is not deletable. For an explanation of deletable views, see “CREATE VIEW” on page 569.

DELETE

correlation-clause

Can be used within the *search-condition* to designate the table or view and column names of the table or view. For an explanation of *correlation-clause*, see Chapter 4, “Queries” on page 329. For an explanation of *correlation-name*, see “Correlation Names” on page 108.

WHERE

Specifies the rows to be deleted. The clause can be omitted, or a *search-condition* or *cursor-name* can be specified. 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 161. 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 of the subquery used in applying the *search condition*. In actuality, a subquery with no correlated references may be executed only 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 the DELETE statement or a dependent table with a delete rule of CASCADE, SET NULL, or SET DEFAULT, the subquery is completely evaluated before any rows are deleted.

CURRENT OF *cursor-name*

Identifies the cursor to be used in the delete operation. The *cursor-name* must identify a declared cursor as explained in the Notes for the DECLARE CURSOR statement.

The table or view identified must also be specified in the FROM clause of the *select-statement* of the cursor and the cursor must be deletable. For an explanation of deletable cursors, see “DECLARE CURSOR” on page 576.

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.

isolation-clause

Specifies the isolation level to be used for this statement.

WITH

Introduces the isolation level, which may be one of:

- RR Repeatable read
- RS Read stability
- CS Cursor stability
- UR Uncommitted read
- NC No commit

If *isolation-clause* is not specified the default isolation is used. See “isolation-clause” on page 357 for a description of how the default is determined.

DELETE Rules

Triggers: If the identified table or the base table of the identified view has a delete trigger, the trigger is activated. A trigger might cause other statements to be executed or raise error conditions based on the deleted values.

Referential Integrity: If the identified table or the base table of the identified table is a parent table, the rows selected must not have any dependents in a relationship with a delete rule of RESTRICT or NO ACTION, and the DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT or NO ACTION.

If the delete operation is not prevented by a RESTRICT or NO ACTION delete rule, the selected rows are deleted. 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.
- The columns of the foreign keys of any rows that are their dependents in a relationship with a delete rule of SET DEFAULT are set to the corresponding default 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.

The referential constraints (other than a referential constraint with a RESTRICT delete rule), are effectively checked at the end of the statement. In the case of a multiple-row delete, this would occur after all rows were deleted and any associated triggers were activated.

Check Constraints: A check constraint can prevent the deletion of a row in a parent table when there are dependents in a relationship with a delete rule of SET NULL or SET DEFAULT. If deleting a row in the parent table would cause a column in a dependent table to be set to null or a default value and the null or default value would cause a search condition of a check constraint to evaluate to false, the row is not deleted.

Notes

Delete operation errors: If an error occurs while executing any delete operation, changes from this statement, referential constraints, and any triggered SQL statements are rolled back (unless the isolation level is NC for this statement or any other triggered SQL statements).

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful DELETE statement. 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 or NC

The locks can prevent other application processes from performing operations on the table. For further information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements, and "Isolation Level" on page 21.

If an application process deletes a row on which any of its non-updateable cursors are positioned, those cursors are positioned before the next row of their result

DELETE

table. Let C be a cursor that is positioned before the next row R (as the 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.

A maximum of 4000000 rows can be deleted or changed in any single DELETE statement when COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) was specified. The number of rows changed includes any rows inserted, updated, or deleted under the same commitment definition as a result of a trigger, a CASCADE, SET NULL, or SET DEFAULT referential integrity delete rule.

Number of rows deleted: When a DELETE statement is completed, the number of rows deleted is returned in SQLERRD(3) in the SQLCA. The value in SQLERRD(3) does not include the number of rows that were deleted as a result of a CASCADE delete rule or a trigger.

For a description of the SQLCA, see Appendix B, "SQL Communication Area" on page 825.

Referential integrity considerations: SQLERRD(5) in the SQLCA shows the number of rows affected by referential constraints. It includes rows that were deleted as the result of a CASCADE delete rule and rows in which foreign keys were set to NULL or the default value as the result of a SET NULL or SET DEFAULT delete rule.

For a description of the SQLCA, see Appendix B, "SQL Communication Area" on page 825.

REXX: Host variables cannot be used in the DELETE statement within a REXX procedure. Instead, the DELETE must be the object of a PREPARE and EXECUTE using parameter markers.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword NONE can be used as a synonym for NC.
- The keyword CHG can be used as a synonym for UR.
- The keyword ALL can be used as a synonym for RS.

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
```

Example 3: Use a Java program statement to delete all the subprojects (MAJPROJ is NULL) from the PROJECT table on the connection context 'ctx', for a department (DEPTNO) equal to that in the host variable HOSTDEPT (java.lang.String).

```
#sql [ctx] { DELETE FROM PROJECT
              WHERE DEPTNO = :HOSTDEPT AND MAJPROJ IS NULL };
```

Example 4: Code a portion of a Java program that will be used to display retired employees (JOB) and then, if requested to do so, remove certain employees from the EMPLOYEE table on the connection context 'ctx'.

```
#sql iterator empIterator implements sqlj.runtime.ForUpdate
( ... );
empIterator C1;

#sql [ctx] C1 = { SELECT * FROM EMPLOYEE
                 WHERE JOB = 'RETIRED' };

#sql { FETCH C1 INTO ... };
while ( !C1.endFetch() ) {
    System.out.println( ... );
    ...
    if ( condition for deleting row ) {
        #sql [ctx] { DELETE FROM EMPLOYEE
                    WHERE CURRENT OF C1 };
    }
    #sql { FETCH C1 INTO ... };
}
C1.close();
```

DESCRIBE

The DESCRIBE statement obtains information about a prepared statement. For an explanation of prepared statements, see “PREPARE” on page 691.

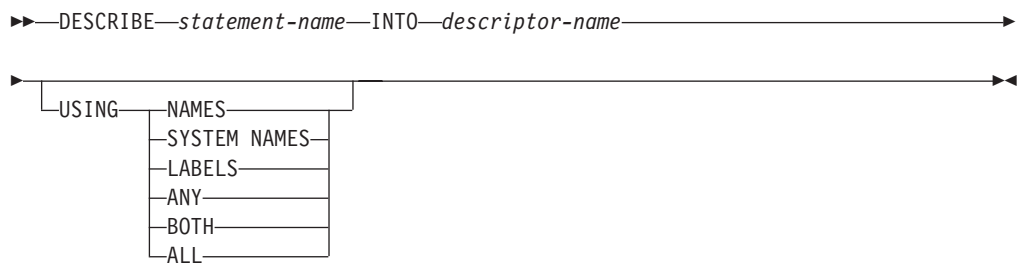
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required. See “PREPARE” on page 691 for the authorization required to create a prepared statement.

Syntax



Description

statement-name

Identifies the prepared statement. When the DESCRIBE statement is executed, the name must identify a prepared statement at the server.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, “SQL Descriptor Area (SQLDA)” on page 847. Before the DESCRIBE statement is executed, the following variable in the SQLDA must be set.

SQLN Indicates the number of SQLVAR entries provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the DESCRIBE statement is executed. For information on techniques to determine the number of occurrences requires, see “Determining How Many SQLVAR Occurrences are Needed” on page 850.

The rules for REXX are different. For more information, see the SQL Programming with Host Languages book.

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 is set based on the result columns described:

- If the SQLDA contains two, three, or four SQLVAR entries for every select list item (or, column of the result table), the seventh

byte is set to '2', '3', or '4'. This technique is used in order to accommodate LOB or distinct type result columns, labels, and system names.

- Otherwise, the seventh byte is set to the space character.

The seventh byte is set to the space character if there is not enough room in the SQLDA to contain the description of all result columns.

The eighth byte is set to the space character.

SQLDABC	Length of the SQLDA in bytes.
SQLD	If the prepared statement is a SELECT, the number of columns in its result table; otherwise, 0.
SQLVAR	If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR. If the value of SQLD is n , where n is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first n 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. For information on the values assigned to SQLVAR occurrences, see "Field Descriptions in an Occurrence of SQLVAR" on page 852.

USING

Specifies what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to a length of 0.

NAMES

Assigns the name of the column. This is the default. For the DESCRIBE of a prepared statement where the name is explicitly listed in the select-list, the name specified is returned.

SYSTEM NAMES

Assigns the system column name of the column.

LABELS

Assigns the label of the column. (Column labels are defined by the LABEL statement.) Only the first 20 bytes of the label are returned.

ANY

Assigns the column label. If the column has no label, the column name is used instead.

BOTH

Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $2*n$ or $3*n$ (where n is the number of columns in the table or view). The first n occurrences of SQLVAR contain the column names. Either the second or third n occurrences contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

ALL

Assigns the label, column name, and system column name. In this case

DESCRIBE

three or four occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $3*n$ or $4*n$ (where n is the number of columns in the result table). The first n occurrences of SQLVAR contain the system column names. The second or third n occurrences contain the column labels. The third or fourth n occurrences contain the column names. If there are no distinct types, the labels are returned in the second set of SQLVAR entries and the column names are returned in the third set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries and the column names are returned in the fourth set of SQLVAR entries.

Notes

PREPARE INTO

Information about a prepared statement can also be obtained by using the INTO clause of the PREPARE statement.

Allocating the SQLDA

In C, COBOL, PL/I, and RPG, before the DESCRIBE or PREPARE INTO statement is executed, enough storage must be allocated for some number of SQLVAR occurrences. SQLN must then be set to the number of SQLVAR occurrences that were allocated. To obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR entries must not be less than the number of columns. Furthermore, if the columns include LOBs or distinct types, the number of occurrences of SQLVAR entries should be two times the number of columns. See "Determining How Many SQLVAR Occurrences are Needed" on page 850 for more information.

Among the possible ways to allocate the SQLDA are the three described below:

First technique

Allocate an SQLDA with enough occurrences of SQLVAR entries to accommodate any select list that the application will have to process. At the extreme, the number of SQLVARs could equal two times the maximum number of columns allowed in a result table. 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 three steps for every processed select list:

1. Execute a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR entries; 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 entries or half the required number. Because there were no SQLVAR entries, a warning will be issued.
2. If the SQLSTATE accompanying that warning is equal to 01005, allocate an SQLDA with $2 * \text{SQLD}$ occurrences and set SQLN in the new SQLDA to $2 * \text{SQLD}$. Otherwise, allocate an SQLDA with SQLD occurrences and set SQLN in the new SQLDA to the value of SQLD.
3. 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. If an execution of DESCRIBE fails because the SQLDA is too small, allocate a larger SQLDA and execute DESCRIBE again. For the new SQLDA, use the value of SQLD (or double the value of SQLD) returned from the first execution of DESCRIBE for the number of occurrences of SQLVAR entries.

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 entries. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR entries 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;
struct sqlda mja;
struct sqlda *mjap;

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 :mja;

if (mja.sqld == 0);
else
{
... /* Code to re-allocate the SQLDA and set mjap */
.
.
.
if (strcmp(SQLSTATE,"01005") == 0)
{
mjap->sqln = 2*mja.sqld;
SETSQLDOUBLED(mjap, SQLDOUBLED);
}
else
{
mjap->sqln = mja.sqld;
SETSQLDOUBLED(mjap, SQLSINGLED);
}
EXEC SQL DESCRIBE STMT1_NAME INTO :newda;
}

... /* code to prepare for the use of the SQLDA */
EXEC SQL OPEN DYN_CURSOR;

... /* loop to fetch rows from result table */
```

DESCRIBE

```
EXEC SQL  FETCH DYN_CURSOR USING DESCRIPTOR :mja;  
.  
.  
.
```

DESCRIBE TABLE

The DESCRIBE TABLE statement obtains information about a table or view.

Invocation

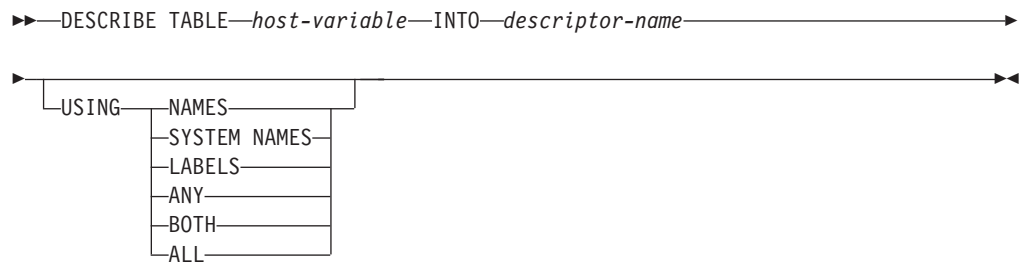
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table or view identified in the statement:
 - The system authority of *OBJOPR on the table or view
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

Syntax



Description

host-variable

Identifies the table or view to describe. When the DESCRIBE TABLE statement is executed:

- The name must identify a table or view that exists at the server.
- The host-variable must be a character-string or UCS-2 graphic-string variable and must not include an indicator variable.
- The table name that is contained within the host-variable must be left-justified and must be padded on the right with blanks if its length is less than that of the host-variable.
- The name of the table must be in uppercase unless it is a delimited name.

When the DESCRIBE TABLE statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, “SQL Descriptor Area (SQLDA)” on page 847. Before the DESCRIBE TABLE statement is executed, the following variable in the SQLDA must be set.

SQLN Specifies the number of SQLVAR occurrences provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the

DESCRIBE TABLE

DESCRIBE TABLE statement is executed. For information on techniques to determine the number of occurrences requires, see "Determining How Many SQLVAR Occurrences are Needed" on page 850.

The rules for REXX are different. For more information, see the SQL Programming with Host Languages book.

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 is set based on the result columns described:

- If the SQLDA contains two, three, or four SQLVAR entries for every select list item (or, column of the result table), the seventh byte is set to '2', '3', or '4'. This technique is used in order to accommodate LOB or distinct type result columns, lables, and system names.
- Otherwise, the seventh byte is set to the space character.

The seventh byte is set to the space character if there is not enough room in the SQLDA to contain the description of all result columns.

The eighth byte is set to the space character.

SQLDABC Length of the SQLDA in bytes.

SQLD If the prepared statement is a SELECT, the number of columns in its result table; otherwise, 0.

SQLVAR If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is n , where n is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first n 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. For information on the values assigned to SQLVAR occurrences, see "Field Descriptions in an Occurrence of SQLVAR" on page 852.

USING

Specifies what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to a length of 0.

NAMES

Assigns the name of the column. This is the default.

SYSTEM NAMES

Assigns the system column name of the column.

LABELS

Assigns the label of the column. (Column labels are defined by the LABEL statement.) Only the first 20 bytes of the label are returned.

ANY

Assigns the column label. If the column has no label, the column name is used instead.

BOTH

Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $2*n$ or $3*n$ (where n is the number of columns in the table or view). The first n occurrences of SQLVAR contain the column names. Either the second or third n occurrences contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

ALL

Assigns the label, column name, and system column name. In this case three or four occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $3*n$ or $4*n$ (where n is the number of columns in the result table). The first n occurrences of SQLVAR contain the system column names. The second or third n occurrences contain the column labels. The third or fourth n occurrences contain the column names. If there are no distinct types, the labels are returned in the second set of SQLVAR entries and the column names are returned in the third set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries and the column names are returned in the fourth set of SQLVAR entries.

Notes

Allocating the SQLDA: Before the DESCRIBE TABLE statement is executed, the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. To obtain the description of the columns of the table or view, the number of occurrences of SQLVAR must not be less than the number of columns. Furthermore, if USING BOTH or USING ALL is specified, or if the columns include LOBs or distinct types, the number of occurrences of SQLVAR should be two, three, or four times the number of columns. See “Determining How Many SQLVAR Occurrences are Needed” on page 850 for more information.

If not enough occurrences are provided to return all sets of occurrences, SQLN is set to the total number of occurrences necessary to return all information. Otherwise, SQLN is set to the number of columns.

For a description of techniques that can be used to allocate the SQLDA, see Appendix D, “SQL Descriptor Area (SQLDA)” on page 847.

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 table_name[201];
EXEC SQL END DECLARE SECTION;
```

DESCRIBE TABLE

```
| EXEC SQL INCLUDE SQLDA;  
| EXEC SQL DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;  
|  
| .../*code to prompt user for a table or view */  
| .../*code to set SQLN to zero and to allocate the SQLDA */  
| EXEC SQL DESCRIBE TABLE :table_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 TABLE :table_name INTO :sqlda;  
|  
| .  
| .  
| .
```

DISCONNECT

The DISCONNECT statement ends one or more connections for unprotected conversations.

Invocation

This statement can only be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

DISCONNECT is not allowed in a trigger. DISCONNECT is not allowed in an external procedure if the external procedure is called on a remote server.

Authorization

None required.

Syntax



Description

server-name or *host-variable*

Identifies the server by the specified server name or the server name contained in the host variable. If a host variable is specified:

- It must be a character-string variable.
- It must not be followed by an indicator variable
- The server name must be left-justified within the host variable and must conform to the rules for forming an ordinary identifier
- If the length of the server name is less than the length of the host variable, it must be padded on the right with blanks.

When the DISCONNECT statement is executed, the specified server name or server name contained in the host variable must identify an existing dormant or current connection of the activation group. The identified connection cannot use a protected conversation.

CURRENT

Identifies the current connection of the activation group. The activation group must be in the connected state. The current connection must not use a protected conversation.

ALL or ALL SQL

Identifies all existing connections of the activation group (local as well as remote connections). An error or warning does not occur if no connections exist when the statement is executed. None of the connections can use protected conversations.

DISCONNECT

Notes

An identified connection must not be a connection that was used to execute SQL statements during the current unit of work and must not be a connection for a protected conversation. To end connections on protected conversations, use the `RELEASE` statement. Local connections are never considered to be protected conversations.

If the `DISCONNECT` statement is successful, each identified connection is ended. If the current connection is destroyed, the activation group is placed in the unconnected state.

If the `DISCONNECT` statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Using `CONNECT` (Type 1) semantics does not prevent using `DISCONNECT`.

`DISCONNECT` closes cursors, releases resources, and prevents further use of the connection.

`ROLLBACK` does not reconnect a connection that has been ended by `DISCONNECT`.

Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be ended as soon as possible and a remote connection that is going to be reused should not be destroyed.

The `DISCONNECT` statement should be executed immediately after a commit operation. If `DISCONNECT` is used to end the current connection, the next executed SQL statement must be `CONNECT` or `SET CONNECTION`.

`DISCONNECT ALL` ends the connection to the local server. A connection is ended even though it has an open cursor defined with the `WITH HOLD` clause.

Examples

Example 1: The connection to `TOROLAB1` is no longer needed. The following statement is executed after a commit operation.

```
EXEC SQL DISCONNECT TOROLAB1;
```

Example 2: The current connection is no longer needed. The following statement is executed after a commit operation.

```
EXEC SQL DISCONNECT CURRENT;
```

Example 3: The existing connections are no longer needed. The following statement is executed after a commit operation.

```
EXEC SQL DISCONNECT ALL;
```

DROP

The DROP statement drops an object. Objects that are directly or indirectly dependent on that object may also be dropped.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.


Authorization

To drop a table, view, index, alias or package, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - The system authorities of *OBJOPR and *OBJEXIST on the object to be dropped
 - If the object is a table or view, the system authorities of *OBJOPR and *OBJEXIST on any views, indexes, and logical files that are dependent on that table or view
 - The system authority *EXECUTE on the library that contains the object to be dropped
- Administrative authority

To drop a schema, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - The system authorities of *OBJEXIST, *OBJOPR, *EXECUTE, and *READ on the library to be dropped.
 - The system authorities of *OBJOPR and *OBJEXIST on all objects in the schema and *OBJOPR and *OBJEXIST on any views, indexes and logical files that are dependent on tables and views in the schema.
 - Any additional authorities required to delete other object types that exist in the schema. For example, *OBJMGT to the data dictionary if the schema contains a data dictionary, and some system data authority to the journal

receiver. For more information, see the iSeries Security Reference  book.

- Administrative authority

To drop a distinct type, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - The system authorities of *OBJOPR and *OBJEXIST on the distinct type to be dropped
 - The DELETE privilege on the SYSTYPES, SYSPARMS, and SYSROUTINES catalog tables, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

To drop a function, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:

DROP

- For SQL functions, the system authority *OBJEXIST on the program object associated with the function, and
- The DELETE privilege on the SYSFUNCS and SYSPARMS catalog tables, and
- The system authority *EXECUTE on library QSYS2
- Administrative authority

To drop a procedure, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - For SQL procedures, the system authority *OBJEXIST on the program object associated with the procedure, and
 - The DELETE privilege on the SYSPROCS and SYSPARMS catalog tables, and
 - The system authority *EXECUTE on library QSYS2
- Administrative authority

The authorization ID of the statement has the DELETE privilege on a table when:

- It is the owner of the table,
- It has been granted the DELETE privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *DLT on the table.

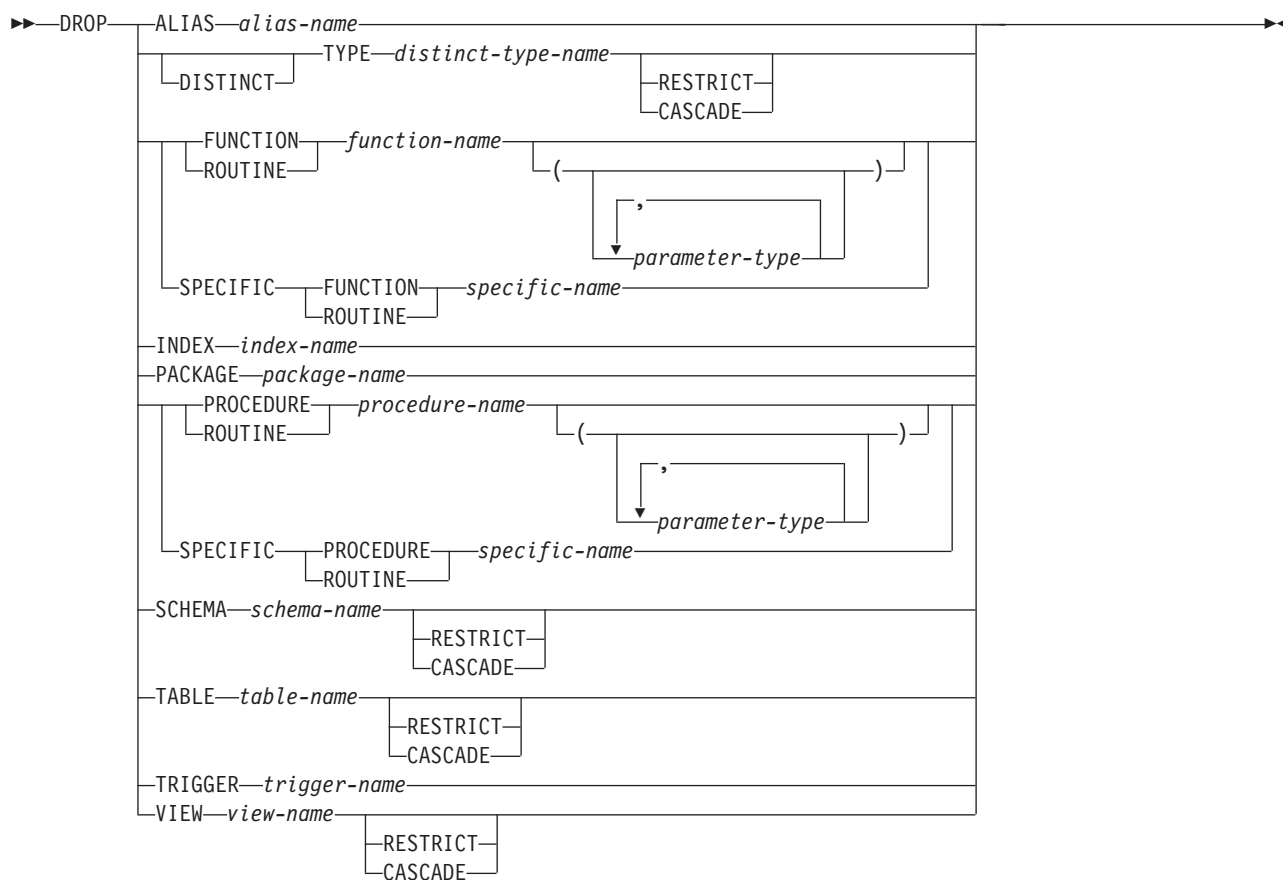
To drop a trigger, the privileges held by the authorization ID of the statement must include at least one of the following:

- The following privileges:
 - The system authority *USE to the Remove Physical File Trigger (RMVPFTRG) command, and
 - For the subject table of the trigger:
 - The ALTER privilege to the subject table, and
 - The system authority *EXECUTE on the library containing the subject table,
 - If the trigger being dropped is an SQL trigger:
 - The system authority *OBJEXIST on the trigger program object, and
 - The system authority *EXECUTE on the library containing the trigger.
- Administrative authority

The authorization ID of the statement has the ALTER privilege on the table when one of the following is true:

- It is the owner of the table.
- It was granted the ALTER privilege to the table.
- It was granted the system authorities of either *OBJALTER or *OBJMGT to the table.

Syntax



parameter-type:

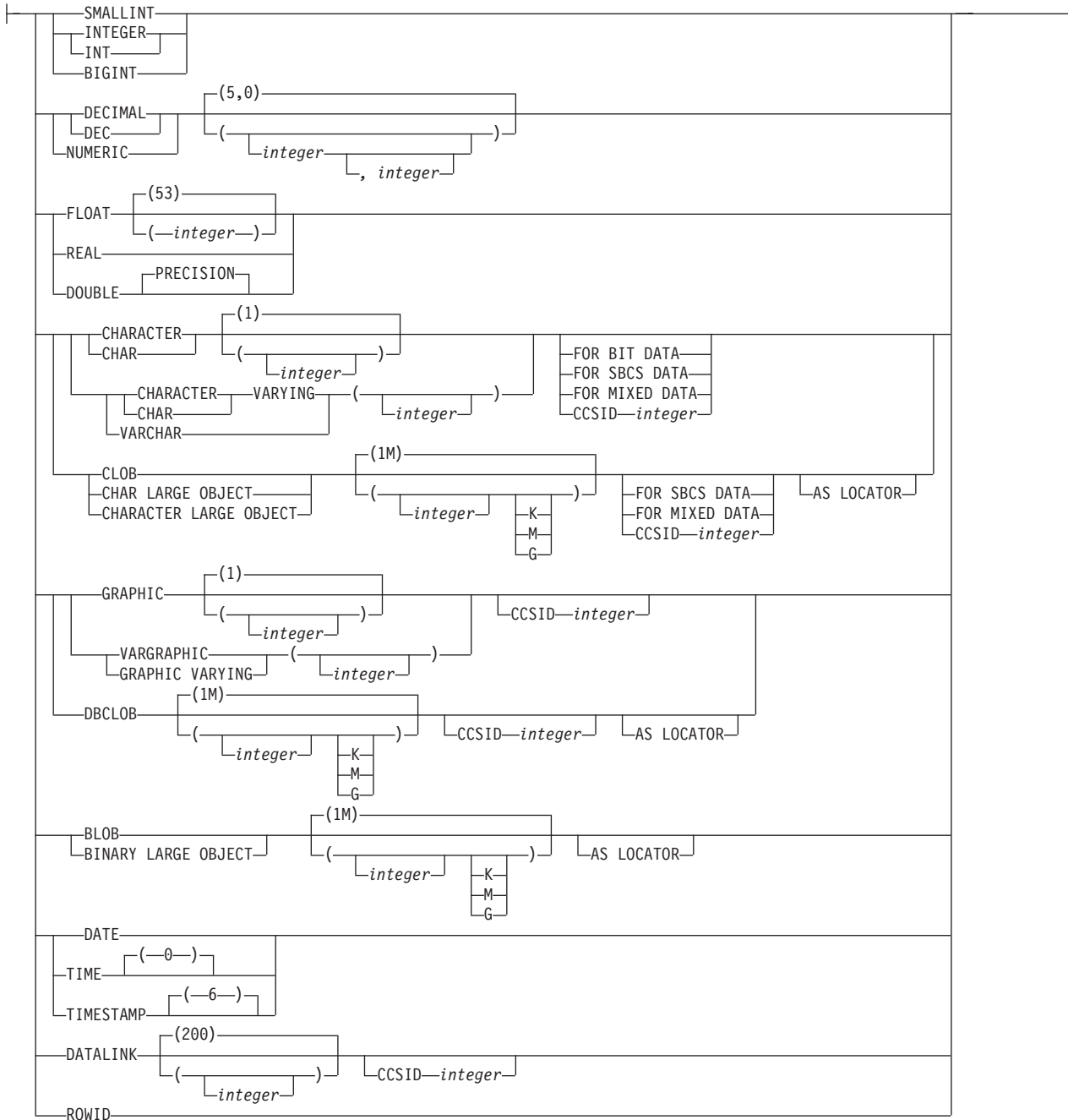


data-type:



DROP

built-in-type:



Description

ALIAS *alias-name*

Identifies the alias that is to be dropped. The *alias-name* must identify an alias that exists at the current server.

The specified alias is deleted from the schema. Dropping an alias has no effect on any constraint or view that was defined using the alias. An alias can be dropped whether or not it is referenced in a function, package, procedure, program, or trigger.

DISTINCT TYPE *distinct-type-name*

Identifies the distinct type that is to be dropped. The *distinct-type-name* must identify a distinct type that exists at the current server. The specified type is deleted from the schema.

Neither CASCADE nor RESTRICT

Specifies that the type cannot be dropped if any constraints, indexes, tables, and views reference the type.

For every procedure or function R that has parameters or a return value of the type being dropped, or a reference to the type being dropped, the following DROP statement is effectively executed:

DROP ROUTINE R

For every trigger T that references the type being dropped, the following DROP statement is effectively executed:

DROP TRIGGER T

It is possible that this statement would cascade to drop dependent functions or procedures. If all of these functions or procedures are in the list to be dropped because of a dependency on the distinct type, the drop of the distinct type will succeed.

CASCADE

Specifies that the type will be dropped even if it is referenced in a constraint, function, index, procedure, table, trigger, or view. All constraints, functions, indexes, procedures, tables, triggers, and views that reference the type are dropped.

RESTRICT

Specifies that the type cannot be dropped if it is referenced in a constraint, function (other than a function that was created when the type was created), index, procedure, table, trigger, or view.

FUNCTION or SPECIFIC FUNCTION

Identifies the function that is to be dropped. The function must exist at the current server and it must be a function that was defined with the CREATE FUNCTION statement. The particular function can be identified by its name, function signature, or specific name.

Functions implicitly generated by the CREATE DISTINCT TYPE statement cannot be dropped using the DROP statement. They are implicitly dropped when the distinct type is dropped.

The function cannot be dropped if another function is dependent on it. A function is dependent on another function if it was identified in the SOURCE clause of the CREATE FUNCTION statement. A function can be dropped whether or not it is referenced in a function, package, procedure, program, trigger, or view.

The specified function is dropped from the schema. All privileges on the user-defined function are also dropped. If this is an SQL function or sourced function, the service program (*SRVPGM) associated with the function is also

DROP

dropped. If this is an external function, the information that was saved in the program or service program specified on the CREATE FUNCTION statement is removed from the object.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is to be dropped. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type, ...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional.

Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX *index-name*

Identifies the index that is to be dropped. The *index-name* must identify an index that exists at the current server.

The specified index is dropped from the schema. An index can be dropped whether or not it is referenced in a function, package, procedure, program, or trigger.

PACKAGE *package-name*

Identifies the package that is to be dropped. The *package-name* must identify a package that exists at the current server.

The specified package is dropped from the schema. All privileges on the package are also dropped.

A package can be dropped whether or not it is referenced in a function, package, procedure, program, or trigger.

PROCEDURE or SPECIFIC PROCEDURE

Identifies the procedure that is to be dropped. The *procedure-name* must identify a procedure that exists at the current server.

The specified procedure is dropped from the schema. All privileges on the procedure are also dropped. If this is an SQL procedure, the program (*PGM) associated with the procedure is also dropped. If this is an external procedure, the information that was saved in the program specified on the CREATE PROCEDURE statement is removed from the object.

A procedure can be dropped whether or not it is referenced in a function, package, procedure, program, trigger, or view.

PROCEDURE *procedure-name*

Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*

Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. 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. Synonyms for data types are considered a match.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name

Identifies the name of the procedure.

DROP

(parameter-type, ...)

Identifies the parameters of the procedure.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional.

Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR

Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC PROCEDURE *specific-name*

Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

SCHEMA *schema-name*

Identifies the schema that is to be dropped. The *schema-name* must identify a schema that exists at the current server.

The specified schema is dropped. Each object in the schema is dropped as if the appropriate DROP statement was executed with the specified drop option (CASCADE, RESTRICT, or neither). See the DROP description of these object types for information on the handling of objects dependent on these objects.

DROP SCHEMA is only valid when the commit level is *NONE.

Neither CASCADE nor RESTRICT

Specifies that the schema will be dropped even if it is referenced in a function, package, procedure, program, table, or trigger in another schema.

CASCADE

Specifies that any triggers that reference the schema will be dropped.

RESTRICT

Specifies that the schema cannot be dropped if it is referenced in an SQL trigger in another schema.

TABLE *table-name*

Identifies the table that is to be dropped. The *table-name* must identify a base table that exists at the current server, but must not identify a catalog table.

The specified table is dropped from the schema. All privileges, constraints, and triggers on the table are also dropped.

Any aliases that reference the specified table are not dropped.

Neither CASCADE nor RESTRICT

Specifies that the table will be dropped even if it is referenced in a constraint, index, trigger, or view. All indexes and views that reference the table are dropped.

CASCADE

Specifies that the table will be dropped even if it is referenced in a constraint, index, trigger, or view. All constraints, indexes, triggers, and views that reference the table are dropped.

RESTRICT

Specifies that the table cannot be dropped if it is referenced in a constraint, index, trigger, or view.

TRIGGER *trigger-name*

Identifies the trigger that is to be dropped. The *trigger-name* must identify a trigger that exists at the current server.

The specified trigger is dropped from the schema. If the trigger is an SQL trigger, the program object associated with the trigger is also deleted from the schema.

VIEW *view-name*

Identifies the view that is to be dropped. The *view-name* must identify a view that exists at the current server, but must not identify a catalog view.

The specified view is dropped from the schema. When a view is dropped, all privileges on that view are dropped.

Neither CASCADE nor RESTRICT

Specifies that the view will be dropped even if it is referenced in a trigger or another view. All views that reference the view are dropped.

CASCADE

Specifies that the view will be dropped even if it is referenced in a trigger or another view. All triggers and views that reference the view are dropped.

RESTRICT

Specifies that the view cannot be dropped if it is referenced in a trigger or another view.

Note

Drop effects: Whenever an object is dropped, its description is dropped from the catalog. If the object is referenced in a function, package, procedure, program, or trigger; any access plans that reference the object are implicitly prepared again when the access plan is next used. If the object does not exist at that time, an error is returned.

DROP

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword SYNONYM can be used as a synonym for ALIAS.
- The keyword DATA can be used as a synonym for DISTINCT.
- The keyword PROGRAM can be used as a synonym for PACKAGE.
- The keyword COLLECTION can be used as a synonym for SCHEMA.

Examples

Example 1: Drop your table named MY_IN_TRAY. Do not allow the drop if any views or indexes are created over this table.

```
DROP TABLE MY_IN_TRAY RESTRICT
```

Example 2: Drop your view named MA_PROJ.

```
DROP VIEW MA_PROJ
```

Example 3: Drop the package named PERS.PACKA.

```
DROP PACKAGE PERS.PACKA
```

Example 4: Drop the distinct type DOCUMENT, if it is not currently in use:

```
DROP DISTINCT TYPE DOCUMENT RESTRICT
```

Example 5: Assume that you are SMITH and that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Drop ATOMIC_WEIGHT.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT RESTRICT
```

Example 6: Drop CENTER, using the function signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTER (INTEGER, FLOAT) RESTRICT
```

Example 7: Assume that you are SMITH and that you created another function named CENTER, which you gave the specific name FOCUS97, in schema JOHNSON. Drop CENTER, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION JOHNSON.FOCUS97
```

Example 8: Assume that you are SMITH and that stored procedure OSMOSIS is in schema BIOLOGY. Drop OSMOSIS.

```
DROP PROCEDURE BIOLOGY.OSMOSIS
```

Example 9: Assume that you are SMITH and that trigger BONUS is in your schema. Drop BONUS.

```
DROP TRIGGER BONUS
```

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of an SQL declare section.

Invocation

| This statement can only be embedded in an application program. It is not an
| executable statement. It must not be specified in RPG, Java, or REXX.

Authorization

None required.

Syntax

►►—END DECLARE SECTION—◄◄

Description

The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear in accordance with the rules of the host language. It is used to indicate the end of an SQL declare section. An SQL declare section starts with a BEGIN DECLARE SECTION statement. For more information about the BEGIN DECLARE SECTION statement, see “BEGIN DECLARE SECTION” on page 394.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and cannot be nested.

Examples

See “BEGIN DECLARE SECTION” on page 394 for examples using the END DECLARE SECTION statement.

EXECUTE

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

Invocation

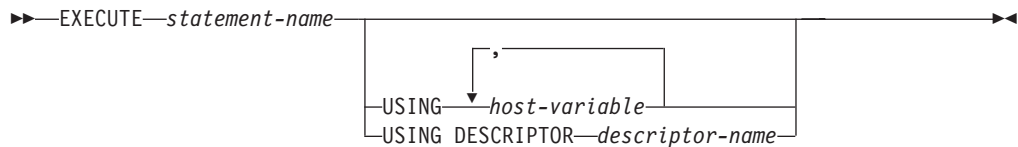
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The authorization rules are those defined for the SQL statement specified by EXECUTE. For example, see the description of INSERT for the authorization rules that apply when an INSERT statement is executed using EXECUTE.

The authorization ID of the statement is the run-time authorization ID unless DYNUSRPRF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see "Authorization IDs and Authorization-Names" on page 57.

Syntax



Description

statement-name

Identifies the prepared statement to be executed. *Statement-name* must identify a statement that was previously prepared. The prepared statement cannot be a SELECT statement.

USING

Introduces a list of host variables whose values are substituted for the parameter markers (question marks) in the prepared statement. For an explanation of parameter markers, see "PREPARE" on page 691. If the prepared statement includes parameter markers, the USING clause must be used. USING is ignored if there are no parameter markers.

host-variable,...

Identifies one of more host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. A reference to a host structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement.

DESCRIPTOR *descriptor-name*

Identifies an SQLDA that must contain a valid description of host variables.

Before the EXECUTE statement is processed, the user must set the following fields in the SQLDA. (The rules for REXX are different. For more information, see the SQL Programming with Host Languages book.)

- 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. If LOBs or distinct types are present in the results, there must be additional SQLVAR entries for each parameter. For more information about the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see Appendix D, “SQL Descriptor Area (SQLDA)” on page 847.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement.

Note that RPG/400 does not provide the function for setting pointers. Because the SQLDA uses pointers to locate the appropriate host variables, you have to set these pointers outside your RPG/400 application.

Notes

Parameter Marker Replacement: Before the prepared statement is executed, each parameter marker in the statement is effectively replaced by its corresponding host variable. The replacement of a parameter marker is an assignment operation in which the source is the value of the host variable, and the target is a variable within the database manager. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Table 58 on page 694.

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 number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of *V* are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, the value of *V* must not be null.

However, unlike the rules for assigning a value to a column:

- If *V* is a string, the value will be truncated (without an error), if its length is greater than the length attribute of the target.

EXECUTE

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.

Example

This example of portions of a COBOL program shows how an INSERT statement with parameter markers is prepared and executed.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  77 EMP          PIC X(6).
  77 PRJ          PIC X(6).
  77 ACT          PIC S9(4) COMP-4.
  77 TIM          PIC S9(3)V9(2).
  01 HOLDER.
    49 HOLDER-LENGTH PIC S9(4) COMP-4.
    49 HOLDER-VALUE  PIC X(80).
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
MOVE 70 TO HOLDER-LENGTH.
MOVE "INSERT INTO EMPPROJACT (EMPNO, PROJNO, ACTNO, EMPTIME)
-  "VALUES (?, ?, ?, ?)" TO HOLDER-VALUE.
EXEC SQL PREPARE MYINSERT FROM :HOLDER END-EXEC.

IF SQLCODE = 0
  PERFORM DO-INSERT THRU END-DO-INSERT
ELSE
  PERFORM ERROR-CONDITION.

DO-INSERT.
  MOVE "000010" TO EMP.
  MOVE "AD3100" TO PRJ.
  MOVE 160      TO ACT.
  MOVE .50     TO TIM.
  EXEC SQL EXECUTE MYINSERT USING :EMP, :PRJ, :ACT, :TIM END-EXEC.
END-DO-INSERT.
.
.
.
```

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 only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The authorization rules are those defined for the SQL statement specified by EXECUTE IMMEDIATE. For example, see “INSERT” on page 673 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

The authorization ID of the statement is the run-time authorization ID unless DYNUSRPRF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see “Authorization IDs and Authorization-Names” on page 57.

Syntax

```

▶▶ EXECUTE IMMEDIATE host-variable
                    └── string-expression ───▶
  
```

Description

host-variable

Identifies a host variable that must be declared in accordance with the rules for declaring character-string or UCS-2 graphic host variables. The host variable must not have a CLOB or DBCLOB data type, and an indicator variable must not be specified.

string-expression

A *string-expression* is any PL/I *string-expression* that yields a character string. SQL expressions that yield a character string are not allowed. A *string-expression* is only allowed in PL/I.

The value of the identified host variable or string expression is called a *statement string*.

The statement string must be one of the following SQL statements:⁵⁷

57. A select-statement is not allowed. To dynamically process a select-statement, use the PREPARE, DECLARE CURSOR, and OPEN statements.

EXECUTE IMMEDIATE

ALTER	DROP	REVOKE
CALL	GRANT	ROLLBACK
COMMENT	INSERT	SET PATH
COMMIT	LABEL	SET TRANSACTION
CREATE	LOCK TABLE	UPDATE
DELETE	RENAME	

The statement string must not:

- Begin with EXEC SQL and end with END-EXEC or a semicolon (;).
- Include references to host variables.
- Include parameter markers.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is not valid, 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.

Note

If the same SQL statement is to be executed more than once, it is more efficient to use the PREPARE and EXECUTE statements rather than the EXECUTE IMMEDIATE statement.

Example

Use C to execute the SQL statement in the host variable Qstring.

```
void main ()
{
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.

    char Qstring[100] = "INSERT INTO WORK_TABLE SELECT * FROM EMPPROJECT
        WHERE ACTNO >= 100";

    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA;
    .
    .
    EXEC SQL EXECUTE IMMEDIATE :Qstring;

    return;
}
```


FETCH

The FETCH statement positions a cursor on a row of the result table. It can return zero, one, or multiple rows, and it assigns the values of the rows returned to host variables.

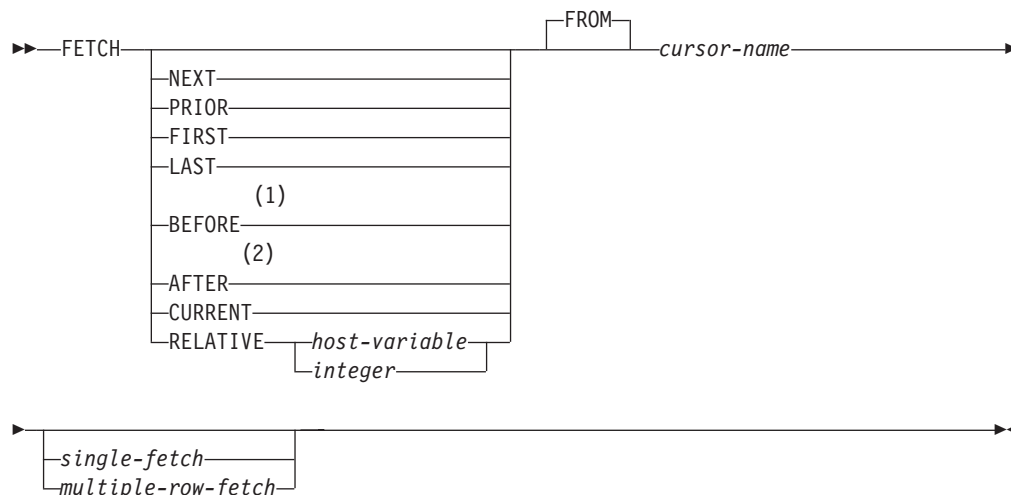
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. Multiple row fetch is not allowed in a REXX procedure.

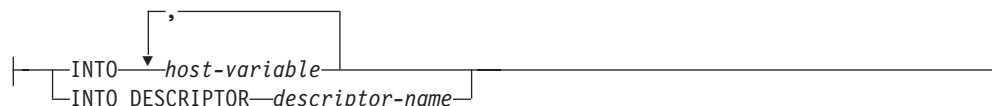
Authorization

See "DECLARE CURSOR" on page 576 for an explanation of the authorization required to use a cursor.

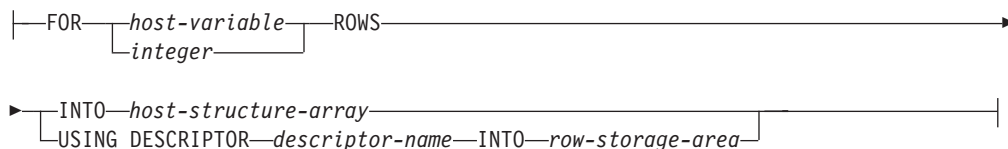
Syntax



single-fetch:



multiple-row-fetch:



row-storage-area:



Notes:

- 1 If BEFORE is specified, a single-fetch-clause or multiple-row-fetch-clause must not be specified.
- 2 If AFTER is specified, a single-fetch-clause or multiple-row-fetch-clause must not be specified.

Description

The following keywords specify a new position for the cursor: NEXT, PRIOR, FIRST, LAST, BEFORE, AFTER, CURRENT, and RELATIVE. Of those keywords, only NEXT may be used for cursors that have not been declared SCROLL.

NEXT

Positions the cursor on the next row of the result table relative to the current cursor position. NEXT is the default if no other cursor orientation is specified.

PRIOR

Positions the cursor on the previous row of the result table relative to the current cursor position.

FIRST

Positions the cursor on the first row of the result table.

LAST

Positions the cursor on the last row of the result table.

BEFORE

Positions the cursor before the first row of the result table.

AFTER

Positions the cursor after the last row of the result table.

CURRENT

Does not reposition the cursor, but maintains the current cursor position. If the cursor has been declared as DYNAMIC SCROLL and the current row has been updated so its place within the sort order of the result table is changed, an error is returned.

RELATIVE

Host-variable or *integer* is assigned to an integer value *k*. RELATIVE positions the cursor to the row in the result table that is either *k* rows after the current row if *k*>0, or *k* rows before the current row if *k*<0. If a *host-variable* is specified, it must be a numeric variable with zero scale and it must not include an indicator variable.

Table 51. Synonymous Scroll Specifications

Specification	Alternative
RELATIVE +1	NEXT
RELATIVE -1	PRIOR
RELATIVE 0	CURRENT

FROM

This keyword is provided for clarity only. If a scroll position option is specified, then this keyword is required. If no scrolling option is specified, then the FROM keyword is optional.

cursor-name

Identifies the cursor to be used in the fetch operation. The *cursor-name* must identify a declared cursor as explained in “Description” on page 577 for the DECLARE CURSOR statement. When the FETCH statement is executed, the cursor must be in the open state.

If a single- or multiple-row fetch clause is not specified, no data is returned to the user. However, the cursor is positioned and a row lock may be acquired. For more information about locking, see “Isolation Level” on page 21.

single-fetch**INTO** *host-variable,...*

Identifies one or more host structures or host variables that must be declared in accordance with the rules for declaring host structures and host variables. In the operational form of INTO, a host structure is replaced by a reference to each of its variables. The first value in the result row is assigned to the first host variable in the list, the second value to the second host variable, and so on.

INTO 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. (The rules for REXX are different. For more information see the SQL Programming with Host Languages book.)

- 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 $16 + \text{SQLN} \times (80)$, where 80 is the length of an SQLVAR occurrence. If LOBs are specified, there must be two SQLVAR entries for each parameter marker and SQLN must be set to two times the number of parameter markers.

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 D, “SQL Descriptor Area (SQLDA)” on page 847.

multiple-row-fetch**FOR** *k* **ROWS**

Evaluates *host-variable* or *integer* to an integral value *k*. If a *host-variable* is specified, it must be a numeric host variable with zero scale and it must not include an indicator variable. *k* must be in the range of 1 to 32767. The cursor is positioned on the row specified by the orientation keyword (for example,

FETCH

NEXT), and that row is fetched. Then the next $k-1$ rows are fetched (moving forward in the table), until the end of the cursor is reached. After the fetch operation, the cursor is positioned on the last row fetched.

For example, `FETCH PRIOR FROM C1 FOR 3 ROWS` causes the previous row, the current row, and the next row to be returned, in that order. The cursor is positioned on the next row. `FETCH RELATIVE -1 FROM C1 FOR 3 ROWS` returns the same result. `FETCH FIRST FROM C1 FOR :x ROWS` returns the first x rows, and leaves the cursor positioned on row number x .

When a multiple-row-fetch is successfully executed, three variables are set in the SQLCA:

- `SQLERRD(3)` shows the number of rows retrieved.
- `SQLERRD(4)` contains the length of the row retrieved.
- `SQLERRD(5)` contains +100 if the last row was fetched.⁵⁸

INTO *host-structure-array*

host-structure-array identifies an array of host structures defined in accordance with the rules for declaring host structures.

The first structure in the array corresponds to the first row, the second structure in the array corresponds to the second row, and so on. In addition, the first value in the row corresponds to the first item in the structure, the second value in the row corresponds to the second item in the structure, and so on. The number of rows to be fetched must be less than or equal to the dimension of the host structure array.

USING DESCRIPTOR *descriptor-name*

Identifies an `SQLDA` that must contain a valid description of zero or more host variables that describe the format of a row in the row-storage-area.

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 host variables.

The values of the other fields of the `SQLDA` (such as `SQLNAME`) may not be defined after the `FETCH` statement is executed and should not be used.

The `SQLDA` must have enough storage to contain all `SQLVAR` occurrences. Therefore, the value in `SQLDABC` must be greater than or equal to $16 + \text{SQLN} * (80)$, where 80 is the length of an `SQLVAR` occurrence. If `LOBs` or distinct types are specified, there must be two `SQLVAR` entries for each parameter marker and `SQLN` must be set to two times the number of parameter markers.

`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 D, "SQL Descriptor Area (`SQLDA`)" on page 847.

⁵⁸ If the number of rows returned is equal to the number of rows requested, then an end of data warning may not occur and `SQLERRD(5)` may not contain +100.

On completion of the FETCH, the SQLDATA pointer in the first SQLVAR entry addresses the returned value for the first column in the allocated storage in the first row, the SQLDATA pointer in the second SQLVAR entry addresses the returned value for the second column in the allocated storage in the first row, and so on. The SQLIND pointer in the first nullable SQLVAR entry addresses the first indicator value, the SQLIND pointer in the second nullable SQLVAR entry addresses the second indicator value, and so on. The SQLDA must be allocated on a 16-byte boundary.

INTO *row-storage-area*

host-identifier-1 specified with a host variable identifies an allocation of storage in which to return the rows. The rows are returned into the storage area in the format described by the SQLDA. *host-identifier-1* must be large enough to hold all the rows requested.

host-identifier-2 identifies the optional indicator area. It should be specified if the SQLTYPE of any SQLVAR occurrence is nullable. The indicators are returned as small integers. *host-identifier-2* must be large enough to contain an indicator for each nullable value for each row to be returned.

The *n*th host 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 host variable must be compatible with its corresponding column.

Each assignment to a variable is made according to the retrieval assignment rules described in “Retrieval Assignment” on page 83. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. Note that there is no warning if there are more variables than the number of result columns. If the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (division by zero, overflow, etc.) or a character conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the host variable is undefined. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.) If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. It is possible that some values have already been assigned to host variables and will remain assigned when the error occurs.

Multiple-row-fetch is not allowed if any of the result columns are LOBs or if the current connection is to a remote server.

Notes

An open cursor has three possible positions:

- Before a row
- On a row
- After the last row

If a cursor is positioned 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 positioned on a row as a result of a FETCH statement.

FETCH

It is possible for an error to occur that makes the state of the cursor unpredictable.

If the specified host variable is character and is not large enough to contain the result, 'W' is assigned to SQLWARN1 in the SQLCA. The actual length of the result is returned in the indicator variable associated with the host-variable, if an indicator variable is provided.

If the specified host variable is a C NUL-terminated host variable and is not large enough to contain the result and the NUL-terminator:

- If the *CNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the following occurs:
 - The result is truncated.
 - The last character is the NUL-terminator.
 - The value 'W' is assigned to SQLWARN1 in the SQLCA.
- If the *NOCNULRQD option on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) is specified, the following occurs:
 - The NUL-terminator is not returned.
 - The value 'N' is assigned to SQLWARN1 in the SQLCA.

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- USING DESCRIPTOR may be used as a synonym for INTO DESCRIPTOR in the single-fetch-clause.

Example

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

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 cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. FREE LOCATOR cannot be used with the EXECUTE IMMEDIATE statement. It must not be specified in Java.

Authorization

None required.

Syntax



Description

host-variable,...

Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables. The locator variable type must be a binary large object locator, a character large object locator, or a double-byte character large object locator.

The *host-variable* must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a CALL, FETCH, SELECT INTO, assignment statement, SET variable, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is returned.

If more than one locator variable is specified and an error occurs on one of the locators, no locators will be freed.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the column values. In a COBOL program, free the CLOB locator variables LOCRES and LOCHIST, and the BLOB locator variable LOCPIC.

```

EXEC SQL
FREE LOCATOR :LOCRES, :LOCHIST, :LOCPIC
END-EXEC.
  
```

GRANT (Distinct Type Privileges)

This form of the GRANT statement grants privileges on a distinct type.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

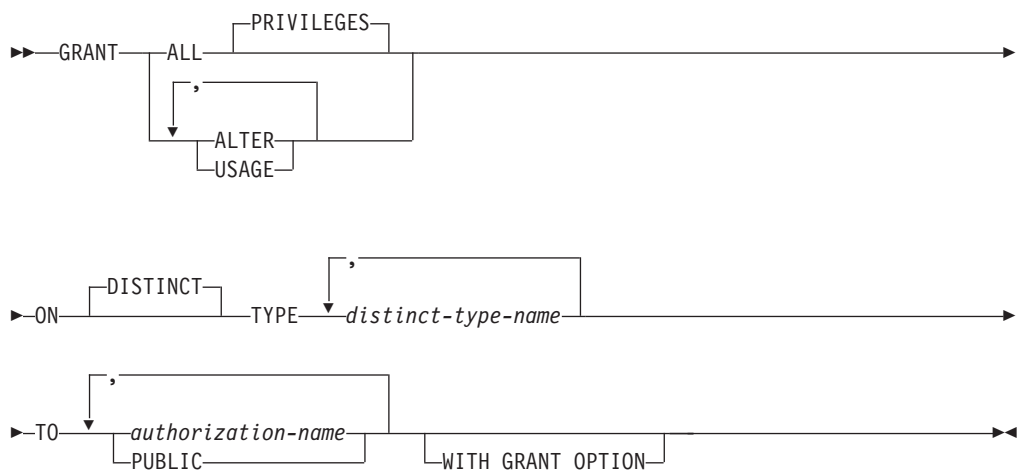
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the distinct type
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the distinct type
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified distinct types. Note that granting ALL PRIVILEGES on a distinct type is not the same as granting the system authority of *ALL.

GRANT (Distinct Type Privileges)

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER

Grants the privilege to use the COMMENT statement.

USAGE

Grants the privilege to use the distinct type in tables, functions, or procedures.

ON DISTINCT TYPE *distinct-type-name*

Identifies the distinct types on which the privilege is granted. The *distinct-type-name* must identify a distinct type that exists at the current server.

TO

Indicates to whom the privileges are granted.

authorization-name,...

Lists one or more authorization IDs.

PUBLIC

Grants the privileges to a set of users (authorization IDs). For more information, see “Authorization, Privileges and Object Ownership” on page 36.

WITH GRANT OPTION

Allows the specified *authorization-names* to grant privileges on the distinct types specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant the USAGE privilege to others unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Note

Corresponding System Authorities: GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 52. Privileges Granted to or Revoked from Distinct Types

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Distinct Type
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *OBJMGT (Revoke only)
ALTER	*OBJALTER
USAGE	*EXECUTE *OBJOPR
WITH GRANT OPTION	*OBJMGT

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword DATA can be used as a synonym for DISTINCT.

GRANT (Distinct Type Privileges)

Example

Grant the USAGE privilege on distinct type SHOE_SIZE to user JONES. This GRANT statement does not give JONES the privilege to execute the cast functions that are associated with the distinct type SHOE_SIZE.

```
GRANT USAGE
ON DISTINCT TYPE SHOE_SIZE
TO JONES
```

GRANT (Function or Procedure Privileges)

This form of the GRANT statement grants privileges on a function or procedure.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

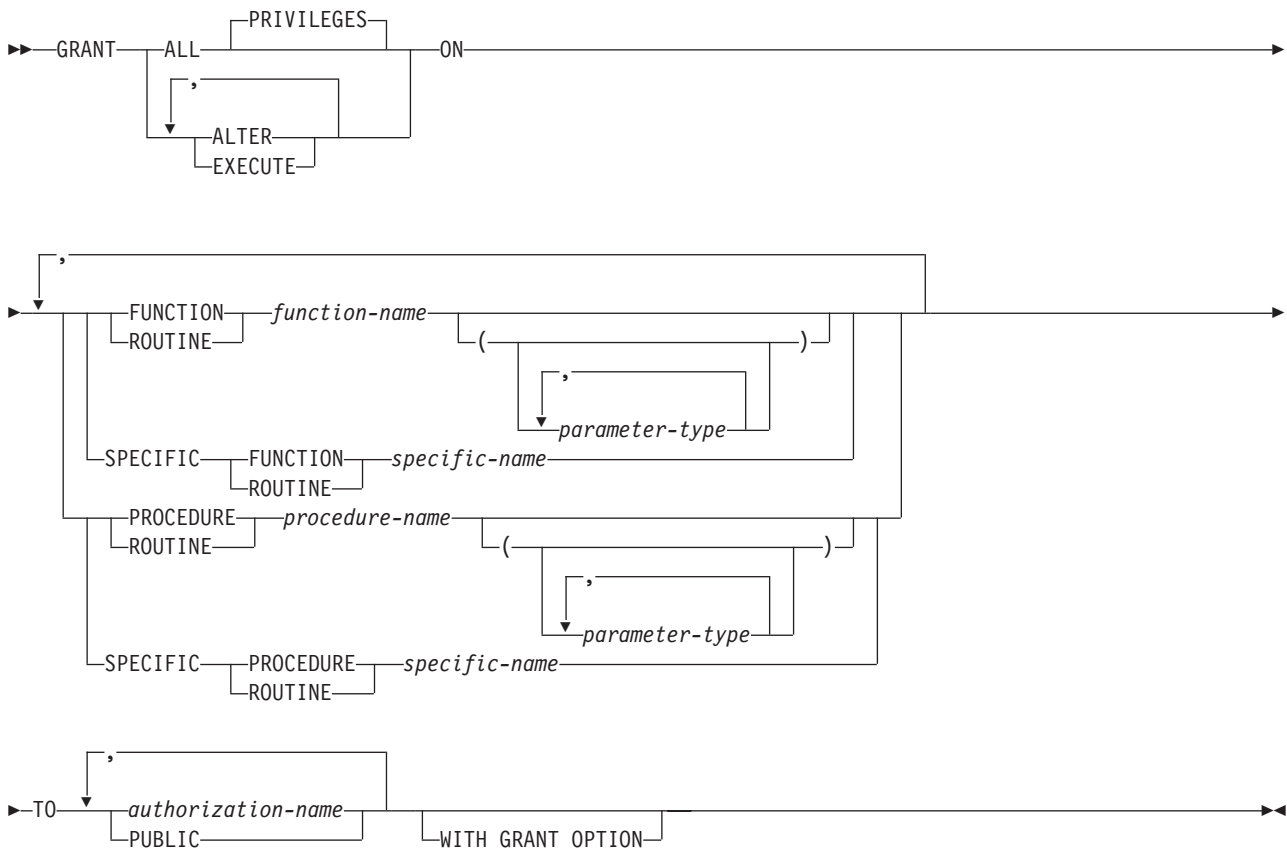
- For each function or procedure identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the function or procedure
 - The system authority *EXECUTE on the library (or directory if this is a Java routine) containing the function or procedure
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the function or procedure
- Administrative authority

Syntax

GRANT (Function or Procedure Privileges)



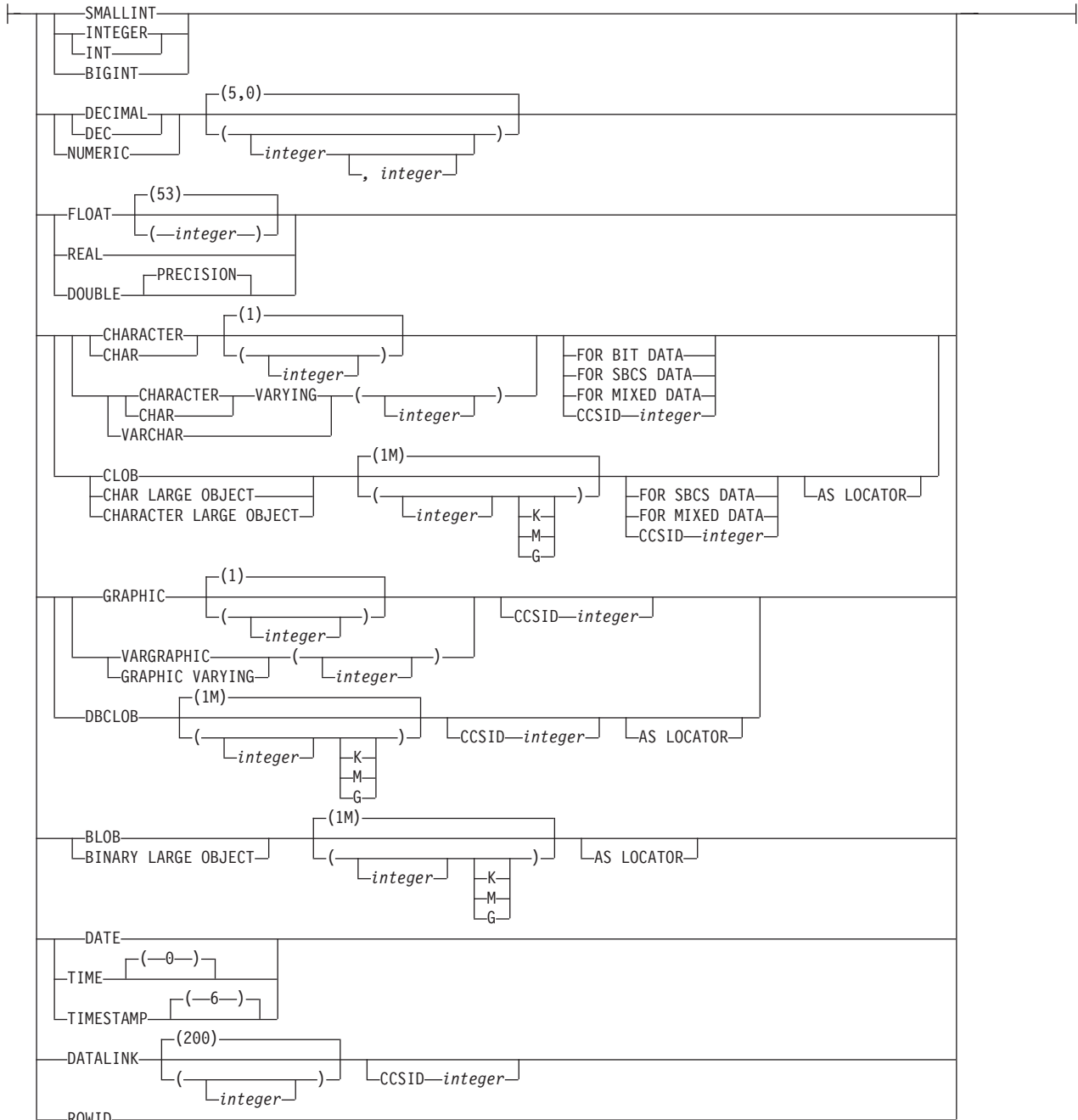
parameter-type:

| *data-type* |
 | AS LOCATOR |

data-type:

| *built-in-type* |
 | *distinct-type-name* |

built-in-type:



Description

ALL or ALL PRIVILEGES

Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified functions or procedures. Note that granting ALL PRIVILEGES on a function or procedure is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

GRANT (Function or Procedure Privileges)

ALTER

Grants the privilege to use the COMMENT statement.

EXECUTE

Grants the privilege to execute the function or procedure.

FUNCTION or SPECIFIC FUNCTION

Identifies the function on which the privilege is granted. The function must exist at the current server and it must be a user-defined function. The function can be identified by its name, function signature, or specific name.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be granted. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type, ...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

GRANT (Function or Procedure Privileges)

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE or SPECIFIC PROCEDURE

Identifies the procedure on which the privilege is granted. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*

Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*

Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. 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 granted. Synonyms for data types are considered a match.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name

Identifies the name of the procedure.

(parameter-type, ...)

Identifies the parameters of the procedure.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If

GRANT (Function or Procedure Privileges)

the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).

- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR

Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC PROCEDURE *specific-name*

Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

TO

Indicates to whom the privileges are granted.

authorization-name,...

Lists one or more authorization IDs.

PUBLIC

Grants the privileges to a set of users (authorization IDs). For more information, see “Authorization, Privileges and Object Ownership” on page 36.

WITH GRANT OPTION

Allows the specified *authorization-names* to grant privileges on the functions or procedures specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant privileges on the functions or procedures specified in the ON clause to another user unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Note

Corresponding System Authorities: Privileges granted to either an SQL or external function or procedure are granted to its associated program (*PGM) or service program (*SRVPGM) object. Privileges granted to a Java external function or procedure are granted to the associated class file or jar file.

GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

GRANT (Function or Procedure Privileges)

Table 53. Privileges Granted to or Revoked from Non-Java Functions or Procedures

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Function or Procedure
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *OBJMGT (Revoke only)
ALTER	*OBJALTER
EXECUTE	*EXECUTE *OBJOPR
WITH GRANT OPTION	*OBJMGT

Table 54. Privileges Granted to or Revoked from Java Functions or Procedures

SQL Privilege	Corresponding Data Authorities when Granting to or Revoking from a Java Function or Procedure	Corresponding Object Authorities when Granting to or Revoking from a Java Function or Procedure
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*RWX	*OBJEXIST *OBJALTER *OBJMGT (Revoke only)
ALTER	*R	*OBJALTER
EXECUTE	*RX	*EXECUTE
WITH GRANT OPTION	*RWX	*OBJMGT

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword RUN can be used as a synonym for EXECUTE.

Example

Grant the EXECUTE privilege on procedure PROCA to PUBLIC.

```
GRANT EXECUTE
ON PROCEDURE PROCA
TO PUBLIC
```

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 interactively. It is an executable statement that can be dynamically prepared.

Authorization

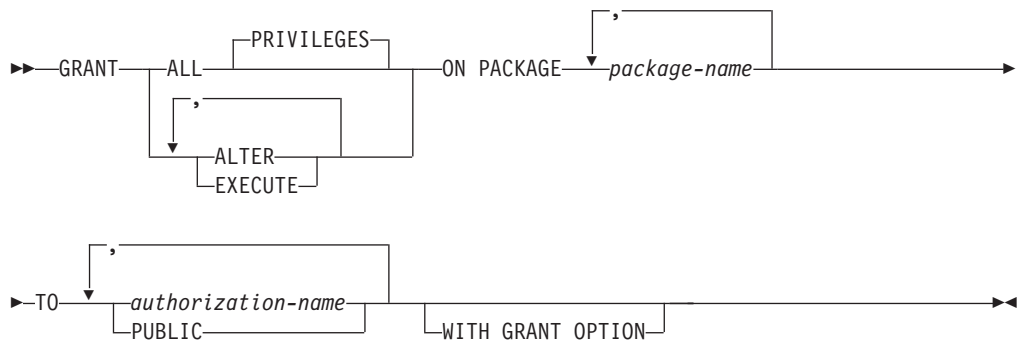
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each package identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the package
 - The system authority *EXECUTE on the library containing the package
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the package
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified packages. Note that granting ALL PRIVILEGES on a package is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword grants the privilege described.

ALTER

Grants the privilege to use the COMMENT and LABEL statements.

GRANT (Package Privileges)

EXECUTE

Grants the privilege to execute statements in a package.

ON PACKAGE *package-name*

Identifies the packages on which you are granting the privilege. The *package-name* must identify a package that exists at the current server.

TO

Indicates to whom the privileges are granted.

authorization-name,...

Lists one or more authorization IDs.

PUBLIC

Grants the privileges to a set of users (authorization IDs). For more information, see “Authorization, Privileges and Object Ownership” on page 36.

WITH GRANT OPTION

Allows the specified *authorization-names* to grant privileges on the packages specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant privileges on the packages specified in the ON clause to another user unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Note

Corresponding System Authorities: GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges:

Table 55. Privileges Granted to or Revoked from Packages

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Package
ALL (Grant or revoke of ALL grants or revokes only those privileges the authorization ID of the statement has)	*OBJALTER *OBJOPR *EXECUTE *OBJMGT (Revoke only)
ALTER	*OBJALTER
EXECUTE	*EXECUTE *OBJOPR
WITH GRANT OPTION	*OBJMGT

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword RUN can be used as a synonym for EXECUTE.
- The keyword PROGRAM can be used as a synonym for PACKAGE.

Example

Grant the EXECUTE privilege on package PKGA to PUBLIC.

```
GRANT EXECUTE
ON PACKAGE PKGA
TO PUBLIC
```

GRANT (Table or View Privileges)

GRANT (Table or View Privileges)

This form of the GRANT statement grants privileges on tables or views.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

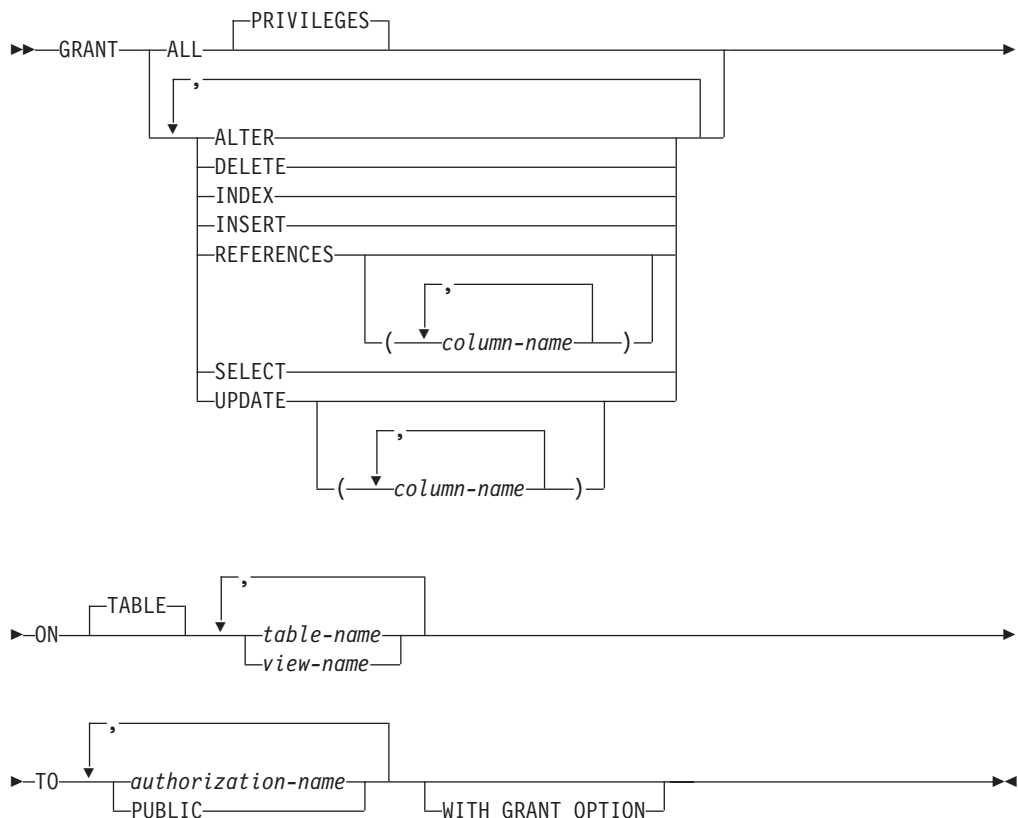
The privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the table or view
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

If WITH GRANT OPTION is specified, the privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Grants one or more privileges. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the specified tables or views. Note that granting ALL PRIVILEGES on a table or view is not the same as granting the system authority of *ALL.

ALTER

Grants the privilege to alter the specified table or create or drop a trigger on the specified table. Grants the privilege to use the COMMENT and LABEL statements on tables and views.

DELETE

Grants the privilege to delete rows from the specified table or view. If a view is specified, it must be a deletable view.

INDEX

Grants the privilege to create an index on the specified table. This privilege cannot be granted on a view.

INSERT

Grants the privilege to insert rows into the specified table or view. If a view is specified, it must be an insertable view.

REFERENCES

Grants the privilege to add a referential constraint in which each specified table is a parent. If a list of columns is not specified or if REFERENCES is granted to all columns of the table or view via the specification of ALL PRIVILEGES, the grantee(s) can add referential constraints using all columns of each table specified in the ON clause as a parent key, even those added later via the ALTER TABLE statement. This privilege can be granted on a view, but the privilege is not used for a view.

REFERENCES (*column-name*,...)

Grants the privilege to add a referential constraint in which each specified table is a parent 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 each table specified in the ON clause. This privilege can be granted on the columns of a view, but the privilege is not used for a view.

SELECT

Grants the privilege to create a view or read data from the specified table or view. For example, the SELECT privilege is required if a table or view is specified in a query.

UPDATE

Grants the privilege to update rows in the specified table or view. If a list of columns is not specified or if UPDATE is granted to all columns of the table or view via the specification of ALL PRIVILEGES, the grantee(s) can update all updateable columns on each table specified in the ON clause, even those added later via the ALTER TABLE statement. If a view is specified, it must be an updatable view.

UPDATE (*column-name*,...)

Grants the privilege to use the UPDATE statement to update only those columns that are identified in the column list. Each *column-name* must be an unqualified name that identifies a column of each table and view specified in the ON clause. If a view is specified, it must be an updatable view and the specified columns must be updatable columns.

GRANT (Table or View Privileges)

| **ON** *table-name* or *view-name*,...
| Identifies the tables or views on which the privileges are granted. The
| *table-name* or *view-name* must identify a table or view that exists at the current
| server, but must not identify a global temporary table.

TO
Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs.

PUBLIC

Grants the privileges to a set of users (authorization IDs). For more information, see “Authorization, Privileges and Object Ownership” on page 36.

WITH GRANT OPTION

Allows the specified *authorization-names* to grant privileges on the tables and views specified in the ON clause to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-names* cannot grant privileges on the tables and views specified in the ON clause unless they have received that authority from some other source (for example, from a grant of the system authority *OBJMGT).

Notes

Corresponding System Authorities: The GRANT and REVOKE statements assign and remove system authorities for SQL objects. The following table describes the system authorities that correspond to the SQL privileges when granting to a table. The left column lists the SQL privilege. The right column lists the equivalent system authorities that are granted or revoked.

GRANT (Table or View Privileges)

Table 56. Privileges Granted to or Revoked from Tables

SQL Privilege	Corresponding System Authorities when Granting to or Revoking from a Table
ALL (GRANT or revoke of ALL only grants or revokes those privileges the authorization ID of the statement has)	*OBJALTER ⁵⁹ *OBJMGT (Revoke only) *OBJOPR *OBJREF *ADD *DLT *READ *UPD
ALTER	*OBJALTER ⁶⁰
DELETE	*OBJOPR *DLT
INDEX	*OBJALTER ⁶⁰
INSERT	*OBJOPR *ADD
REFERENCES	*OBJREF ⁶⁰
SELECT	*OBJOPR *READ
UPDATE	*OBJOPR *UPD
WITH GRANT OPTION	*OBJMGT

The following table describes the system authorities that correspond to the SQL privileges when granting to a view. The left column lists the SQL privilege. The middle column lists the equivalent system authorities that are granted to or revoked from the view itself. The right column lists the system authorities that are granted to all tables and views referenced in the views definition, and if a view is referenced, all tables and views referenced in its definition, and so on. ⁶¹

If a view references more than one table or view, the *DLT, *ADD, and *UPD system authorities are only granted to the first table or view in the subselect of the view definition. The *READ system authority is granted to all tables and views referenced in the view definition.

If more than one system authority will be granted with an SQL privilege, and any one of the authorities cannot be granted, then a warning occurs and no authorities will be granted for that privilege. Unlike GRANT, REVOKE only revokes system authorities to the view. No system authorities are revoked from the referenced tables and views.

59. The SQL INDEX and ALTER privilege correspond to the same system authority of *OBJALTER. Granting both INDEX and ALTER will not provide the user with any additional authorities.

60. If the WITH GRANT OPTION is given to a user, the user will also be able to perform the functions given by ALTER and REFERENCES authority.

61. The specified rights are only granted to the tables and views referenced in the view definition if the user to whom the rights are being granted doesn't already have the rights from another authority source, for example public authority.

GRANT (Table or View Privileges)

Table 57. Privileges Granted to or Revoked from Views

SQL Privilege	Corresponding System Authorities Granted to or Revoked from View	Corresponding System Authorities Granted to or Revoked from Referenced Tables and Views
ALL (GRANT or REVOKE of ALL only grants or revokes those privileges the authorization ID of the statement has)	*OBJALTER *OBJMGT (Revoke only) *OBJOPR *OBJREF *ADD *DLT *READ *UPD	*ADD *DLT *READ *UPD
ALTER	*OBJALTER ⁶²	None
DELETE	*OBJOPR *DLT	*DLT
INDEX	Not Applicable	Not Applicable
INSERT	*OBJOPR *ADD	*ADD
REFERENCES	*OBJREF ⁶²	None
SELECT	*OBJOPR *READ	*READ
UPDATE	*OBJOPR *UPD	*UPD
WITH GRANT OPTION	*OBJMGT	None

Examples

Example 1: Grant all privileges on the table WESTERN_CR to PUBLIC.

```
GRANT ALL PRIVILEGES ON WESTERN_CR  
TO PUBLIC
```

Example 2: Grant the appropriate privileges on the CALENDAR table so that 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 PHIL, CLAIRE
```

Example 3: Grant column privileges on TABLE1 and VIEW1 to FRED. Note that both columns specified in this GRANT statement must be found in both TABLE1 and VIEW1.

```
GRANT UPDATE(column_1, column_2)  
ON TABLE1, VIEW1  
TO FRED WITH GRANT OPTION
```

62. If the WITH GRANT OPTION is given to a user, the user will also be able to perform the functions given by ALTER and REFERENCES authority.

HOLD LOCATOR

The HOLD LOCATOR statement allows a LOB locator variable to retain its association with a value beyond a unit of work.

Invocation

This statement can only be embedded in an application program. It cannot be issued interactively. It is an executable statement that can be dynamically prepared. However, the EXECUTE statement with the USING clause must be used to execute the prepared statement. HOLD LOCATOR cannot be used with the EXECUTE IMMEDIATE statement.

Authorization

None required.

Syntax

```

HOLD LOCATOR host-variable

```

Description

host-variable,...

Identifies a host variable that must be declared in accordance with the rules for declaring host variable locator variables. An indicator variable must not be specified. The locator variable type must be a binary large object locator, a character large object locator, or a double-byte character large object locator.

After the HOLD LOCATOR statement is executed, each locator variable in the *host-variable* list has the hold property.

The host variable must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a CALL, FETCH, SELECT INTO, SET variable, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is raised.

If more than one host variable is specified in the HOLD LOCATOR statement and an error occurs on one of the locators, no locators will be held.

Note

A host-variable LOB locator variable that has the hold property is freed (has its association between it and its value removed) when:

- The SQL FREE LOCATOR statement is executed for the locator variable.
- The SQL ROLLBACK statement is executed.
- The SQL session is terminated.

Example

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that locators have been established in a program to represent the

HOLD LOCATOR

| values represented by the columns. Give the CLOB locator variables LOCRES and
| LOCHIST, and the BLOB locator variable LOCPIC the hold property.

| **HOLD LOCATOR** :LOCRES, :LOCHIST, :LOCPIC

|

INCLUDE

The INCLUDE statement must be specified at a point in your program where the resulting source statements are acceptable to the compiler.

Notes

If the CCSID of the source file specified on the SRCFILE parameter is different from the CCSID of the source file specified on the INCFILE parameter, the source from the INCLUDE statement is converted to the CCSID of the source file.

Example

Include an SQL descriptor area in a C program.

```
EXEC SQL INCLUDE SQLDA;

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.

There are three forms of this statement:

- The *INSERT using VALUES* form is used to insert a single row into the table or view using the values provided or referenced.
- The *INSERT using SELECT* form is used to insert one or more rows into the table or view using values from other tables or views.
- The *INSERT using n ROWS* form is used to insert multiple rows into the table or view using the values provided in a host-structure-array.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared with the exception of the *n ROWS* form, which must be a static statement embedded in an application program. The *n ROWS* form is not allowed in a REXX procedure.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table or view identified in the statement:
 - The INSERT privilege on the table or view, and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The authorization ID of the statement has the INSERT privilege on a table when:

- It is the owner of the table,
- It has been granted the INSERT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *ADD on the table.

The authorization ID of the statement has the INSERT privilege on a view when:⁶³

- It has been granted the INSERT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *ADD on the view and the system authority *ADD on the first table or view in the first FROM clause of the view definition; and if this is a view, then the system authority *ADD on the first table or view in the first FROM clause of that view definition; and so forth.

If a subselect is specified, the privileges held by the authorization ID of the statement must also include one of the following:

- For each table or view identified in the subselect:
 - The SELECT privilege on the table or view, and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

63. When a view is created, the owner does not necessarily acquire the INSERT privilege on the view. The owner only acquires the INSERT privilege if the view allows inserts and the owner also has the INSERT privilege on the first table referenced in the subselect.

INSERT

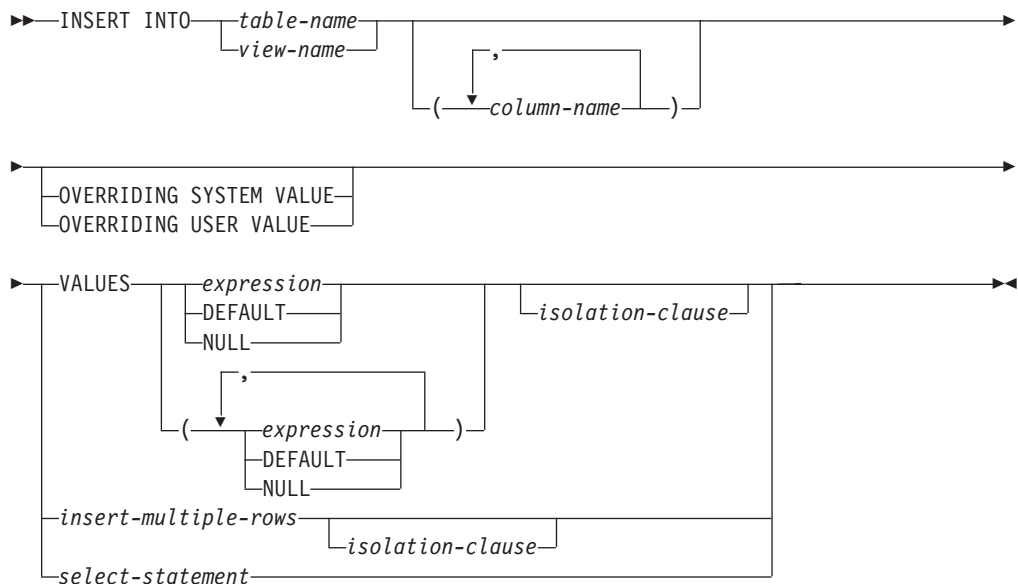
The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table

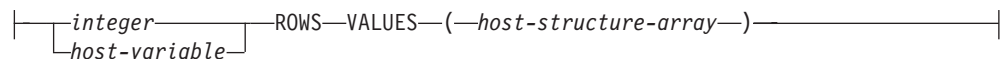
The authorization ID of the statement has the SELECT privilege on a view when:

- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views that this view is directly or indirectly dependent on. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

Syntax



insert-multiple-rows:



isolation-clause:



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 current server, but it must not identify a catalog table, a view of a catalog table, or a view that is not insertable. For an explanation of insertable views, see “CREATE VIEW” on page 569.

(*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 is not updatable must not be identified. 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 those columns. For an explanation of updatable columns in views, see “CREATE VIEW” on page 569.

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.

If the INSERT statement is embedded in an application and the referenced table or view exists at create program time, the statement is prepared at create program time. Otherwise, the statement is prepared at the first successful execute of the INSERT statement.

OVERRIDING SYSTEM VALUE or OVERRIDING USER VALUE

Specifies whether system generated values or user-specified values for a ROWID or identity column are used. If **OVERRIDING SYSTEM VALUE** is specified, the implicit or explicit list of columns for the INSERT statement must contain a column defined as **GENERATED ALWAYS**. If **OVERRIDING USER VALUE** is specified, the implicit or explicit list of columns for the INSERT statement must contain a column defined as either **GENERATED ALWAYS** or **GENERATED BY DEFAULT**.

OVERRIDING SYSTEM VALUE

Specifies that the value specified in the VALUES clause or produced by a fullselect for a column that is defined as **GENERATED ALWAYS** is used. A system-generated value is not inserted.

OVERRIDING USER VALUE

Specifies that the value specified in the VALUES clause or produced by a fullselect for a column that is defined as either **GENERATED ALWAYS** or **GENERATED BY DEFAULT** is ignored. Instead, a system-generated value is inserted, overriding the user-specified value.

If neither **OVERRIDING SYSTEM VALUE** nor **OVERRIDING USER VALUE** is specified:

- A value cannot be specified for a ROWID or identity column that is defined as **GENERATED ALWAYS**.
- A value can be specified for a ROWID or identity column that is defined as **GENERATED BY DEFAULT**. If a value is specified that value is assigned to the column. However, a value can be inserted into a ROWID column defined **BY DEFAULT** only if the specified value is a valid row ID value that was previously generated by DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries. When a value is inserted into an identity column defined **BY DEFAULT**, the database manager does not verify that the specified value is a unique value for the column unless the identity column is the sole key in a

INSERT

unique constraint or unique index. Without a unique constraint or unique index, the database manager can guarantee unique values only among the set of system-generated values as long as NO CYCLE is in effect.

If a value is not specified the database manager generates a new value.

VALUES

Specifies one new row in the form of a list of values. Each host variable in the clause must identify a host structure or host variable that is declared in accordance with the rules for declaring host structures and host variables. In the operational form of the statement, a reference to a host structure is replaced by a reference to each of its variables. For further information on host variables and structures, see “References to Host Variables” on page 114 and “Host Structures” on page 120.

The number of values in the VALUES clause must equal the number of names in the column list. The first value is inserted in the first column in the list, the second value in the second column, and so on.

expression

An *expression* of the type described in “Expressions” on page 129, that does not include a column function or column name. If *expression* is a *host-variable*, the host variable can identify a structure.

DEFAULT

Specifies that the default value is assigned to a column. 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 *column-definition* in “CREATE TABLE” on page 525).
- If the WITH DEFAULT clause or the NOT NULL clause is not used, the value inserted is NULL.
- 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.
- If the column is a ROWID or identity column, the database manager will generate a new value.

For a ROWID or an identity column that was defined as GENERATED ALWAYS, you must specify DEFAULT unless you specify the OVERRIDING USER VALUE clause to indicate that any user-specified value will be ignored and a unique system-generated value will be inserted.

NULL

Specifies the value for a column is the null value. NULL should only be specified for nullable columns.

select-statement

Specifies a set of new rows in the form of the result table of a select-statement. The FOR READ ONLY, FOR UPDATE, and OPTIMIZE clauses are not valid for a select-statement used with insert. If an ORDER BY clause is specified on the select-statement, the rows are inserted according to the values of the columns identified in the ORDER BY clause. For an explanation of select-statement, see “select-statement” on page 349.

There can be one, more than one, or zero rows inserted when using the select-statement. If no rows are inserted, SQLCODE is set to +100 and SQLSTATE is set to '02000'.

When the base object of the INSERT and a base object of any subselect in the select statement are the same table, the select statement 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.

isolation-clause

Specifies the isolation level to be used for this statement.

WITH

Introduces the isolation level, which may be one of:

- RR Repeatable read
- RS Read stability
- CS Cursor stability
- UR Uncommitted read
- NC No commit

If *isolation-clause* is not specified the default isolation is used. See “*isolation-clause*” on page 357 for a description of how the default is determined.

insert-multiple-rows

integer or *host-variable* **ROWS**

Specifies the number of rows to be inserted. If a host-variable is specified, it must be numeric with zero scale and cannot include an indicator variable.

VALUES (*host-structure-array*)

Specifies a set of new rows in the form of an array of host structures. The host-structure-array must be declared in the program in accordance with the rules for declaring host structure arrays. A parameter marker may be used in place of the host-structure-array name.

The number of variables in the host structure must equal the number of names in the column-list. The first host structure in the array corresponds to the first row, the second host structure in the array corresponds to the second row, and so on. In addition, the first variable in the host structure corresponds with the first column of the row, the second variable in the host structure corresponds with the second column of the row, and so on.

For an explanation of arrays of host structures see “Host Structure Arrays” on page 122.

Insert-multiple-rows is not allowed if any of the insert values are LOBs or if the current connection is to a non-iSeries remote server.

INSERT Rules

Default Values: The value inserted in any column that is not in the column list is the default value of the column. Columns without a default value must be included in the column list. Similarly, if you insert into a view, the default value is inserted into any column of the base table that is not included in the view. Hence, all columns of the base table that are not in the view must have default values.

Assignment: Insert values are assigned to columns in accordance with the assignment rules described in Chapter 2.

INSERT

Validity: If the identified table or the base table of the identified view has one or more unique indexes or unique constraints, each row inserted into the table must conform to the constraints imposed by those indexes.

The unique indexes and unique constraints are effectively checked at the end of the statement unless COMMIT(*NONE) was specified. In the case of a multiple-row insert, this would occur after all rows were inserted and any associated triggers were activated. If COMMIT(*NONE) is specified, checking is performed as each row is inserted.

If the identified table or the base table of the identified view has one or more check constraints, each check constraint must be true or unknown for each row inserted into the table.

The check constraints are effectively checked at the end of the statement. In the case of a multiple-row insert, this would occur after all rows were inserted.

If a view is identified, the inserted rows must conform to any applicable WITH CHECK OPTION. For more information, see "CREATE VIEW" on page 569.

Triggers: If the identified table or the base table of the identified view has an insert trigger, the trigger is activated. A trigger might cause other statements to be executed or raise error conditions based on the insert values.

Referential Integrity: Each nonnull insert value of a foreign key must equal some value of the parent key of the parent table in the relationship.

The referential constraints (other than a referential constraint with a RESTRICT delete rule) are effectively checked at the end of the statement. In the case of a multiple-row insert, this would occur after all rows were inserted and any associated triggers were activated.

Notes

Insert operation errors: If an insert value violates any constraints, or if any other error occurs during the execution of an INSERT statement and COMMIT(*NONE) was not specified, all changes made during the execution of the statement are backed out. However, other changes in the unit of work made prior to the error are not backed out. If COMMIT(*NONE) is specified, changes are not backed out.

Number of rows inserted: After executing an INSERT statement, the value of SQLERRD(3) of the SQLCA is the number of rows that the database manager inserted. The value in SQLERRD(3) does not include the number of rows that were inserted as a result of a trigger.

Locking: If COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) is specified, one or more exclusive locks are acquired during the execution of a successful INSERT statement. Until the locks are released by a commit or rollback operation, an inserted row can only be accessed by:

- The application process that performed the insert
- Another application process using COMMIT(*NONE) or COMMIT(*CHG) through a read-only cursor, SELECT INTO statement, or subquery

The locks can prevent other application processes from performing operations on the table. For further information about locking, see the description of the

COMMIT, ROLLBACK, and LOCK TABLE statements. Also, see “Isolation Level” on page 21 and the Database Programming book.

A maximum of 500 000 000 rows can be inserted or changed in any single INSERT statement when COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) was specified. The number of rows changed includes any rows inserted, updated, or deleted under the same commitment definition as a result of a trigger.

REXX: Host variables cannot be used in the INSERT statement within a REXX procedure. Instead, the INSERT must be the object of a PREPARE and EXECUTE using parameter markers.

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword NONE can be used as a synonym for NC.
- The keyword CHG can be used as a synonym for UR.
- The keyword ALL can be used as a synonym for RS.

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'
- 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: Create a table MA_EMPPROJACT with the same columns as the EMPPROJACT table. Populate MA_EMPPROJACT with the rows from the EMPPROJACT table with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE TABLE MA_EMPPROJACT LIKE EMPPROJACT

INSERT INTO MA_EMPPROJACT
SELECT * FROM EMPPROJACT
WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 4: Use a Java program statement to add a skeleton project to the PROJECT table on the connection context 'ctx'. 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.

```
#sql [ctx] { INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
VALUES (:PRJNO, :PRJNM, :DPTNO, :REMP, CURRENT DATE) };
```

INSERT

Example 5: In a PL/I program, use a blocked insert to add 10 rows to table DEPARTMENT. The host structure array DEPT contains the data to be inserted.

```
DCL 1 DEPT(10),
    3 DEPT CHAR(3),
    3 LASTNAME CHAR(29) VARYING,
    3 WORKDEPT CHAR(6),
    3 JOB CHAR(3);

EXEC SQL INSERT INTO DEPARTMENT 10 ROWS VALUES (:DEPT);
```

Example 6: Insert a new project into the EMPPROJECT table using the Read Uncommitted (UR, CHG) option:

```
INSERT INTO EMPPROJECT
VALUES ('000140', 'PL2100', 30)
WITH CHG
```

LABEL

The LABEL statement adds or replaces labels in the catalog descriptions of tables, views, aliases, packages, or columns.

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 of the statement must include at least one of the following:

- For the table, view, alias, or package identified in the statement,
 - The ALTER privilege on the table, view, alias, or package, and
 - The system authority *EXECUTE on the library containing the table, view, alias, or package
- Administrative authority

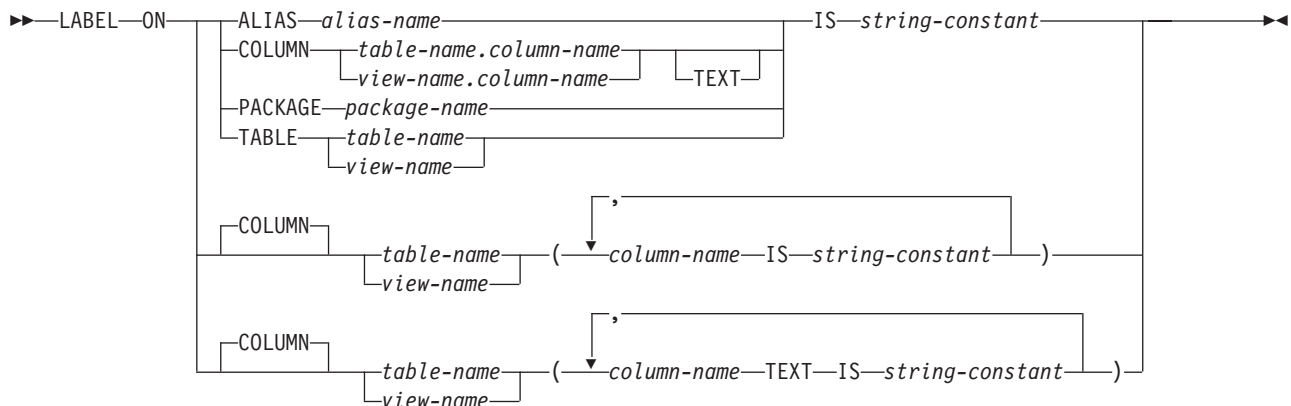
The authorization ID of the statement has the ALTER privilege on the table, view or package when:

- It is the owner of the table, view or package,
- It has been granted the ALTER privilege to the table, view or package, or
- It has been granted the system authorities of either *OBJALTER or *OBJMGT to the table, view, or package

The authorization ID of the statement has the ALTER privilege on an alias when:

- It is the owner of the alias, or
- It has been granted the system authorities of either *OBJALTER or *OBJMGT to the alias

Syntax



LABEL

Description

ALIAS

Specifies that the label is for an alias. Labels on aliases are implemented as system object text.

alias-name

Identifies the alias to which the label applies. The name must identify an alias that exists at the current server.

COLUMN

Specifies that the label is for a column. Labels on columns are implemented as system column headings or column text. Column headings are used when displaying or printing query results.

table-name.column-name or *view-name.column-name*

Identifies the column to which the label applies. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a global temporary table. The *column-name* must identify a column of that table or view.

TEXT

Specifies that OS/400 column text is specified. If TEXT is omitted, a column heading is specified.

PACKAGE

Specifies that the label is for a package. Labels on packages are implemented as system object text.

package-name

Identifies the package to which the label applies. The name must identify a package that exists at the current server.

TABLE

Specifies that the label is for a table or a view. Labels on tables or views are implemented as system object text.

table-name or *view-name*

Identifies the table or view on which you want to add a label. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a global temporary table.

IS

Introduces the label you want to provide.

string-constant

Can be any SQL character-string constant of up to either 50 bytes in length for tables, views, aliases, SQL packages, or column text, or 60 bytes in length for column headings. The constant may contain single-byte and double-byte characters.

The label for a column heading consists of three 20-byte segments. Interactive SQL, the Query/400 program, DB2 Query Manager and SQL Development Kit for iSeries, and other products can display or print each 20-byte segment on a separate line. If the label for a column contains mixed data, each 20-byte segment must be a valid mixed data character string. The shift characters must be paired within each 20-byte segment.

Notes

Column Headings: Column headings are used when displaying or printing query results. The first column heading is displayed or printed on the first line, the

second column heading is displayed or printed on the second line, and the third column heading is displayed or printed on the third line. The column headings can be up to 60 bytes in length, where the first 20 bytes is the first column heading, the second 20 bytes is the second column heading, and the third 20 bytes is the third column heading. Blanks are trimmed from the end of each 20-byte column heading.

All 60 bytes of column heading information are available in the catalog view SYSCOLUMNS; however, only the first column heading is returned in an SQLDA on a DESCRIBE or DESCRIBE TABLE statement.

Column text is not returned on a DESCRIBE or DESCRIBE TABLE statement. When the database manager changes the column heading information in a record format description that is shared, the change is reflected in all files sharing the format description. To find out if a file shares a format with another file, use the RCDFMT parameter on the CL command, Display Database Relations (DSPDBR).

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword PROGRAM can be used as a synonym for PACKAGE.

Examples

Example 1: Enter a label on the DEPTNO column of table DEPARTMENT.

```
LABEL ON COLUMN DEPARTMENT.DEPTNO
IS 'DEPARTMENT NUMBER'
```

Example 2: Enter a label on the DEPTNO column of table DEPARTMENT where the column heading is shown on two separate lines.

```
LABEL ON COLUMN DEPARTMENT.DEPTNO
IS 'Department          Number'
```

Example 3: Enter a label on the PAYROLL package.

```
LABEL ON PACKAGE PAYROLL
IS 'Payroll Package'
```

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

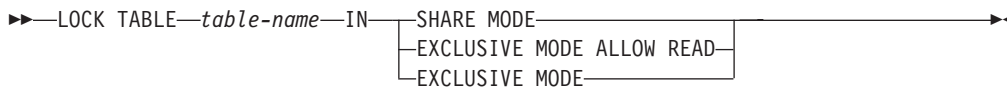
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table identified in the statement,
 - The system authority of *OBJOPR on the table, and
 - The system authority *EXECUTE on the library containing the table
- Administrative authority

Syntax



Description

table-name

Identifies the table to be locked. The table-name must identify a base table that exists at the current server, but must not identify a catalog table or a global temporary table.

IN SHARE MODE

Prevents concurrent application processes from executing any but read-only operations on the table.

A shared lock (*SHRNUP) is acquired for the application process in which the statement is executed. Other application processes may also acquire a shared lock (*SHRNUP) and prevent this application process from executing any but read-only operations.

IN EXCLUSIVE MODE ALLOW READ

Prevents concurrent application processes from executing any but read-only operations on the table.

An exclusive allow read lock (*EXCLRD) is acquired for the application process in which the statement is executed. Other application processes may not acquire a shared lock (*SHRNUP) and cannot prevent this application process from executing updates, deletes, and inserts on the table.

IN EXCLUSIVE MODE

Prevents concurrent application processes from executing any operations at all on the table.

An exclusive lock (*EXCL) is acquired for the application process in which the statement is executed.

Notes

Locks Obtained: Locking is used to prevent concurrent operations.

The lock is released:

- When the unit of work ends, unless the unit of work is ended by a COMMIT HOLD or ROLLBACK HOLD
- When the first SQL program in the program stack ends, unless CLOSQLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) was specified on the CRTSQLxxx command
- When the activation group ends
- When the connection is changed using a CONNECT (Type 1) statement
- When the connection associated with the lock is disconnected using the DISCONNECT statement
- When the connection is in the release-pending state and a successful COMMIT occurs

You may also issue the Deallocate Object (DLCOBJ) command to unlock the table.

Conflicting locks already held by other application processes will cause your application to wait up to the default wait time.

Example

Obtain a lock on the DEPARTMENT table.

```
LOCK TABLE DEPARTMENT IN EXCLUSIVE MODE
```

OPEN

The OPEN statement opens a cursor.

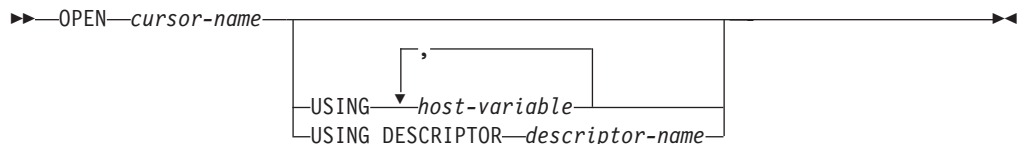
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See "DECLARE CURSOR" on page 576 for the authorization required to use a cursor.

Syntax



Description

cursor-name

Identifies the cursor to be opened. The *cursor-name* must identify a declared cursor as explained in the Notes for the DECLARE CURSOR statement. When the OPEN statement is executed, the cursor must be in the closed state.

The SELECT statement associated with the cursor is either:

- The *select-statement* specified in the DECLARE CURSOR statement, or
- The prepared *select-statement* identified by the *statement-name* specified in the DECLARE CURSOR statement. If the statement has not been successfully prepared, or is not a *select-statement*, the cursor cannot be successfully opened.

The result table of the cursor is derived by evaluating the SELECT statement. The evaluation uses the current values of any special registers specified in the SELECT statement and the current values of any host variables specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table can be derived during the execution of the OPEN statement and a temporary table can be created to hold them; or they can 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 position 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 691. If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, you must use USING. If the prepared statement does not include parameter markers, USING is ignored.

host-variable,...

Identifies host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. A reference to a host structure is replaced by a reference to each of its variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement.

DESCRIPTOR *descriptor-name*

Identifies an SQLDA that must contain a valid description of input host variables.

Before the OPEN statement is processed, the user must set the following fields in the SQLDA. (The rules for REXX are different. For more information see the SQL Programming with Host Languages book.)

- 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. If LOBs or distinct types are present in the results, there must be additional SQLVAR entries for each parameter. For more information about the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR occurrences, see Appendix D, "SQL Descriptor Area (SQLDA)" on page 847.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement. The *n*th variable described by the SQLDA corresponds to the *n*th parameter marker in the prepared statement.

Note that because RPG/400 does not provide the facility for setting pointers and the SQLDA uses pointers to locate the appropriate host variables, you will have to set these pointers outside your RPG/400 application.

Notes

Closed state of cursors: All cursors in a program are in the closed state when:

- The program is called:
 - If CLOSQLCSR(*ENDPGM) is specified, all cursors are in the closed state each time the program is called.
 - If CLOSQLCSR(*ENDSQL) is specified, all cursors are in the closed state only the first time the program is called as long as one SQL program remains on the call stack.
 - If CLOSQLCSR(*ENDJOB) is specified, all cursors are in the closed state only the first time the program is called as long as the job remains active.
 - If CLOSQLCSR(*ENDMOD) is specified, all cursors are in the closed state each time the module is initiated.
 - If CLOSQLCSR(*ENDACTGRP) is specified, all cursors are in the closed state only the first time the module in the program is initiated in the activation group.

OPEN

- A program starts a new unit of work by executing a COMMIT or ROLLBACK statement without a HOLD option. Cursors declared with the HOLD option are not closed by a COMMIT statement.
- A CONNECT (Type 1) statement was executed.

A cursor can also be in the closed state because:

- A CLOSE statement was executed.
- A DISCONNECT statement disconnected the connection with which the cursor was associated.
- The connection with which the cursor was associated was in the release-pending state and a successful COMMIT occurred.
- A CONNECT (Type 1) statement was executed.

To retrieve rows from the result table of a cursor, the FETCH statement must be executed 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: If the result table of a cursor is not read-only, its rows are derived during the execution of subsequent FETCH statements. The same method may be used for a read-only result table. However, if a result table is read-only, DB2 UDB for iSeries may choose to use the temporary table method instead. With this method the entire result table is inserted into 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.
- The INSERT, UPDATE, and DELETE statements that are executed 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. The effect of such operations is not always predictable. For example, if cursor C is positioned on a row of its result table defined as SELECT * FROM T, and you insert a row into T, the effect of that insert on the result table is not predictable because its rows are not ordered. A subsequent FETCH C might or might not retrieve the new row of T.

Parameter Marker Replacement: When the SELECT statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by its corresponding host variable. The replacement of a parameter marker is an assignment operation in which the source is the value of the host variable, and the target is a variable within the database manager. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker. For the rules that affect parameter markers, see Table 58 on page 694.

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 number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.

- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, the value of V must not be null.

However, unlike the rules for assigning a value to a column:

- If V is a string, the value will be truncated (without an error), if its length is greater than the length attribute 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.

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. Assume each prepared select-statement always defines two items in its select list with the first item having a data type of integer and the second item having a data type of VARCHAR(64). (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 3, but in this case the number and data types of the items in the select statement are not known.

```
EXEC SQL  BEGIN DECLARE SECTION;
          char stmt1_str[200];
EXEC SQL  END DECLARE SECTION;
```

OPEN

```
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 creates an executable form of an SQL statement from a character-string form of the statement. The character-string form is called a *statement string*, and the executable form is called a *prepared statement*.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

The authorization rules are the same as those defined for the SQL statement specified by the PREPARE statement. For example, see “select-statement” on page 349 for the authorization rules that apply when a SELECT statement is prepared.

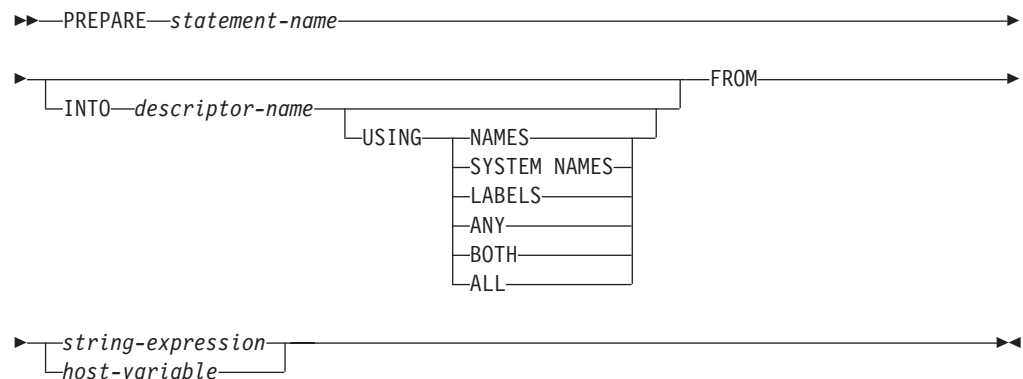
If DLYPRP(*NO) is specified on the CRTSQLxxx command, the authorization checking is performed when the statement is prepared, except:

- If a DROP SCHEMA statement is prepared, the system authority *OBJEXIST on all objects in the schema is not checked until the statement is executed.
- If a DROP TABLE statement is prepared, the system authority *OBJEXIST on all views, indexes, and logical files that reference the table is not checked until the statement is executed.
- If a DROP VIEW statement is prepared, the system authority of *OBJEXIST on all views that reference the view is not checked until the statement is executed.

If DLYPRP(*YES) is specified on the CRTSQLxxx command, all authorization checking is deferred until the statement is executed or used in an OPEN statement.

The authorization ID of the statement is the run-time authorization ID unless DYNUSRPRF(*OWNER) was specified on the CRTSQLxxx command when the program was created. For more information, see “Authorization IDs and Authorization-Names” on page 57.

Syntax



PREPARE

Description

statement-name

Names the prepared statement. If the name identifies an existing prepared statement, that prepared statement is destroyed if:

- it was prepared in the same instance of the same program, or
- CLOSQLCSR(*ENDJOB), CLOSQLCSR(*ENDACTGRP), or CLOSQLCSR(*ENDSQL) are specified on the CRTSQLxxx commands associated with both prepared statements.

The name must not identify a prepared statement that is the SELECT statement of an open cursor of this instance of the program.

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. Thus, the PREPARE statement:

```
EXEC SQL PREPARE S1 INTO :SQLDA FROM :V1;
```

is equivalent to:

```
EXEC SQL PREPARE S1 FROM :V1;  
EXEC SQL DESCRIBE S1 INTO :SQLDA;
```

descriptor-name

Identifies an SQL descriptor area (SQLDA), which is described in Appendix D, "SQL Descriptor Area (SQLDA)" on page 847. Before the PREPARE statement is executed, the following variable in the SQLDA must be set (The rules for REXX are different. For more information, see the SQL Programming with Host Languages book.):

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 PREPARE statement is executed. For information on techniques to determine the number of occurrences required, see "Determining How Many SQLVAR Occurrences are Needed" on page 850.

See "DESCRIBE" on page 618 for an explanation of the information that is placed in the SQLDA.

USING

Specifies what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to length 0.

NAMES

Assigns the name of the column. This is the default. For a prepared statement where the names are explicitly specified in the select-list, the name specified is returned.

SYSTEM NAMES

Assigns the system column name of the column.

LABELS

Assigns the label of the column. (Column labels are defined by the LABEL statement.) Only the first 20 bytes of the label are returned.

ANY

Assigns the column label. If the column has no label, the label is the column name.

BOTH

Assigns both the label and name of the column. In this case, two or three occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $2*n$ or $3*n$ (where n is the number of columns in the table or view). The first n occurrences of SQLVAR contain the column names. Either the second or third n occurrences contain the column labels. If there are no distinct types, the labels are returned in the second set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries.

If the same SQLDA is used on a subsequent FETCH statement, set SQLN to n after the PREPARE is complete.

ALL

Assigns the label, column name, and system column name. In this case three or four occurrences of SQLVAR per column, depending on whether the result set contains distinct types, are needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to $3*n$ or $4*n$ (where n is the number of columns in the result table). The first n occurrences of SQLVAR contain the system column names. The second or third n occurrences contain the column labels. The third or fourth n occurrences contain the column names. If there are no distinct types, the labels are returned in the second set of SQLVAR entries and the column names are returned in the third set of SQLVAR entries. Otherwise, the labels are returned in the third set of SQLVAR entries and the column names are returned in the fourth set of SQLVAR entries.

If the same SQLDA is used on a subsequent FETCH statement, set SQLN to n after the PREPARE is complete.

FROM

Introduces the statement string. The statement string is the value of the specified *string-expression* or the identified *host-variable*.

string-expression

A *string-expression* is any PL/I *string-expression* that yields a character string. SQL expressions that yield a character string are not allowed. A *string-expression* is only allowed in PL/I.

host-variable

Identifies a *host-variable* that is declared in the program in accordance with the rules for declaring character-string or UCS-2 graphic host variables. The host variable must not have a CLOB or DBCLOB data type, and an indicator variable must not be specified.

The statement string must be one of the following SQL statements:

	ALTER	GRANT	SAVEPOINT
	CALL	HOLD LOCATOR	select-statement
	COMMENT	INSERT	SET PATH
	COMMIT	LABEL	SET SCHEMA
	CREATE	LOCK TABLE	SET TRANSACTION
	DECLARE GLOBAL TEMPORARY TABLE	RELEASE SAVEPOINT	UPDATE
	DELETE	RENAME	VALUES INTO
	DROP	REVOKE	
	FREE LOCATOR	ROLLBACK	

PREPARE

The statement string must not:

- Begin with EXEC SQL and end with END-EXEC or a semicolon (;).
- Include references to host variables.

Notes

Parameter Markers: Although a statement string cannot include references to host variables, it may include *parameter markers*. These can be replaced by the values of host variables when the prepared statement is executed. A parameter marker is a question mark (?) that is used where a host variable could be used if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see “OPEN” on page 686 and “EXECUTE” on page 640.

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). For more information, refer to “CAST Specification” on page 143.

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 parameter 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 58. The locations are grouped in this table into expressions, predicates and functions to assist in determining applicability of an untyped parameter marker.

Table 58. Untyped Parameter Marker Usage

Untyped Parameter Marker Location	Data Type
Expressions (including select list, CASE, and VALUES)	
Alone in a select list that is not in a subquery	Error
Alone in a select list that is in an EXISTS subquery	Error

Table 58. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
Alone in a select list that is in a subquery	The data type of the other operand of the subquery. ⁶⁴
Alone in a select list that is in a select-statement of an INSERT statement	The data type of the associated column of the target table. ⁶⁴
Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules. Includes cases such as: $? + ? + 10$	Error
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
Any operands of a CONCAT operator	Error
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. ⁶⁴
The expression following the CASE keyword 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 93 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" on page 93 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.
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. ⁶⁴
As a value on the right side of a SET special register statement	The data type of the special register.

PREPARE

Table 58. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
As a value in the INTO clause of the VALUES INTO statement	The data type of the associated expression. ⁶⁴
As a value in a FREE LOCATOR or HOLD LOCATOR statement	Locator.
Predicates	
Both operands of a comparison operator	Error
One operand of a comparison operator where the other operand is other than an untyped parameter marker or a distinct type.	The data type of the other operand. ⁶⁴
One operand of a comparison operator where the other operand is a distinct type.	Error
All operands of a BETWEEN predicate	Error
Two operands of a BETWEEN predicate (either the first and second, or the first and third)	Same as that of the only non-parameter marker.
Only one operand of a BETWEEN predicate	Result of applying the "Rules for Result Data Types" on page 93 on all operands that are other than untyped parameter markers, except the CCSID attribute is the CCSID of the value specified at execution time.
All operands of an IN predicate, for example, ? IN (?,?,?)	Error
The first operand of an IN predicate where the right hand side is a subselect, for example, ? IN (subselect).	Data type of the selected column
The first operand of an IN predicate where the right hand side is not a subselect, for example, ? IN (?,A,B) or for example, ? IN (A,?,B,?).	Result of applying the "Rules for Result Data Types" on page 93 on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers, except the CCSID attribute is the CCSID of the value specified at execution time.
Any or all operands of the IN list of the IN predicate, for example, A IN (?,B,?).	Result of applying the "Rules for Result Data Types" on page 93 on all operands of the IN predicate (operands to the left and right of the IN predicate) that are other than untyped parameter markers, except the CCSID attribute is the CCSID of the value specified at execution time.
All three operands of the LIKE predicate.	Error
The match expression of the LIKE predicate.	Error
The pattern expression of the LIKE predicate.	Either VARCHAR(32740) or VARGRAPHIC(16370) or BLOB(32740) depending on the data type of the match expression. For information about using fixed-length host variables for the value of the pattern see page "LIKE Predicate" on page 155.

Table 58. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
The escape expression of the LIKE predicate.	Either VARCHAR(1) or VARGRAPHIC(1) or BLOB(1) depending on the data type of the match expression.
Operand of the NULL predicate	Error
Functions	
All operands of COALESCE, IFNULL, LAND, LOR, MIN, MAX, NULLIF, VALUE, or XOR	Error
Any operand of COALESCE, IFNULL, LAND, LOR, MIN, MAX, NULLIF, VALUE, or XOR where at least one operand is other than an untyped parameter marker.	Result of applying the “Rules for Result Data Types” on page 93 on all operands that are other than untyped parameter markers.
The first operand of POSITION or the second operand of POSSTR.	Either VARCHAR(32740) or VARGRAPHIC(16370) or BLOB(32740) depending on the data type of the other operand.
All other operands of all other scalar functions including user-defined functions.	Error
Operand of a column function	Error

Error Checking: When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, a prepared statement is not created and the error condition that prevents its creation is reported in the SQLCA.

In local and remote processing, the DLYPREP(*YES) option can cause some SQL statements to receive “delayed” errors. For example, DESCRIBE, EXECUTE, and OPEN might receive an SQLCODE that normally occurs during PREPARE processing.

Reference and Execution Rules: Prepared statements can be referred to in the following kinds of statements, with the following restrictions shown:

Statement	The prepared statement restrictions
DESCRIBE	None
DECLARE CURSOR	Must be SELECT when the cursor is opened
EXECUTE	Must not be SELECT

A prepared statement can be executed many times. If a prepared statement is not executed more than once and does not contain parameter markers, it is more efficient to use the EXECUTE IMMEDIATE statement rather than the PREPARE and EXECUTE statements.

Prepared Statement Persistence: All prepared statements are destroyed when:⁶⁵

- A CONNECT (Type 1) statement is executed.
- A DISCONNECT statement disconnects the connection with which the prepared statement is associated.

64. If the data type is DATE, TIME, or TIMESTAMP, then VARCHAR(32740) is used.

65. Prepared statements may be cached and not actually destroyed. However, a cached statement can only be used if the same statement is prepared again.

PREPARE

- A prepared statement is associated with a release-pending connection and a successful commit occurs.
- The associated scope (job, activation group, or program) of the SQL statement ends.

Scope of a Statement: The scope of *statement-name* is the source program in which it is defined. You can only reference a prepared statement by other SQL statements that are precompiled with the PREPARE statement. For example, a program called from another separately compiled program cannot use a prepared statement that was created by the calling program.

The scope of *statement-name* is also limited to the thread in which the program that contains the statement is running. For example, if the same program is running in two separate threads in the same job, the second thread cannot use a statement that was prepared by the first thread.

Although the scope of a statement is the program in which it is defined, each package created from the program includes a separate instance of the prepared statement and more than one prepared statement can exist at run time. For example, assume a program using CONNECT (Type 2) statements connects to location X and location Y in the following sequence:

```
EXEC SQL CONNECT TO X;
EXEC SQL PREPARE S FROM :hv1;
EXEC SQL EXECUTE S;
.
.
.
EXEC SQL CONNECT TO Y;
EXEC SQL PREPARE S FROM :hv1;
EXEC SQL EXECUTE S;
```

The second prepare of S prepares another instance of S at Y.

A prepared statement can only be referenced in the same instance of the program in the program stack, unless CLOSQLCSR(*ENDJOB), CLOSQLCSR(*ENDACTGRP), or CLOSQLCSR(*ENDSQL) is specified on the CRTSQLxxx commands.

- If CLOSQLCSR(*ENDJOB) is specified, the prepared statement can be referred to by any instance of the program (that prepared the statement) on the program stack. In this case, the prepared statement is destroyed at the end of the job.
- If CLOSQLCSR(*ENDSQL) is specified, the prepared statement can be referred to by any instance of the program (that prepared the statement) on the program stack until the last SQL program on the program stack ends. In this case, the prepared statement is destroyed when the last SQL program on the program stack ends.
- If CLOSQLCSR(*ENDACTGRP) is specified, the prepared statement can be referred to by all instances of the module in the program that prepared the statement until the activation group ends. In this case, the prepared statement is destroyed when the activation group ends.

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.

PREPARE

```
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 assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL PREPARE STMT_NAME FROM :HOLDER END-EXEC.
```

```
EXEC SQL EXECUTE STMT_NAME USING DESCRIPTOR :INSERT_DA END-EXEC.
```

Assume that the following statement is to be prepared:

```
INSERT INTO DEPARTMENT VALUES(?, ?, ?, ?)
```

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 executing the EXECUTE statement.

SQLDAID		
SQLDABC	336	
SQLN	4	
SQLD	4	
SQLTYPE	452	
SQLLEN	3	
SQLDATA		→ G01
SQLIND		
SQLNAME		
SQLTYPE	448	
SQLLEN	29	
SQLDATA		→ COMPLAINTS
SQLIND		
SQLNAME		
SQLTYPE	453	
SQLLEN	6	
SQLDATA		
SQLIND		→ 1
SQLNAME		
SQLTYPE	452	
SQLLEN	3	
SQLDATA		→ A00
SQLIND		
SQLNAME		

RBAL3501-0

RELEASE (Connection)

RELEASE (Connection)

The RELEASE statement places one or more connections in the release-pending state.

Invocation

This statement can only be embedded within an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

RELEASE is not allowed in a trigger. RELEASE is not allowed in an external procedure if the external procedure is called on a remote server.

Authorization

None required.

Syntax



Description

server-name or *host-variable*

Identifies a connection by the specified server name or the server name contained in the host variable. If a host variable is specified:

- It must be a character-string variable.
- It must not be followed by an indicator variable.
- The server name must be left-justified within the host variable and must conform to the rules for forming an ordinary identifier.
- If the length of the server name is less than the length of the host variable, it must be padded on the right with blanks.

When the RELEASE statement is executed, the specified server name or the server name contained in the host variable must identify an existing connection of the activation group.

CURRENT

Identifies the current connection of the activation group. The activation group must be in the connected state.

ALL or ALL SQL

Identifies all existing connections of the activation group (local as well as remote connections).

An error or warning does not occur if no connections exist when the statement is executed.

If the RELEASE statement is successful, each identified connection is placed in the release-pending state and will therefore be ended during the next commit

RELEASE (Connection)

operation. If the RELEASE statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Notes

RELEASE and CONNECT (Type 1): Using CONNECT (Type 1) semantics does not prevent using RELEASE.

Scope of RELEASE: RELEASE does not close cursors, does not release any resources, and does not prevent further use of the connection.

Resource Considerations for Remote Connections: Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be in the release-pending state and one that is going to be reused should not be in the release-pending state.

Connection States: ROLLBACK does not reset the state of a connection from release-pending to held.

If the current connection is in the release-pending state when a commit operation is performed, the connection is ended and the activation group is in the unconnected state. In this case, the next executed SQL statement must be CONNECT or SET CONNECTION.

RELEASE ALL places the connection to the local server in the release-pending state. A connection in the release-pending state is ended during a commit operation even though it has an open cursor defined with the WITH HOLD clause.

Examples

Example 1: The connection to TOROLAB1 is not needed in the next unit of work. The following statement will cause it to be ended during the next commit operation.

```
EXEC SQL RELEASE TOROLAB1;
```

Example 2: The current connection is not needed in the next unit of work. The following statement will cause it to be ended during the next commit operation.

```
EXEC SQL RELEASE CURRENT;
```

Example 3: None of the existing connections are needed in the next unit of work. The following statement will cause it to be ended during the next commit operation.

```
EXEC SQL RELEASE ALL;
```

RELEASE SAVEPOINT

The RELEASE SAVEPOINT statement releases the identified savepoint and any subsequently established savepoints within a unit of work.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

```

▶▶—RELEASE—TO—SAVEPOINT—savepoint-name—▶▶

```

Description

savepoint-name

Identifies a savepoint to release. If the named savepoint does not exist, an error occurs. The named savepoint and all the savepoints that were subsequently established in the unit of work are released. After a savepoint is released, it is no longer maintained and rollback to the savepoint is no longer possible.

Note

Savepoint Names: The name of the savepoint that was released can be re-used in another SAVEPOINT statement, regardless of whether the UNIQUE keyword was specified on an earlier SAVEPOINT statement specifying this same savepoint name.

Isolation Level Restriction: A RELEASE SAVEPOINT statement is not allowed if commitment control is not active for the activation group. For information on determining which commitment definition is used, see “Notes” on page 413.

Example

Assume that a main routine sets savepoint A and then invokes a subroutine that sets savepoints B and C. When control returns to the main routine, release savepoint A and any subsequently set savepoints. Savepoints B and C, which were set by the subroutine, are released in addition to A.

```
RELEASE SAVEPOINT A
```

RENAME

The RENAME statement renames a table, view, or index. The name and/or the system object name of the table, view, or index can be changed.

Invocation

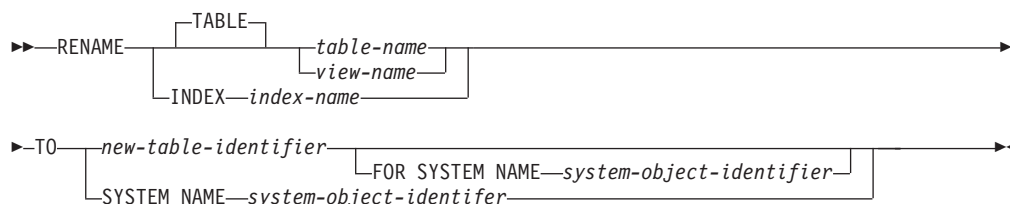
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The following system authorities:
 - If the name of the object is changed:
 - The system authority of *OBJMGT on the table, view, or index to be renamed
 - The system authority *EXECUTE on the library containing the table, view, or index to be renamed
 - If the system name of the object is changed:
 - The system authority of *OBJMGT on the table, view, or index to be renamed
 - The system authorities *EXECUTE and *UPD on the library containing the table, view, or index to be renamed
- Administrative authority

Syntax



Description

TABLE *table-name* or *view-name*

Identifies the table or view that will be renamed. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a catalog table or a global temporary table. The specified name can be an alias name. The specified table or view is renamed to the new name. All privileges, constraints, indexes, triggers, views, and logical files on the table or view are preserved.

Any access plans that reference the table or view are implicitly prepared again when a program that uses the access plan is next run. Since the program refers to a table or view with the original name, if a table or view with the original name does not exist at that time, a negative value will be returned in the SQLCODE field of the SQLCA.

RENAME

INDEX *index-name*

Identifies the index that will be renamed. The *index-name* must identify an index that exists at the current server. The specified index is renamed to the new name.

Any access plans that reference the index are not affected by rename.

new-table-identifier

Identifies the new *table-name*, *view-name*, or *index-name* of the table, view, or index, respectively. *new-table-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *new-table-identifier* must be an unqualified SQL identifier.

SYSTEM NAME *system-object-identifier*

Identifies the new *system-object-identifier* of the table, view, or index, respectively. *system-object-identifier* must not be the same as a table, view, alias, or index that already exists at the current server. The *system-object-identifier* must be an unqualified system identifier.

If the name of the object and the system name of the object are the same and *new-table-identifier* is not specified, specifying *system-object-identifier* will be the new name and system object name. Otherwise, specifying *system-object-identifier* will only affect the system name of the object and not affect the name of the object.

If both *new-table-identifier* and *system-object-identifier* are specified, they cannot both be valid system object names.

Notes

Effects of the Statement: The specified table is renamed to the new name. All privileges, constraints, and indexes on the table are preserved.

Any access plans that refer to that table are invalidated. For more information see “Packages and Access Plans” on page 13.

Considerations for Aliases: If an alias name is specified for *table-name*, the table must exist at the current server, and the table that is identified by the alias is renamed. The name of the alias is not changed and continues to refer to the old table name after the rename.

There is no support for changing the name of an alias with the RENAME statement. To change the name to which the alias refers, the alias must be dropped and recreated.

Rename Rules: The rename operation performed depends on the new name specified.

- If the new name is a valid system identifier,
 - the alternative name (if any) is removed, and
 - the system object name is changed to the new name.
- If the new name is not a valid system identifier,
 - the alternative name is added or changed to the new name, and
 - a new system object name is generated if the system object name (of the table or view) was specified as the table, view, or index to rename. For more information about generated table name rules, see “Rules for Table Name Generation” on page 553.

| If an alias name is specified for *table-name*, the alias must exist at the current server,
| and the table that is identified by the alias is renamed. The name of the alias is not
| changed and continues to refer to the old table after the rename. There is no
| support for changing the name of an alias.

Examples

Example 1: Rename a table named MY_IN_TRAY to MY_IN_TRAY_94. The system object name will remain unchanged (MY_IN_TRAY).

```
RENAME TABLE MY_IN_TRAY TO MY_IN_TRAY_94  
FOR SYSTEM NAME MY_IN_TRAY
```

Example 2: Rename a table named MA_PROJ to MA_PROJ_94.

```
RENAME TABLE MA_PROJ  
TO SYSTEM NAME MA_PROJ_94
```

REVOKE (Distinct Type Privileges)

This form of the REVOKE statement removes the privileges on a distinct type.

Invocation

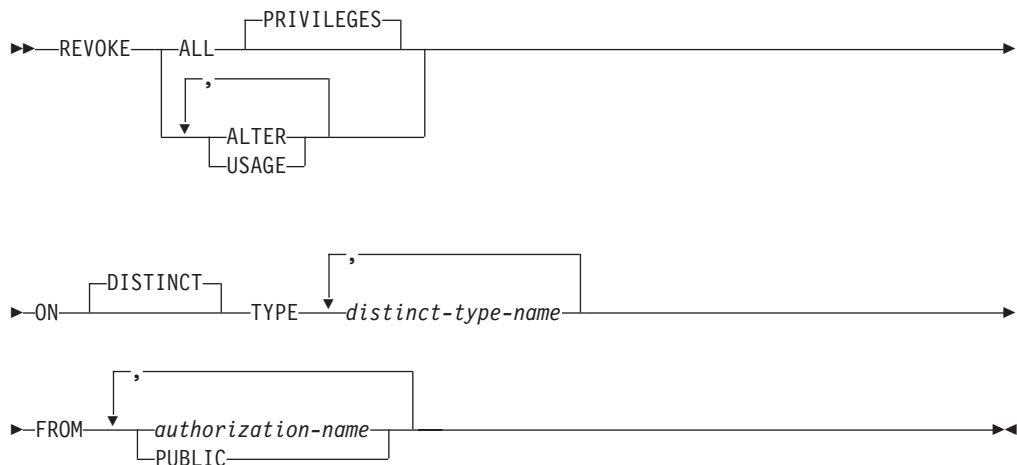
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each distinct type identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the distinct type
 - The system authority *EXECUTE on the library containing the distinct type
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Revokes one or more distinct type privileges from each *authorization-name*. The privileges revoked are those privileges on the identified distinct types that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a distinct type is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER

Revokes the privilege to use the COMMENT statement.

USAGE

Revokes the privilege to use distinct types in tables, functions, procedures, or as the source type in a CREATE DISTINCT TYPE statement.

REVOKE (Distinct Type Privileges)

ON DISTINCT TYPE *distinct-type-name*

Identifies the distinct types from which you are revoking privileges. The *distinct-type-name* must identify a distinct type that exists at the current server.

FROM

Identifies from whom the privileges are revoked.

authorization-name,...

Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC

Revokes the specified privileges from PUBLIC.

Notes

Multiple grants: If authorization ID A granted the same privilege to authorization ID B more than once, revoking that privilege from B nullifies all those grants.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding System Authorities: When a distinct type privilege is revoked, the corresponding system authorities are revoked. For information on the system authorities that correspond to SQL privileges see “GRANT (Distinct Type Privileges)” on page 652.

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword DATA can be used as a synonym for DISTINCT.

Example

Revoke the USAGE privilege on distinct type SHOESIZE from user JONES.

```
REVOKE USAGE
ON DISTINCT TYPE SHOESIZE
FROM JONES
```

REVOKE (Function or Procedure Privileges)

REVOKE (Function or Procedure Privileges)

This form of the REVOKE statement removes the privileges on a function or procedure.

Invocation

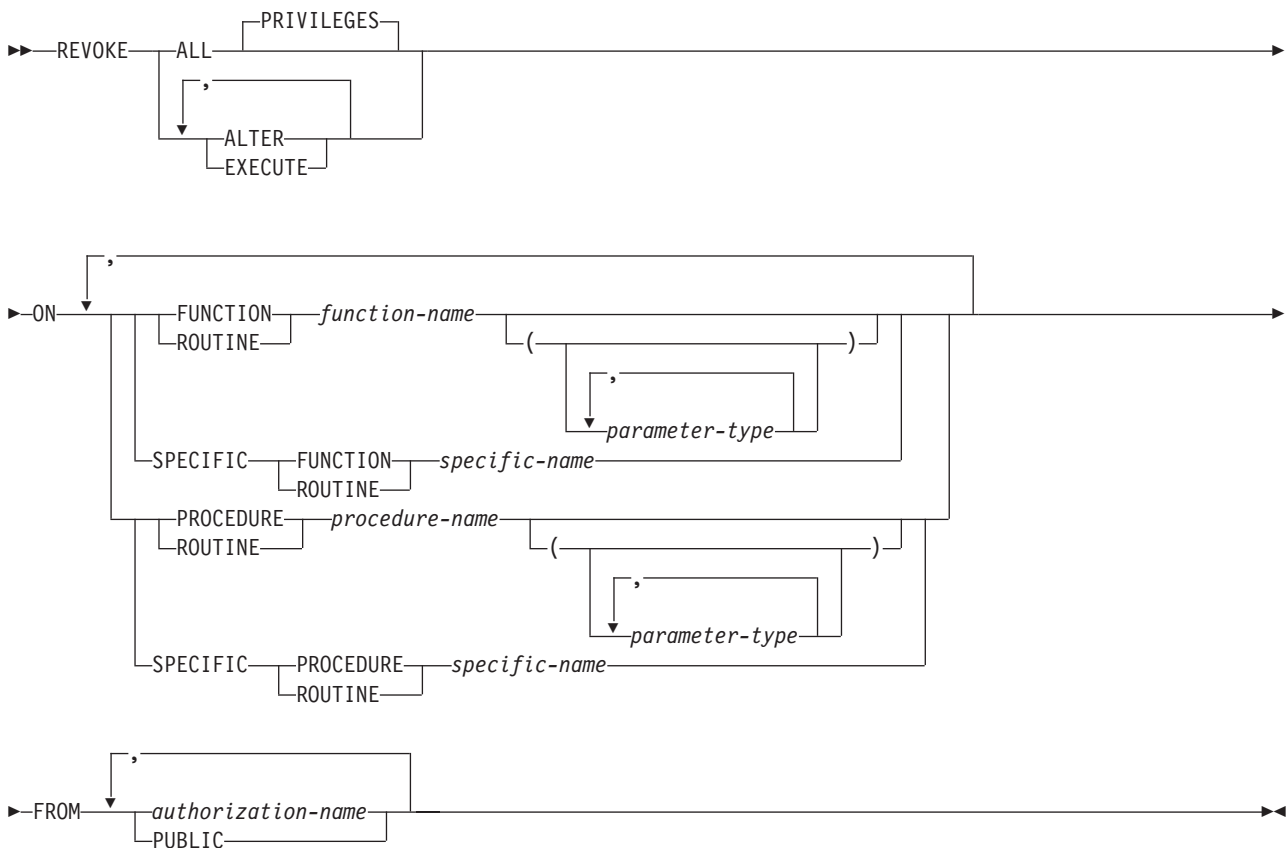
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each function or procedure identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the function or procedure
 - The system authority *EXECUTE on the library (or directory if this is a Java routine) containing the function or procedure
- Administrative authority

Syntax



REVOKE (Function or Procedure Privileges)

parameter-type:

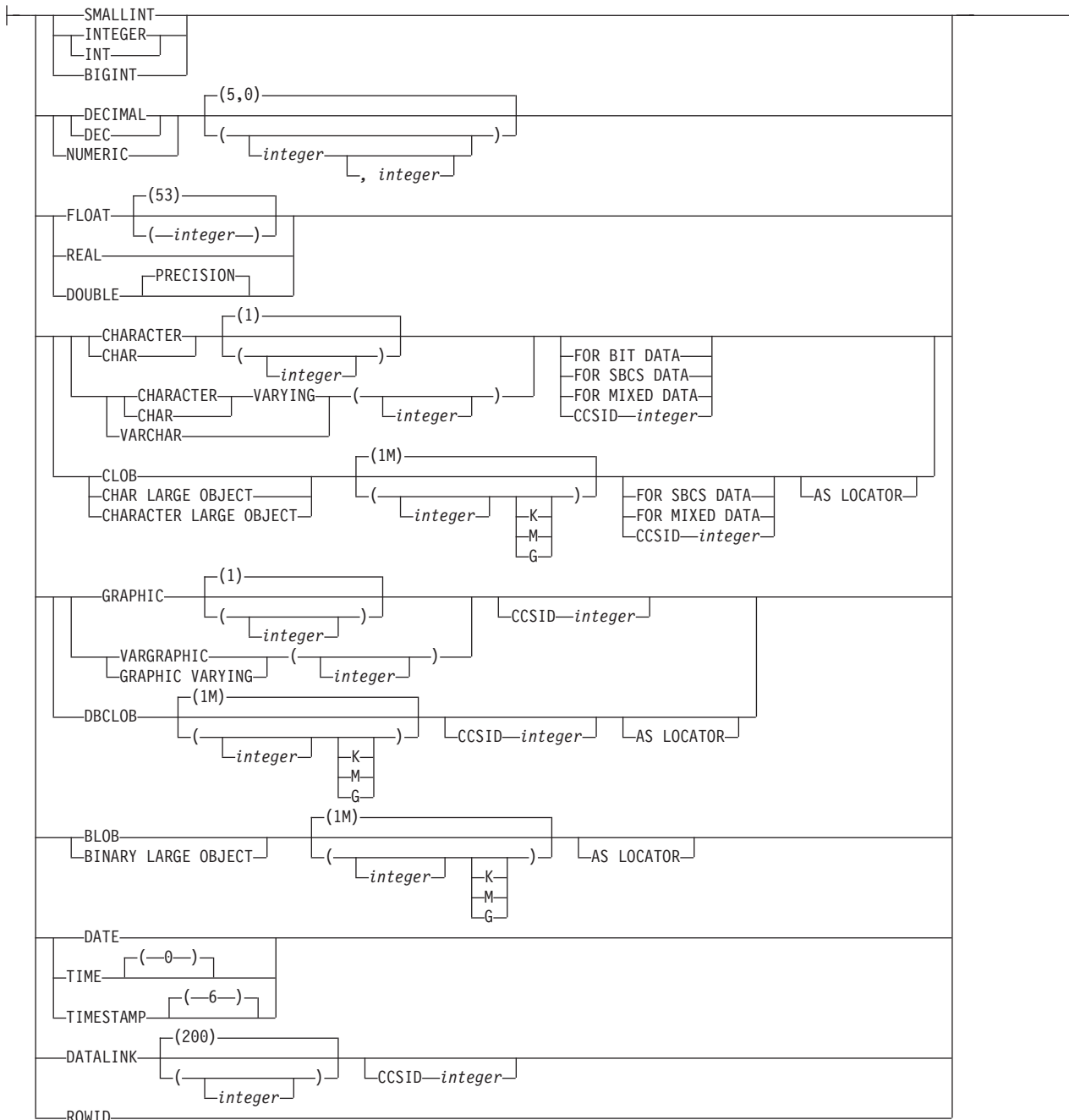
|—*data-type*—|
| AS LOCATOR |

data-type:

|—*built-in-type*—|
| *distinct-type-name* |

REVOKE (Function or Procedure Privileges)

built-in-type:



Description

ALL or ALL PRIVILEGES

Revokes one or more function or procedure privileges from each *authorization-name*. The privileges revoked are those privileges on the identified functions or procedures that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a function or procedure is not the same as revoking the system authority of *ALL.

REVOKE (Function or Procedure Privileges)

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER

Revokes the privilege to use the COMMENT statement.

EXECUTE

Revokes the privilege to execute a function or procedure.

FUNCTION or SPECIFIC FUNCTION

Identifies the function from which the privilege is revoked. The function must exist at the current server and it must be a user-defined function. The function can be identified by its name, function signature, or specific name.

FUNCTION *function-name*

Identifies the function by its name. The *function-name* must identify exactly one function. The function may have any number of parameters defined for it. If there is more than one function of the specified name in the specified or implicit schema, an error is returned.

FUNCTION *function-name (parameter-type, ...)*

Identifies the function by its function signature, which uniquely identifies the function. The *function-name (parameter-type, ...)* must identify a function with the specified function signature. The specified parameters must match the data types in the corresponding position that were specified when the function was created. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance on which the privilege is to be revoked. Synonyms for data types are considered a match.

If *function-name ()* is specified, the function identified must have zero parameters.

function-name

Identifies the name of the function.

(parameter-type, ...)

Identifies the parameters of the function.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a function defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type

REVOKE (Function or Procedure Privileges)

are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE FUNCTION statement.

AS LOCATOR

Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*

Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE or SPECIFIC PROCEDURE

Identifies the procedure from which the privilege is revoked. The *procedure-name* must identify a procedure that exists at the current server.

PROCEDURE *procedure-name*

Identifies the procedure by its name. The *procedure-name* must identify exactly one procedure. The procedure may have any number of parameters defined for it. If there is more than one procedure of the specified name in the specified or implicit schema, an error is returned.

PROCEDURE *procedure-name (parameter-type, ...)*

Identifies the procedure by its procedure signature, which uniquely identifies the procedure. The *procedure-name (parameter-type, ...)* must identify a procedure with the specified procedure signature. The specified parameters must match the data types in the corresponding position that were specified when the procedure was created. 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 revoked. Synonyms for data types are considered a match.

If *procedure-name ()* is specified, the procedure identified must have zero parameters.

procedure-name

Identifies the name of the procedure.

(parameter-type, ...)

Identifies the parameters of the procedure.

If an unqualified distinct type name is specified, the database manager searches the SQL path to resolve the schema name for the distinct type.

For data types that have a length, precision, or scale attribute, use one of the following:

- Empty parentheses indicate that the database manager ignores the attribute when determining whether the data types match. For example, DEC() will be considered a match for a parameter of a procedure defined with a data type of DEC(7,2). However, FLOAT cannot be specified with empty parenthesis because its parameter value indicates a specific data type (REAL or DOUBLE).

REVOKE (Function or Procedure Privileges)

- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE PROCEDURE statement.

Specifying the FOR DATA clause or CCSID clause is optional. Omission of either clause indicates that the database manager ignores the attribute when determining whether the data types match. If either clause is specified, it must match the value that was implicitly or explicitly specified in the CREATE PROCEDURE statement.

AS LOCATOR

Specifies that the procedure is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC PROCEDURE *specific-name*

Identifies the procedure by its specific name. The *specific-name* must identify a specific procedure that exists at the current server.

FROM

Identifies from whom the privileges are revoked.

authorization-name,...

Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC

Revokes the specified privileges from PUBLIC.

Notes

Multiple Grants: If you revoke a privilege on a function or procedure, it nullifies any grant of the privilege on that function or procedure, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding System Authorities: When a function or procedure privilege is revoked, the corresponding system authorities are revoked. For information on the system authorities that correspond to SQL privileges see “GRANT (Function or Procedure Privileges)” on page 655.

Privileges revoked from either an SQL or external function or procedure are revoked from its associated program (*PGM) or service program (*SRVPGM) object.

REVOKE (Function or Procedure Privileges)

| **Keyword Synonym:** The following keywords are synonyms supported for
| compatibility to prior releases. These keywords are non-standard and should not
| be used:

- The keyword RUN can be used as a synonym for EXECUTE.

Example

Revoke the EXECUTE privilege on procedure PROCA from PUBLIC.

```
REVOKE EXECUTE  
ON PROCEDURE PROCA  
FROM PUBLIC
```

REVOKE (Package Privileges)

This form of the REVOKE statement removes the privileges on a package.

Invocation

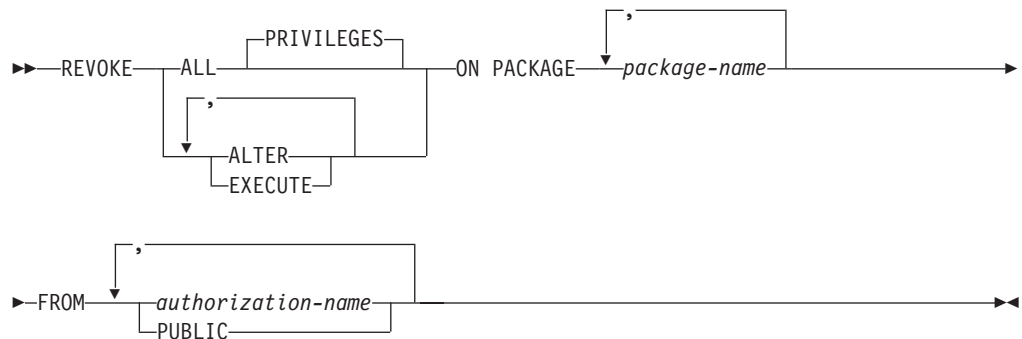
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each package identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the package
 - The system authority *EXECUTE on the library containing the package
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Revokes one or more package privileges from each *authorization-name*. The privileges revoked are those privileges on the identified packages that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a package is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described.

ALTER

Revokes the privilege to use the COMMENT and LABEL statements.

EXECUTE

Revokes the privilege to execute statements in a package.

ON PACKAGE *package-name*

Identifies the packages from which you are revoking privileges. The *package-name* must identify a package that exists at the current server.

FROM

Identifies from whom the privileges are revoked.

REVOKE (Package Privileges)

authorization-name,...

Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC

Revokes the specified privileges from PUBLIC.

Notes

Multiple Grants: If you revoke a privilege on a package, it nullifies any grant of the privilege on that package, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

Corresponding System Authorities: When a package privilege is revoked, the corresponding system authorities are revoked. For information on the system authorities that correspond to SQL privileges see "GRANT (Package Privileges)" on page 662.

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword RUN can be used as a synonym for EXECUTE.
- The keyword PROGRAM can be used as a synonym for PACKAGE.

Example

Example 1: Revoke the EXECUTE privilege on package PKGA from PUBLIC.

```
REVOKE EXECUTE
ON PACKAGE PKGA
FROM PUBLIC
```

Example 2: Revoke the EXECUTE privilege on package RRSP_PKG from user FRANK and PUBLIC.

```
REVOKE EXECUTE
ON PACKAGE RRSP_PKG
FROM FRANK, PUBLIC
```


REVOKE (Table or View Privileges)

This form of the REVOKE statement removes privileges on a table or view.

Invocation

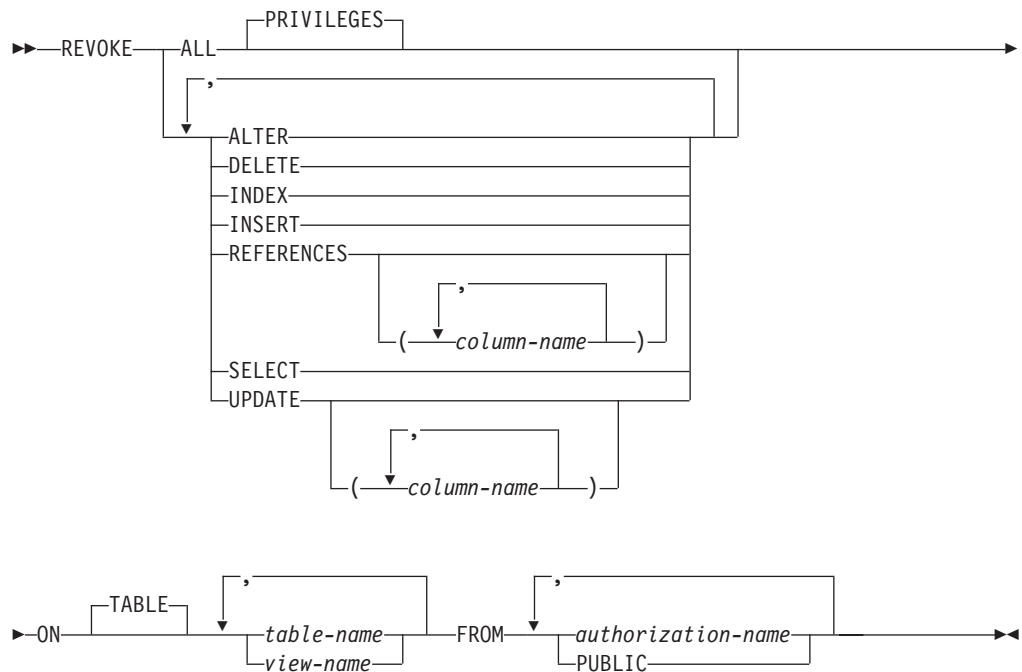
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement:
 - Every privilege specified in the statement
 - The system authority of *OBJMGT on the table or view
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

Syntax



Description

ALL or ALL PRIVILEGES

Revokes one or more privileges from each *authorization-name*. The privileges revoked are those privileges on the identified tables and views that were granted to the *authorization-names*. Note that revoking ALL PRIVILEGES on a table or view is not the same as revoking the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described, but only as it applies to the tables and views named in the ON clause.

REVOKE (Table or View Privileges)

ALTER

Revokes the privilege to use the ALTER TABLE statement on tables. Revokes the privilege to use the COMMENT and LABEL statements on tables and views.

DELETE

Revokes the privilege to use the DELETE statement.

INDEX

Revokes the privilege to use the CREATE INDEX statement.

INSERT

Revokes the privilege to use the INSERT statement.

REFERENCES

Revokes the privilege to add a referential constraint in which the table is a parent.

REFERENCES (*column-name,...*)

Revokes the privilege to add a referential constraint using the specified column(s) in the parent key. Each column name must be an unqualified name that identifies a column in each table identified in the ON clause.

SELECT

Revokes the privilege to use the SELECT or CREATE VIEW statement.

UPDATE

Revokes the privilege to use the UPDATE statement.

UPDATE (*column-name,...*)

Revokes the privilege to update the specified columns. Each column name must be an unqualified name that identifies a column in each table identified in the ON clause.

ON *table-name* or *view-name*, ...

Identifies the table or view on which you are revoking the privileges. The *table-name* or *view-name* must identify a table or view that exists at the current server, but must not identify a global temporary table.

FROM

Identifies from whom the privileges are revoked.

authorization-name,...

Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC

Revokes the specified privileges from PUBLIC.

Notes

Multiple Grants: If the same privilege is granted to the same user more than once, revoking that privilege from that user nullifies all those grants.

If you revoke a privilege, it nullifies any grant of that privilege, regardless of who granted it.

Revoking WITH GRANT OPTION: The only way to revoke the WITH GRANT OPTION is to revoke ALL.

REVOKE (Table Privileges)

Privilege warning: Revoking a specific privilege from a user does not necessarily prevent that user from performing an action that requires that privilege. For example, the user may still have the privilege through PUBLIC or administrative privileges.

If more than one system authority will be revoked with an SQL privilege, and any one of the authorities cannot be revoked, then a warning occurs and no authorities will be revoked for that privilege.

Corresponding System Authorities: When a table privilege is revoked, the corresponding system authorities are revoked, except:

- When revoking authorities to a table or view, *OBJOPR is revoked only when *ADD, *DLT, *READ, and *UPD have all been revoked.
- When revoking authorities to a view, authorities will not be revoked from any tables or views referenced in the subselect of the view definition.

For information on the system authorities that correspond to SQL privileges see “GRANT (Table or View Privileges)” on page 664.

Revoking either the INDEX or ALTER privilege, revokes the system authority *OBJALTER.

Examples

Example 1: Revoke SELECT privileges on table EMPLOYEE from user ENGLES.

```
REVOKE SELECT
ON TABLE EMPLOYEE
FROM ENGLES
```

Example 2: Revoke update privileges on table EMPLOYEE previously granted to all users. Note that grants to specific users are not affected.

```
REVOKE UPDATE
ON TABLE EMPLOYEE
FROM PUBLIC
```

Example 3: Revoke all privileges on table EMPLOYEE from users PELLOW and ANDERSON.

```
REVOKE ALL
ON TABLE EMPLOYEE
FROM PELLOW, ANDERSON
```

Example 4: Revoke the privilege to update column_1 in VIEW1 from FRED.

```
REVOKE UPDATE(column_1)
ON VIEW1
FROM FRED
```

ROLLBACK

ROLLBACK

The ROLLBACK statement can be used to either:

- End a unit of work and back out all the relational database changes that were made by that unit of work. If relational databases are the only recoverable resources used by the application process, ROLLBACK also ends the unit of work.
- Back out only the changes made after a savepoint was set within the unit of work without ending the unit of work. Rolling back to a savepoint enables selected changes to be undone.

Invocation

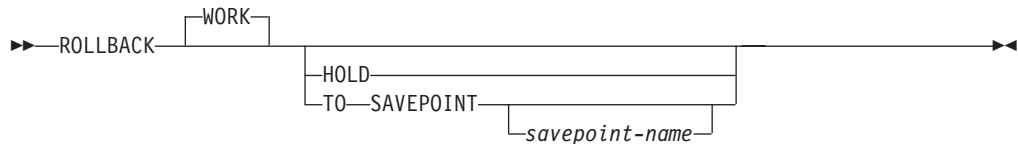
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

ROLLBACK is not allowed in a trigger if the trigger program and the triggering program are run under the same commitment definition. ROLLBACK is not allowed in an external procedure if the external procedure and the program that issued the CALL statement run under the same commitment definition.

Authorization

None required.

Syntax



Description

When ROLLBACK is used without the SAVEPOINT clause, the unit of work in which it is executed is ended and a new unit of work is started. All changes made by ALTER, CALL, COMMENT, CREATE, DECLARE GLOBAL TEMPORARY TABLE, DELETE, DROP (except for DROP SCHEMA), GRANT, INSERT, LABEL, RENAME, REVOKE, and UPDATE statements executed 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:

- CONNECT
- DISCONNECT
- RELEASE CONNECTION
- SET CONNECTION
- SET PATH
- SET SCHEMA

The impact of ROLLBACK or ROLLBACK TO SAVEPOINT on the contents of a declared global temporary table is determined by the setting of the ON ROLLBACK clause of the DECLARE GLOBAL TEMPORARY TABLE statement.

WORK

ROLLBACK WORK has the same effect as ROLLBACK.

HOLD

Specifies a hold on resources. If specified, currently open cursors are not closed and all resources acquired during the unit of work, except locks on the rows of tables, are held. Locks on specific rows implicitly acquired during the unit of work, however, are released.

If HOLD is omitted, ROLLBACK without the TO SAVEPOINT clause also causes the following to occur under this unit of work's commitment definition:

- Cursors opened under this unit of work's commitment definition are closed.
- Table locks acquired by the LOCK TABLE statement under this unit of work's commitment definition are released.
- All LOB locators, including those that are held, are freed.

At the end of a ROLLBACK HOLD, the cursor position is the same as it was at the start of the unit of work, unless ALWBLK(*ALLREAD) was specified when the program or routine that contains the cursor was created

TO SAVEPOINT

Specifies that the unit of work is not to be ended and that only a partial rollback (to a savepoint) is to be performed. If a savepoint name is not specified, rollback is to the last active savepoint. For example, if in a unit of work, savepoints A, B, and C are set in that order and then C is released, ROLLBACK TO SAVEPOINT causes a rollback to savepoint B.

savepoint-name

Identifies the savepoint to which to roll back. If the named savepoint does not exist, an error occurs.

After a successful ROLLBACK TO SAVEPOINT, the savepoint continues to exist.

All database changes (including changes made to declared temporary tables) that were made after the savepoint was set are backed out. All locks and LOB locators are retained.

The impact on cursors resulting from a ROLLBACK TO SAVEPOINT depends on the statements within the savepoint:

- If the savepoint contains DDL on which a cursor is dependent, the cursor is closed. Attempts to use such a cursor after a ROLLBACK TO SAVEPOINT results in an error.
- Otherwise, the cursor is not affected by the ROLLBACK TO SAVEPOINT (it remains open and positioned).

Any savepoints that are set after the one to which rollback is performed are released. The savepoint to which rollback is performed is not released.

Notes

Recommended coding practices: An explicit COMMIT or ROLLBACK statement should be coded at the end of an application process. Either an implicit commit or rollback operation will be performed at the end of an application process

ROLLBACK

depending on the application environment. Thus, a portable application should explicitly execute a COMMIT or ROLLBACK before execution ends in those environments where explicit COMMIT or ROLLBACK is permitted.

Implicit ROLLBACK: The ending of the default activation group causes an implicit rollback. Thus, an explicit COMMIT or ROLLBACK statement should be issued before the end of the default activation group.

A ROLLBACK is automatically performed when:

1. The default activation group ends without a final COMMIT being issued.]
2. A failure occurs that prevents the activation group from completing its work (for example, a power failure).

If the unit of work is in the prepared state because a COMMIT was in progress when the failure occurred, a rollback is not performed. Instead, resynchronization of all the connections involved in the unit of work will occur. For more information, see the Commitment control topic.

3. A failure occurs that causes a loss of the connection to a server (for example, a communications line failure).

If the unit of work is in the prepared state because a COMMIT was in progress when the failure occurred, a rollback is not performed. Instead, resynchronization of all the connections involved in the unit of work will occur. For more information, see the Commitment control topic.

4. A nondefault activation group ends abnormally.

Row Lock Limit: A unit of work may include the processing of up to and including 4 million rows, including rows retrieved during a SELECT INTO or FETCH statement⁶⁶, and rows inserted, deleted, or updated as part of INSERT, DELETE, and UPDATE operations.⁶⁷

Unaffected Statements: The commit and rollback operations do not affect the DROP SCHEMA statement, and this statement is not, therefore, allowed in an application program that also specifies COMMIT(*CHG), COMMIT(*CS), COMMIT(*ALL), or COMMIT(*RR).

ROLLBACK Restrictions: A ROLLBACK statement is not allowed if commitment control is not active for the activation group. For information on determining which commitment definition is used, see the commitment definition discussion in the COMMIT statement.

Any cursors associated with a prepared statement that is destroyed cannot be opened until the statement is prepared again. ROLLBACK has no effect on the state of connections.

If, within a unit of work, a CLOSE is followed by a ROLLBACK, all changes made within the unit of work are backed out. The CLOSE itself is not backed out and the file is not reopened.

66. Unless you specified COMMIT(*CHG) or COMMIT(*CS), in which case these rows are not included in this total.

67. This limit also includes:

- Any rows accessed or changed through files opened under commitment control through high-level language file processing
- Any rows deleted, updated, or inserted as a result of a trigger or CASCADE, SET NULL, or SET DEFAULT referential integrity delete rule.

Examples

Example 1: See the examples under COMMIT on page 413 for examples using the ROLLBACK statement.

Example 2: After a unit of recovery started, assume that three savepoints A, B, and C were set and that C was released:

```
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;  
...  
SAVEPOINT B ON ROLLBACK RETAIN CURSORS;  
...  
SAVEPOINT C ON ROLLBACK RETAIN CURSORS;  
...  
RELEASE SAVEPOINT C
```

SAVEPOINT

The SAVEPOINT statement sets a savepoint within a unit of work to identify a point in time within the unit of work to which relational database changes can be rolled back.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

```

▶▶ SAVEPOINT savepoint-name [UNIQUE] ON ROLLBACK RETAIN CURSORS
▶▶ [ON ROLLBACK RETAIN LOCKS]

```

(1)

Notes:

- 1 The ROLLBACK options can be specified in any order.

Description

savepoint-name

Identifies a new savepoint.

UNIQUE

Specifies that the application program cannot reuse the savepoint name within the unit of work. An error occurs if a savepoint with the same name as *savepoint-name* already exists within the unit of work.

Omitting UNIQUE indicates that the application can reuse the savepoint name within the unit of work. If *savepoint-name* identifies a savepoint that already exists within the unit of work and the savepoint was not created with the UNIQUE option, the existing savepoint is destroyed and a new savepoint is created. Destroying a savepoint to reuse its name for another savepoint is not the same as releasing the savepoint. Reusing a savepoint name destroys only one savepoint. Releasing a savepoint with the RELEASE SAVEPOINT statement releases the savepoint and all savepoints that have been subsequently set.

ON ROLLBACK RETAIN CURSORS

Specifies that cursors that are opened after the savepoint is set are not closed upon rollback to the savepoint.

- If the savepoint contains DDL on which a cursor is dependent, the cursor is closed. Attempts to use such a cursor after a ROLLBACK TO SAVEPOINT results in an error.

- Otherwise, the cursor is not affected by the ROLLBACK TO SAVEPOINT (it remains open and positioned).

Although these cursors remain open after rollback to the savepoint, they might not be usable. For example, if rolling back to the savepoint causes the insertion of a row on which the cursor is positioned to be rolled back, using the cursor to update or delete the row results in an error.

ON ROLLBACK RETAIN LOCKS

Specifies that any locks that are acquired after the savepoint is set are not released on rollback to the savepoint.

Note

Effect on INSERT: In an application, inserts may be buffered. The buffer will be flushed when SAVEPOINT, ROLLBACK, or RELEASE TO SAVEPOINT statements are issued.

SAVEPOINT Restriction: A SAVEPOINT statement is not allowed if commitment control is not active for the activation group. For information on determining which commitment definition is used, see “Notes” on page 413.

Example

Assume that you want to set three savepoints at various points in a unit of work. Name the first savepoint A and allow the savepoint name to be reused. Name the second savepoint B and do not allow the name to be reused. Because you no longer need savepoint A when you are ready to set the third savepoint, reuse A as the name of the savepoint.

```
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
.
.
.
SAVEPOINT B UNIQUE ON ROLLBACK RETAIN CURSORS;
.
.
.
SAVEPOINT A ON ROLLBACK RETAIN CURSORS;
```

SELECT

| **SELECT**

| The SELECT statement is a form of query. It can be only be issued interactively.
| For detailed information, see “select-statement” on page 349 and Chapter 4,
| “Queries” on page 329.

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.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the statement,
 - The SELECT privilege on the table or view, and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

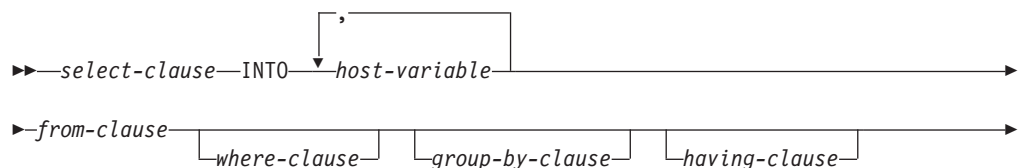
The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table.

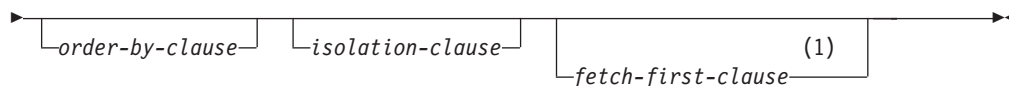
The authorization ID of the statement has the SELECT privilege on a view when:

- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views that this view is directly or indirectly dependent on. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

Syntax



SELECT INTO



Notes:

- 1 Only one row may be specified in the fetch-first-clause.

Description

| The result table is derived by evaluating the *isolation-clause*, *from-clause*,
| *where-clause*, *group-by-clause*, *having-clause*, *select-clause*, *order-by-clause*, and
| *fetch-first-clause*, in this order.

| See Chapter 4, “Queries” on page 329 for a description of the *select-clause*,
| *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*,
| *fetch-first-clause*, and *isolation-clause*.

INTO host variable,...

Identifies one or more host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. In the operational form of the INTO clause, a reference to a host structure is replaced by a reference to each of its variables. The first value in the result row is assigned to the first host variable in the list, the second value to the second host variable, and so on. The data type of each host variable must be compatible with its corresponding column.

Note

Host variable assignment: Each assignment to a host variable is performed according to the retrieval assignment rules described in “Assignments and Comparisons” on page 80. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. Note that there is no warning if there are more variables than the number of result columns. If a value is null, an indicator variable must be provided for that value.

If the specified host variable is character and is not large enough to contain the result, 'W' is assigned to SQLWARN1 in the SQLCA. The actual length of the result may be returned in the indicator variable associated with the host-variable, if an indicator variable is provided. For further information, see “References to Variables” on page 114.

If an assignment error occurs, the value of that host variable and any following host variables is unpredictable. Any values that have already been assigned to variables remain assigned.

Empty result table: If the result table is empty, the statement assigns '02000' to the SQLSTATE variable and does not assign values to the host variables.

Result tables with more than one row: If more than one row satisfies the search condition, statement processing is terminated and an error is returned (SQLSTATE 21000). If an error occurs because the result table has more than one row, values may or may not be assigned to the host variables. If values are assigned to the host variables, the row that is the source of the values is undefined and not predictable.

Result column evaluation considerations: When a TIME value is selected, if the ISO, EUR, or JIS format is used, the length of the variable must not be less than 5. If the length is 5, 6, or 7, the seconds part of the time is omitted from the result, a warning is returned (SQLSTATE 01004) and SQLWARN1 is set to 'W'. In this case, the seconds part of the time is assigned to the indicator variable if one is provided, and, if the length is 6 or 7, blank padding occurs so that the value is a valid string representation of a time.

If an error occurs while evaluating a result column in the SELECT list of a SELECT INTO statement, as the result of an arithmetic expression (such as division by zero, or overflow) or a numeric or character conversion error, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the host variable is undefined. In this case, however, the indicator variable is set to the value of -2. Processing of the statement continues and a warning is returned. If an indicator variable is not provided, an error is returned and no more values are assigned to variables. It is possible that some values have already been assigned to host variables and will remain assigned when the error is returned.

Examples

Example 1: Using a COBOL program statement, put the maximum salary (SALARY) from the EMPLOYEE table into the host variable MAX-SALARY (DECIMAL(9,2)).

```
EXEC SQL  SELECT MAX(SALARY)
          INTO :MAX-SALARY
          FROM EMPLOYEE WITH CS
END-EXEC.
```

Example 2: Using a Java program statement, select the row from the EMPLOYEE table on the connection context 'ctx' with an employee number (EMPNO) value the same as that stored in the host variable HOST_EMP (java.lang.String). Then put the last name (LASTNAME) and education level (EDLEVEL) from that row into the host variables HOST_NAME (String) and HOST_EDUCATE (Integer).

```
#sql [ctx] {  SELECT LASTNAME, EDLEVEL
             INTO :HOST_NAME, :HOST_EDUCATE
             FROM EMPLOYEE
             WHERE EMPNO = :HOST_EMP  };
```

|

SET CONNECTION

The SET CONNECTION statement establishes the current server of the activation group by identifying one of its existing connections.

Invocation

This statement can only be embedded within an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

SET CONNECTION is not allowed in a trigger. SET CONNECTION is not allowed in an external procedure if the external procedure is called on a remote server.

Authorization

None required.

Syntax

```

▶▶ SET CONNECTION server-name | host-variable

```

Description

server-name or *host-variable*

Identifies the connection by the specified server name or the server name contained in the host variable. If a host variable is specified:

- It must be a character-string variable with a length attribute that is not greater than 18.
- It must not be followed by an indicator variable.
- The server name must be left-justified within the host variable and must conform to the rules for forming an ordinary identifier.
- If the length of the server name is less than the length of the host variable, it must be padded on the right with blanks.

Let S denote the specified server name or the server name contained in the host variable. S must identify an existing connection of the application process. If S identifies the current connection, the state of S and all other connections of the application process are unchanged, but information about S is placed in the SQLERRP field of the SQLCA. The following rules apply when S identifies a dormant connection.

If the SET CONNECTION statement is successful:

- Connection S is placed in the current state.
- S is placed in the CURRENT SERVER special register.
- Information about server S is placed in the SQLERRP field of the SQLCA. If the server is an IBM relational database product, the information has the form *pppvrrm*, where:
 - *ppp* identifies the product as follows:
 - ARI for DB2 for VSE and VM
 - DSN for DB2 UDB for OS/390 and z/OS
 - QSQ for DB2 UDB for iSeries

SQL for all other DB2 products

- *vv* is a two-digit version identifier such as '04'
- *rr* is a two-digit release identifier such as '01'
- *m* is a one-digit modification level such as '0'

For example, if the server is Version 4 of DB2 UDB for OS/390 and z/OS, the value of SQLERRP is 'DSN04010'.

- Additional information about the connection is placed in the SQLERRD(4) field of the SQLCA. SQLERRD(4) will contain values indicating whether the server allows commitable updates to be performed. Following is a list of values and their meanings for the SQLERRD(4) field of the SQLCA on the CONNECT :
 - 1 - commitable updates can be performed and either the connection uses an unprotected conversation, is a connection established to an application requester driver program using a CONNECT (Type 1) statement, or is a local connection established using a CONNECT (Type 1) statement.
 - 2 - No commitable updates can be performed; conversation is unprotected.
 - 3 - It is unknown if commitable updates can be performed; conversation is protected.
 - 4 - It is unknown if commitable updates can be performed; conversation is unprotected.
 - 5 - It is unknown if commitable updates can be performed and the connection is either a local connection established using a CONNECT (Type 2) statement or a connection to an application requester driver program established using a CONNECT (Type 2) statement.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. Refer to Appendix B, "SQL Communication Area" for a description of the information in the SQLERRMC field.
- Any previously current connection is placed in the dormant state.

If the SET CONNECTION statement is unsuccessful, the connection state of the activation group and the states of its connections are unchanged.

Notes

SET CONNECTION for CONNECT (Type 1): The use of CONNECT (Type 1) statements does not prevent the use of SET CONNECTION, but the statement either fails or does nothing because dormant connections do not exist.

Status After Connection is Restored: When a connection is used, made dormant, and then restored to the current state in the same unit of work, the status of locks, cursors, and prepared statements for that connection reflects its last use by the activation group.

Local Connections: A SET CONNECTION to a local connection will fail if the current independent auxiliary Storage pool (IASP) name space does not match the local connection's relational database.

Example

Execute SQL statements at TOROLAB1, execute SQL statements at TOROLAB2, and then execute more SQL statements at TOROLAB1.

```
EXEC SQL CONNECT TO TOROLAB1;
```

SET CONNECTION

(Execute statements referencing objects at TOROLAB1)

```
EXEC SQL CONNECT TO TOROLAB2;
```

(Execute statements referencing objects at TOROLAB2)

```
EXEC SQL SET CONNECTION TOROLAB1;
```

(Execute statements referencing objects at TOROLAB1)

The first **CONNECT** statement creates the TOROLAB1 connection, the second **CONNECT** statement places it in the dormant state, and the **SET CONNECTION** statement returns it to the current state.

SET OPTION

The SET OPTION statement establishes the processing options to be used for SQL statements.

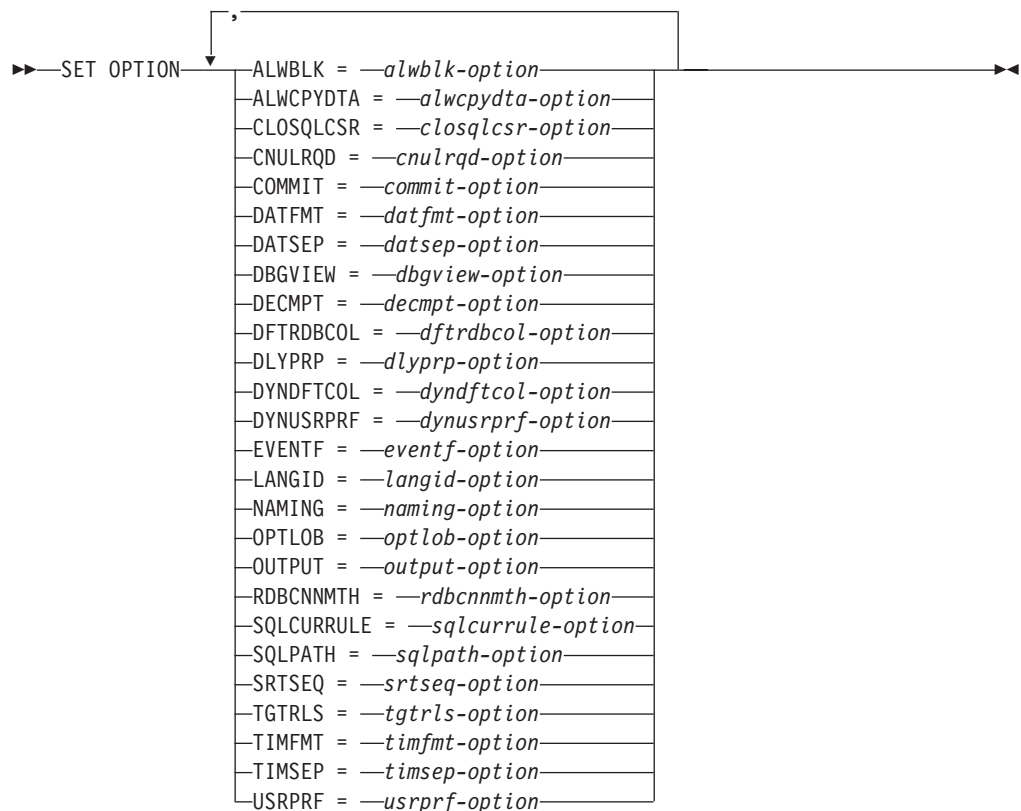
Invocation

This statement can be used in a REXX procedure or embedded in an application program. If used in a REXX procedure, it is an executable statement. If embedded in an application program, it is not executable and must precede any other SQL statements. This statement cannot be dynamically prepared.

Authorization

None required.

Syntax



alwblk-option:



SET OPTION

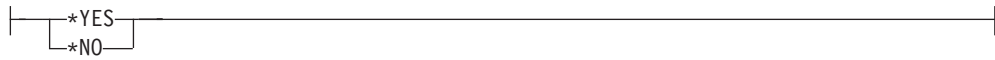
alwcpydta-option:



closqlcsr-option:



cnulrqd-option:



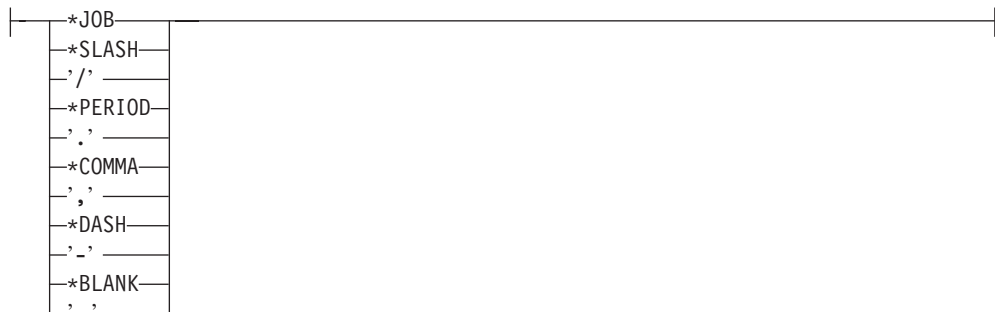
commit-option:

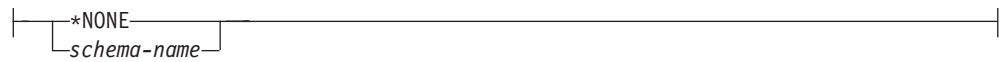
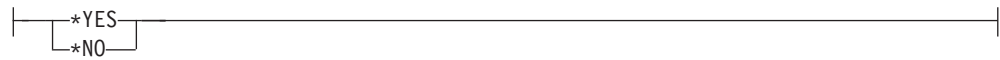
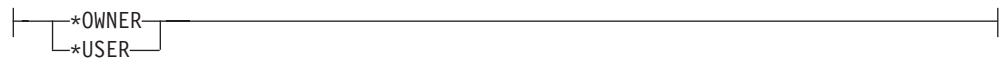
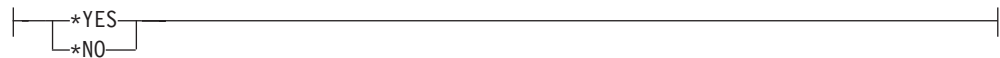


datfmt-option:



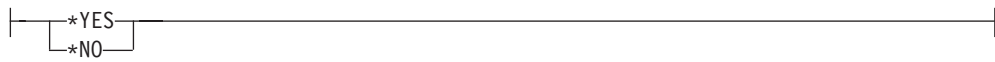
datsep-option:



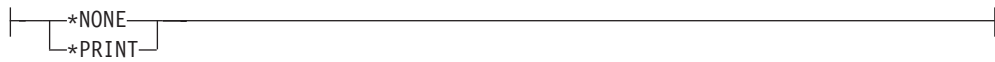
decmtpt-option:**dbgview-option:****dftrdbcol-option:****dlyprp-option:****dyndftcol-option:****dynusrprf-option:****eventf-option:****langid-option:****naming-option:**

SET OPTION

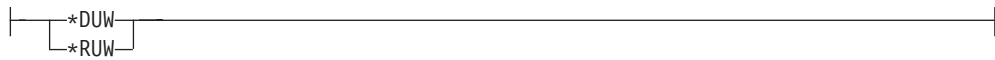
optlob-option:



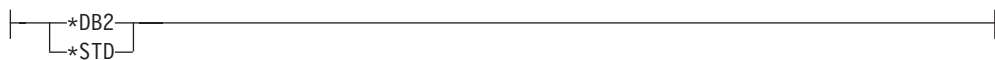
output-option:



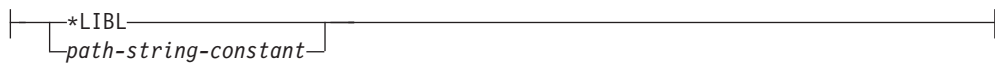
rdbcnmth-option:



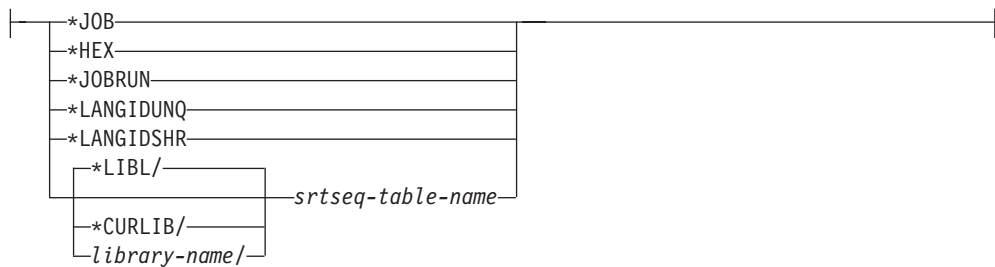
sqlcurrule-option:



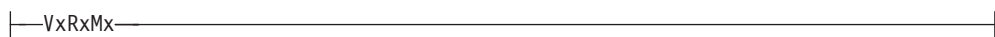
sqlpath-option:



srtseq-option:



tgtrls-option:



timfmt-option:



timsep-option:**usrprf-option:**

Description

ALWBLK

Specifies whether the database manager can use row blocking and the extent to which blocking can be used for read-only cursors. This option will be ignored in REXX.

***ALLREAD**

Rows are blocked for read-only cursors if COMMIT is *NONE or *CHG. All cursors in a program that are not explicitly able to be updated are opened for read-only processing even though EXECUTE or EXECUTE IMMEDIATE statements may be in the program.

Specifying *ALLREAD:

- Allows row blocking under commitment control level *CHG in addition to the blocking allowed for *READ.
- Can improve the performance of almost all read-only cursors in programs, but limits queries in the following ways:
 - The Rollback (ROLLBACK) command, a ROLLBACK statement in host languages, or the ROLLBACK HOLD SQL statement does not reposition a read-only cursor when *ALLREAD is specified.
 - Dynamic running of a positioned UPDATE or DELETE statement (for example, using EXECUTE IMMEDIATE), cannot be used to update a row in a cursor unless the DECLARE statement for the cursor includes the FOR UPDATE clause.

***NONE**

Rows are not blocked for retrieval of data for cursors.

Specifying *NONE:

- Guarantees that the data retrieved is current.
- May reduce the amount of time required to retrieve the first row of data for a query.
- Stops the database manager from retrieving a block of data rows that is not used by the program when only the first few rows of a query are retrieved before the query is closed.

SET OPTION

- Can degrade the overall performance of a query that retrieves a large number of rows.

***READ**

Rows are blocked for read-only retrieval of data for cursors when:

- *NONE is specified on the COMMIT parameter, which indicates that commitment control is not used.
- The cursor is declared with a FOR READ ONLY clause or there are no dynamic statements that could run a positioned UPDATE or DELETE statement for the cursor.

Specifying *READ can improve the overall performance of queries that meet the above conditions and retrieve a large number of rows.

ALWCPYDTA

Specifies whether a copy of the data can be used in a SELECT statement. This option will be ignored in REXX.

***OPTIMIZE**

The system determines whether to use the data retrieved directly from the database or to use a copy of the data. The decision is based on which method provides the best performance. If COMMIT is *CHG or *CS and ALWBLK in not *ALLREAD, or if COMMIT is *ALL or *RR, then a copy of the data is used only when it is necessary to run a query.

***YES**

A copy of the data is used only when necessary.

***NO**

A copy of the data is not allowed. If a temporary copy of the data is required to perform the query, an error message is returned.

CLOSQLCSR

Specifies when SQL cursors are implicitly closed, SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released. SQL cursors are explicitly closed when you issue the CLOSE, COMMIT, or ROLLBACK (without HOLD) SQL statements. This option will be ignored in REXX.

*ENDACTGRP and *ENDMOD are for use by ILE programs and modules.

*ENDPGM, *ENDSQL, and *ENDJOB are for use by non-ILE programs.

This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

***ENDACTGRP**

SQL cursors are closed, SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released when the activation group ends.

***ENDMOD**

SQL cursors are closed and SQL prepared statements are implicitly discarded when the module is exited. LOCK TABLE locks are released when the first SQL program on the call stack ends.

***ENDPGM**

SQL cursors are closed and SQL prepared statements are discarded when the program ends. LOCK TABLE locks are released when the first SQL program on the call stack ends.

***ENDSQL**

SQL cursors remain open between calls and can be fetched without running another SQL OPEN. One of the programs higher on the call stack must have run at least one SQL statement. SQL cursors are closed, SQL prepared statements are discarded, and LOCK TABLE locks are released

when the first SQL program on the call stack ends. If *ENDSQL is specified for a program that is the first SQL program called (the first SQL program on the call stack), the program is treated as if *ENDPGM was specified.

***ENDJOB**

SQL cursors remain open between calls and can be fetched without running another SQL OPEN. The programs higher on the call stack do not need to have run SQL statements. SQL cursors are left open, SQL prepared statements are preserved, and LOCK TABLE locks are held when the first SQL program on the call stack ends. SQL cursors are closed, SQL prepared statements are discarded, and LOCK TABLE locks are released when the job ends.

CNULRQD

Specifies whether a NUL-terminator is returned for character and graphic host variables. This option will only be used for SQL statements in C and C++ programs.

This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

***YES**

Output character and graphic host variables always contain the NUL-terminator. If there is not enough space for the NUL-terminator, the data is truncated and the NUL-terminator is added. Input character and graphic host variables require a NUL-terminator.

***NO**

For output character and graphic host variables, the NUL-terminator is not returned when the host variable is exactly the same length as the data. Input character and graphic host variables do not require a NUL-terminator.

COMMIT

Specifies the isolation level to be used. In REXX, files that are referred to in the source are not affected by this option. Only tables, views, and packages referred to in SQL statements are affected. For more information about isolation levels, see "Isolation Level" on page 21

***CHG**

Specifies the isolation level of Uncommitted Read.

***NONE**

Specifies the isolation level of No Commit. If the DROP SCHEMA statement is included in a REXX procedure, *NONE must be used.

***CS**

Specifies the isolation level of Cursor Stability.

***ALL**

Specifies the isolation level of Read Stability.

***RR**

Specifies the isolation level of Repeatable Read.

DATFMT

Specifies the format used when accessing date result columns. All output date fields are returned in the specified format. For input date strings, the specified value is used to determine whether the date is specified in a valid format.

Note: An input date string that uses the format *USA, *ISO, *EUR, or *JIS is always valid.

SET OPTION

***JOB:**

The format specified for the job is used. Use the Display Job (DSPJOB) command to determine the current date format for the job.

***ISO**

The International Organization for Standardization (ISO) date format (yyyy-mm-dd) is used.

***EUR**

The European date format (dd.mm.yyyy) is used.

***USA**

The United States date format (mm/dd/yyyy) is used.

***JIS**

The Japanese Industrial Standard date format (yyyy-mm-dd) is used.

***MDY**

The date format (mm/dd/yy) is used.

***DMY**

The date format (dd/mm/yy) is used.

***YMD**

The date format (yy/mm/dd) is used.

***JUL**

The Julian date format (yy/ddd) is used.

DATSEP

Specifies the separator used when accessing date result columns.

Note: This parameter applies only when *JOB, *MDY, *DMY, *YMD, or *JUL is specified on the DATFMT parameter.

***JOB**

The date separator specified for the job is used. Use the Display Job (DSPJOB) command to determine the current value for the job.

***SLASH** or **'/'**

A slash (/) is used.

***PERIOD** or **'.'**

A period (.) is used.

***COMMA** or **','**

A comma (,) is used.

***DASH** or **'-'**

A dash (-) is used.

***BLANK** or **' '**

A blank () is used.

DBGVIEW

Specifies the type of debug information to be provided by the compiler. The DBGVIEW parameter can only be specified in the body of SQL functions, procedures, and triggers. The possible choices are:

***NONE**

A debug view will not be generated.

***SOURCE**

Allows the compiled module object to be debugged using SQL statement source.

|
|
|

***STMT**

Allows the compiled module object to be debugged using program statement numbers and symbolic identifiers.

***LIST**

Generates the listing view for debugging the compiled module object.

DECMPT

Specifies the symbol that you want to represent the decimal point. The possible choices are:

***PERIOD**

The representation for the decimal point is a period.

***COMMA**

The representation for the decimal point is a comma.

***SYSVAL**

The representation for the decimal point is the system value (QDECFMT).

***JOB**

The representation for the decimal point is the job value (DECFMT).

DFTRDBCOL

Specifies the schema name used for the unqualified names of tables, views, indexes, and SQL packages. This parameter applies only to static SQL statements. This option will be ignored in REXX.

This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

***NONE**

The naming convention specified on the OPTION precompile parameter or by the SET OPTION NAMING option will be used.

schema-name

Specify the name of the schema. This value is used instead of the naming convention specified on the OPTION precompile parameter or by the SET OPTION NAMING option.

DLYPRP

Specifies whether the dynamic statement validation for a PREPARE statement is delayed until an OPEN, EXECUTE, or DESCRIBE statement is run. Delaying validation improves performance by eliminating redundant validation. This option will be ignored in REXX.

***NO**

Dynamic statement validation is not delayed. When the dynamic statement is prepared, the access plan is validated. When the dynamic statement is used in an OPEN or EXECUTE statement, the access plan is revalidated. Because the authority or the existence of objects referred to by the dynamic statement may change, you must still check the SQLCODE or SQLSTATE after issuing the OPEN or EXECUTE statement to ensure that the dynamic statement is still valid.

***YES**

Dynamic statement validation is delayed until the dynamic statement is used in an OPEN, EXECUTE, or DESCRIBE SQL statement. When the dynamic statement is used, the validation is completed and an access plan is built. If you specify *YES, you should check the SQLCODE and SQLSTATE after running an OPEN, EXECUTE, or DESCRIBE statement to ensure that the dynamic statement is valid.

SET OPTION

Note: If you specify *YES, performance is not improved if the INTO clause is used on the PREPARE statement or if a DESCRIBE statement uses the dynamic statement before an OPEN is issued for the statement.

DYNDFTCOL

Specifies the schema name specified for the DFTRDBCOL parameter is also used for dynamic statements. This option will be ignored in REXX.

This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

*NO

Do not use the value specified for DFTRDBCOL for unqualified names of tables, views, indexes, and SQL packages for dynamic SQL statements. The naming convention specified on the OPTION precompile parameter or by the SET OPTION NAMING option will be used.

*YES

The schema name specified for DFTRDBCOL will be used for the unqualified names of the tables, views, indexes, and SQL packages in dynamic SQL statements.

DYNUSRPRF

Specifies the user profile to be used for dynamic SQL statements. This option will be ignored in REXX.

*USER

Local dynamic SQL statements are run under the user profile of the job. Distributed dynamic SQL statements are run under the user profile of the server job.

*OWNER

Local dynamic SQL statements are run under the user profile of the program's owner. Distributed dynamic SQL statements are run under the user profile of the SQL package's owner.

EVENTF

Specifies whether an event file will be generated. CoOperative Development Environment/400 (CODE/400) uses the event file to provide error feedback integrated with the CODE/400 editor.

*YES

The compiler produces an event file for use by CoOperative Development Environment/400 (CODE/400).

*NO

The compiler will not produce an event file for use by CoOperative Development Environment/400 (CODE/400).

LANGID

Specifies the language identifier to be used when SRTSEQ(*LANGIDUNQ) or SRTSEQ(*LANGIDSHR) is specified.

*JOB or *JOBRUN

The LANGID value for the job is used.

For distributed applications, LANGID(*JOBRUN) is valid only when SRTSEQ(*JOBRUN) is also specified.

language-id

Specify a language identifier to be used. For information on the values that can be used for the language identifier, see the Language identifier topic in the iSeries Information Center.

NAMING

Specifies whether the SQL naming convention or the system naming convention is to be used. This option is not allowed in an SQL function, SQL procedure, or SQL trigger.

The possible choices are:

***SYS**

The system naming convention will be used.

***SQL**

The SQL naming convention will be used.

OPTLOB

Specifies whether accesses to LOBs can be optimized when accessing through DRDA. The possible choices are:

***YES**

LOB accesses should be optimized. The first FETCH for a cursor determines how the cursor will be used for LOBs on all subsequent FETCHes. This option remains in effect until the cursor is closed.

If the first FETCH uses a LOB locator to access a LOB column, no subsequent FETCH for that cursor can fetch that LOB column into a LOB host variable.

If the first FETCH places the LOB column into a LOB host variable, no subsequent FETCH for that cursor can use a LOB locator for that column.

***NO**

LOB accesses should not be optimized. There is no restriction on whether a column is retrieved into a LOB locator or into a LOB host variable. This option can cause performance to degrade.

OUTPUT

Specifies whether the precompiler and compiler listings are generated. The OUTPUT parameter can only be specified in the body of SQL functions, procedures, and triggers. The possible choices are:

***NONE**

The precompiler and compiler listings are not generated.

***PRINT**

The precompiler and compiler listings are generated.

RDBCNNMTH

Specifies the semantics used for CONNECT statements. This option will be ignored in REXX.

***DUW**

CONNECT (Type 2) semantics are used to support distributed unit of work. Consecutive CONNECT statements to additional relational databases do not result in disconnection of previous connections.

***RUW**

CONNECT (Type 1) semantics are used to support remote unit of work. Consecutive CONNECT statements result in the previous connection being disconnected before a new connection is established.

SQLCURRULE

Specifies the semantics used for SQL statements.

SET OPTION

***DB2**

The semantics of all SQL statements will default to the rules established for DB2. The following semantics are controlled by this option:

- Hexadecimal constants are treated as character data.

***STD**

The semantics of all SQL statements will default to the rules established by the ISO and ANSI SQL standards. The following semantics are controlled by this option:

- Hexadecimal constants are treated as binary data.

SQLPATH

Specifies the path to be used to find procedures, functions, and user defined types in static SQL statements. This option will be ignored in REXX.

***LIBL**

The path used is the library list at runtime.

character-string

A character constant with one or more schema names that are separated by commas.

SRTSEQ

Specifies the sort sequence table to be used for string comparisons in SQL statements.

Note: *HEX must be specified if a REXX procedure connects to a server that is not a DB2 UDB for iSeries or an iSeries system whose release level is prior to V2R3M0.

***JOB or *JOB RUN**

The SRTSEQ value for the job is used.

***HEX**

A sort sequence table is not used. The hexadecimal values of the characters are used to determine the sort sequence.

***LANGIDUNQ**

The sort sequence table must contain a unique weight for each character in the code page.

***LANGIDSHR**

The shared-weight sort table for the LANGID specified is used.

srtseq-table-name

Specify the name of the sort sequence table to be used with this program. The name of the sort sequence table can be qualified by one of the following library values:

***LIBL**

All libraries in the user and system portions of the job's library list are searched until the first match is found.

***CURLIB**

The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name

Specify the name of the library to be searched.

TGTRLS

Specifies the release of the operating system on which the user intends to use

the object being created. The TGTRLS parameter can only be specified in the body of SQL functions, procedures, and triggers. The possible choices are:

VxRxMx

Specify the release in the format VxRxMx, where Vx is the version, Rx is the release, and Mx is the modification level. For example, V5R1M0 is version 5, release 1, modification level 0. The object can be used on a system with the specified release or with any subsequent release of the operating system installed.

Valid values depend on the current version, release, and modification level, and they change with each new release. If you specify a release-level which is earlier than the earliest release level supported by the database manager, an error message is sent indicating the earliest supported release.

The TGTRLS option can only be specified for SQL functions, SQL procedures, and triggers.

TIMFMT

Specifies the format used when accessing time result columns. All output time fields are returned in the specified format. For input time strings, the specified value is used to determine whether the time is specified in a valid format.

Note: An input time string that uses the format *USA, *ISO, *EUR, or *JIS is always valid.

***HMS**

The (hh:mm:ss) format is used.

***ISO**

The International Organization for Standardization (ISO) time format (hh.mm.ss) is used.

***EUR**

The European time format (hh.mm.ss) is used.

***USA**

The United States time format (hh:mm xx) is used, where xx is AM or PM.

***JIS**

The Japanese Industrial Standard time format (hh:mm:ss) is used.

TIMSEP

Specifies the separator used when accessing time result columns.

Note: This parameter applies only when *HMS is specified on the TIMFMT parameter.

***JOB**

The time separator specified for the job is used. Use the Display Job (DSPJOB) command to determine the current value for the job.

***COLON** or ':'

A colon (:) is used.

***PERIOD** or '.'

A period (.) is used.

***COMMA** or ','

A comma (,) is used.

***BLANK** or ''

A blank () is used.

SET OPTION

USRPRF

Specifies the user profile that is used when the compiled program object is run, including the authority that the program object has for each object in static SQL statements. The profile of either the program owner or the program user is used to control which objects can be used by the program object. This option will be ignored in REXX.

*NAMING

The user profile is determined by the naming convention. If the naming convention is *SQL, USRPRF(*OWNER) is used. If the naming convention is *SYS, USRPRF(*USER) is used.

*USER

The profile of the user running the program object is used.

*OWNER

The user profiles of both the program owner and the program user are used when the program is run.

Notes

At the start of a REXX procedure the options are set to their default value. The default value for each option is the first value listed in the syntax diagram. When an option is changed by a SET OPTION statement, the new value will stay in effect until the option is changed again or the REXX procedure ends.

For application programs, the processing options are initially set to the values specified on the CRTSQLxxx command. Each option is updated as it is encountered within a SET OPTION statement. All SET OPTION statements must precede any other embedded SQL statements.

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- *UR can be used as a synonym for *CHG.
- *NC can be used as a synonym for *NONE.
- *RS can be used as a synonym for *ALL.

Examples

Example 1: Set the isolation level to *ALL and the naming mode to SQL names.

```
EXEC SQL SET OPTION COMMIT =*ALL, NAMING =*SQL
```

Example 2: Set the date format to European, the isolation level to *CS, and the decimal point to the comma.

```
EXEC SQL SET OPTION DATFMT = *EUR, COMMIT = *CS, DECMPNT = *COMMA
```

SET PATH

The SET PATH statement changes the value of the CURRENT PATH special register.

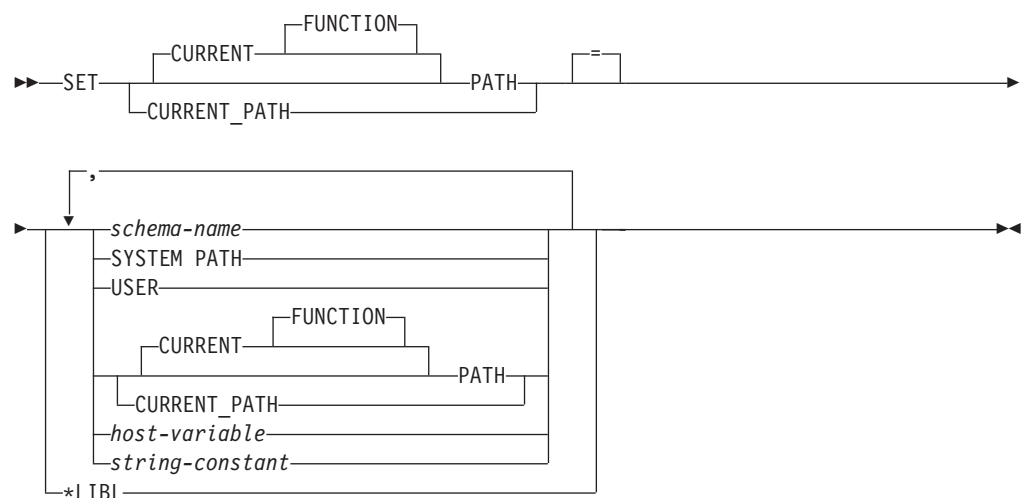
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

No authorization is required to execute this statement.

Syntax



Description

schema-name

Identifies a schema. No validation that the schema exists is made at the time the PATH is set. For example, if a *schema-name* is misspelled, it could affect the way subsequent SQL operates. Although not recommended, PATH can be specified as a *schema-name* if it is specified as a delimited identifier.

SYSTEM PATH

This value is the same as specifying the schema names "QSYS","QSYS2".

USER

This value is the USER special register.

CURRENT PATH

Specifies the value of the CURRENT PATH special register before the execution of this statement.

host-variable

Specifies a host variable that contains a schema name.

The host variable:

SET PATH

- Must be a CHAR, VARCHAR, UCS-2 GRAPHIC, or UCS-2 VARGRAPHIC variable. The actual length of the contents of the *host-variable* must not exceed the maximum length of a schema name.
- Must not be followed by an indicator variable.
- Must not be the null value.
- Must include a schema that is left justified and must conform to the rules for forming an ordinary or delimited identifier.
- Must not contain lowercase letters or characters that cannot be specified in an ordinary identifier.
- Must be padded on the right with blanks if the host variable is fixed length character.
- Must not contain SYSTEM PATH, USER, or CURRENT PATH.

string-constant

A character constant with 1 or more schema names that are separated by commas.

Notes

Transaction Considerations: The SET PATH statement is not a committable operation. ROLLBACK has no effect on the CURRENT PATH.

Rules for the Content of the SQL path:

- A schema name must not appear more than once in the path.
- 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, and separating each schema name by a comma. An error is returned if the length of the resulting string exceeds 3483 bytes. A maximum of 268 schema names can be represented in the path.
- There is a difference between specifying a single keyword (such as USER, or PATH, or CURRENT_PATH) as a single keyword, or as a delimited identifier. To indicate that the current value of a special register specified as a single keyword should be used in the SQL path, specify the name of the special register as a keyword. If the name of the special register is specified as a delimited identifier instead (for example, "USER"), it is interpreted as a schema name of that value ('USER'). For example, assuming that the current value of the USER special register is SMITH, then SET PATH = SYSIBM, USER, "USER" results in a CURRENT PATH value of "SYSIBM","SMITH","USER".
- The following rules are used to determine whether a value specified in a SET PATH statement is a variable or a *schema-name*:
 - If *name* is the same as a parameter or SQL variable in the SQL procedure, *name* is interpreted as a parameter or SQL variable, and the value in *name* is assigned to PATH.
 - If *name* is not the same as a parameter or SQL variable in the SQL procedure, *name* is interpreted as *schema-name*, and the value *name* is assigned to PATH.

The System Path: SYSTEM PATH refers to the system path for a platform. The schemas QSYS and QSYS2 do not need to be specified. If not included in the path, they are implicitly assumed as the last schemas (in this case, it is not included in the CURRENT PATH special register).

The initial value of the CURRENT PATH special register is *LIBL if system naming was used for the first SQL statement run in the activation group. The initial value

is "QSYS","QSYS2", "X" (where X is the value of the USER special register) if SQL naming was used for the first SQL statement.

Using the SQL path: The CURRENT PATH special register is used to resolve user-defined distinct types and functions in dynamic SQL statements. For more information see "SQL Path" on page 53.

Example

The following statement sets the CURRENT PATH special register.

```
SET PATH = FERMAT, "McDuff", SYSIBM
```

The following statement retrieves the current value of the SQL path special register into the host variable called CURPATH.

```
EXEC SQL VALUES (CURRENT PATH) INTO :CURPATH;
```

The value would be "FERMAT","McDuff","SYSIBM" if set by the previous example.

SET RESULT SETS

The SET RESULT SETS statement identifies one or more result sets that can be returned from an external procedure when the procedure is called by a iSeries Access client, the SQL Call Level Interface, or when accessed from a remote system using DRDA.

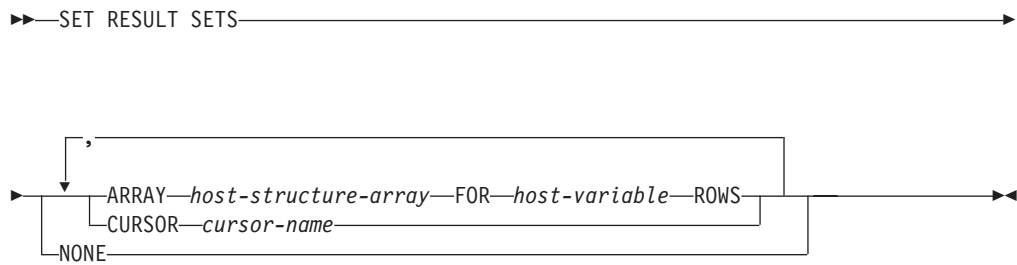
Invocation

| This statement can only be embedded in an application program. It is an
 | executable statement that cannot be dynamically prepared. It is not allowed in a
 | Java or REXX procedure.

Authorization

None required.

Syntax



Description

CURSOR *cursor-name*

Identifies a cursor to be used to define a result set that can be returned from a procedure. The *cursor-name* must identify a declared cursor as explained in "Description" on page 577 for the DECLARE CURSOR statement. When the SET RESULT SETS statement is executed, the cursor must be in the open state.

ARRAY *host-structure-array*

host-structure-array identifies an array of host structures defined in accordance with the rules for declaring host structures. The array cannot contain a C NUL-terminated host variable.

The first structure in the array corresponds to the first row of the result set, the second structure in the array corresponds to the second row of the result set, and so on. In addition, the first value in the row corresponds to the first item in the structure, the second value in the row corresponds to the second item in the structure, and so on.

LOBs cannot be returned in an array when using DRDA.

Only one array can be specified in a SET RESULT SETS statement.

FOR *host-variable* **ROWS**

Specifies the number of rows in the result set. The *host-variable* must be a numeric host variable with zero scale, and it must not include an indicator variable. The number of rows specified must be in the range of 0 to 32767 and must be less than or equal to the dimension of the host structure array.

NONE

Specifies that no result sets will be returned. Cursors left open when the procedure ends will not be returned.

Notes

Result sets are only returned from a procedure when the procedure is called from a client using the iSeries Access Open Database Connectivity (ODBC) driver, a client using the iSeries Access Optimized SQL API, from the SQL Call Level Interface, or from JDBC. Result sets are also returned whenever a non-iSeries client accesses the iSeries as a server using a Distributed Relational Database Architecture (DRDA) connection.

External procedures: There are three ways to return result sets from an external procedure:

- If a SET RESULT SETS statement is executed in the procedure, the SET RESULT SETS statement identifies the result sets. The result sets are returned in the order specified on the SET RESULT SETS statement.
- If a SET RESULT SETS statement is not executed in the procedure,
 - If no cursors have specified a WITH RETURN clause, each cursor that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.
 - If any cursors have specified a WITH RETURN clause, each cursor that is defined with the WITH RETURN clause that the procedure opens and leaves open when it returns identifies a result set. The result sets are returned in the order in which the cursors are opened.

When a result set is returned using an open cursor, the rows are returned starting with the current cursor position.

The RESULT SETS clause should be specified on the CREATE PROCEDURE (External) statement or DECLARE PROCEDURE statement to return result sets from a procedure. The maximum number of result sets returned cannot be larger than the number specified on the CREATE PROCEDURE (External) statement or DECLARE PROCEDURE statement.

SQL procedures: In order to return result sets from an SQL procedure, the procedure must be created with the RESULT SETS clause. Each cursor that is defined with the WITH RETURN clause that the procedure opens and leaves open when it returns identifies a result set.

- If a SET RESULT SETS statement is executed in the procedure, the SET RESULT SETS statement identifies which of these result sets to return. The result sets are returned in the order specified on the SET RESULT SETS statement.
- If a SET RESULT SETS statement is not executed in the procedure the result sets are returned in the order in which the cursors are opened.

When a result set is returned using an open cursor, the rows are returned starting with the current cursor position.

The RESULT SETS clause must be specified on the CREATE PROCEDURE (SQL) statement to return any result sets from an SQL procedure. The maximum number of result sets returned cannot be larger than the number specified on the CREATE PROCEDURE statement.

SET RESULT SETS

Example

The following SET RESULT SETS statement specifies cursor X as the result set that will be returned when the procedure is called. For more information and complete examples showing the use of result sets from ODBC clients, see the iSeries Access category in the iSeries Information Center.

```
EXEC SQL SET RESULT SETS CURSOR X;
```

SET SCHEMA

The SET SCHEMA statement changes the value of the CURRENT SCHEMA special register.

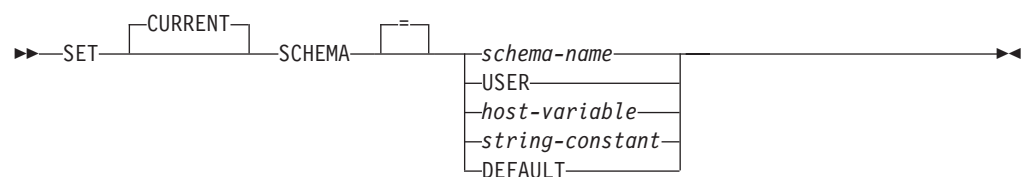
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

No authorization is required to execute this statement.

Syntax



Description

The value of the CURRENT SCHEMA special register is replaced by the value specified.

schema-name

Identifies a schema. No validation that the schema exists is made at the time the CURRENT SCHEMA is set.

USER

This value is the USER special register.

host-variable

A host variable which contains a schema name.

The host variable:

- Must be a character-string variable.
- Must not be followed by an indicator variable.
- Must include a schema that is left justified and must conform to the rules for forming an ordinary identifier.
- Must be padded on the right with blanks.
- Must not be the null value.

string-constant

A character constant with a schema name.

DEFAULT

The CURRENT SCHEMA is set to its initial value. The initial value for SQL naming is USER. The initial value for system naming is *LIBL.

Notes

CURRENT SCHEMA: The value of the CURRENT SCHEMA special register is used as the qualifier for all unqualified names in all dynamic SQL statements

SET SCHEMA

except in programs where the DYNDFTCOL has been specified. If DYNDFTCOL is specified in a program, its schema name is used instead of the CURRENT SCHEMA schema name.

For SQL naming, the initial value of the CURRENT SCHEMA special register is equivalent to USER. For system naming, the initial value of the CURRENT SCHEMA special register is '*LIBL'.

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 distinct 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.

Transaction Considerations: The SET SCHEMA statement is not a committable operation. ROLLBACK has no effect on the CURRENT SCHEMA.

Keyword Synonym: 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.

SET SCHEMA is equivalent to calling the QSQCHGDC API.

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 TRANSACTION

The SET TRANSACTION statement sets the isolation level and read only attribute for the current unit of work.

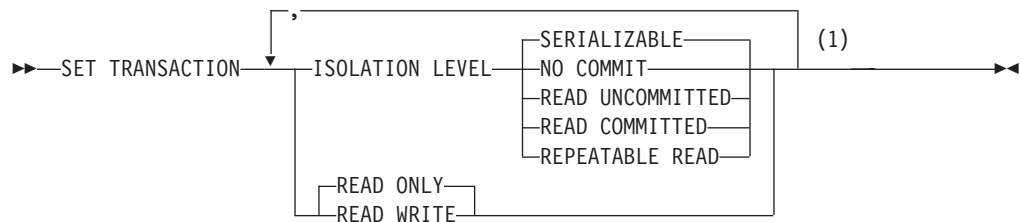
Invocation

This statement can be embedded within an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Notes:

- 1 Only one ISOLATION LEVEL clause and one READ WRITE or READ ONLY clause may be specified.

Description

ISOLATION LEVEL

Specifies the isolation level of the transaction. If the ISOLATION LEVEL clause is not specified, ISOLATION LEVEL SERIALIZABLE is implicit

NO COMMIT

Specifies isolation level NC (COMMIT(*NONE)).

READ UNCOMMITTED

Specifies isolation level UR (COMMIT(*CHG)).

READ COMMITTED

Specifies isolation level CS (COMMIT(*CS)).

REPEATABLE READ ⁶⁸

Specifies isolation level RS (COMMIT(*ALL)).

SERIALIZABLE

Specifies isolation level RR (COMMIT(*RR)).

READ WRITE or READ ONLY

Specifies whether the transaction allows data change operations.

68. REPEATABLE READ is the ISO and ANS standard term that corresponds to the isolation level of *ALL for DB2 UDB for iSeries and the isolation level of Read Stability (RS) in IBM SQL. SERIALIZABLE is used in the ISO and ANS standard for what IBM SQL calls Repeatable Read (RR).

SET TRANSACTION

READ WRITE

Specifies that all SQL operations are allowed. This is the default unless ISOLATION LEVEL READ UNCOMMITTED is specified.

READ ONLY

Specifies that only SQL operations that do not change SQL data are allowed. If ISOLATION LEVEL READ UNCOMMITTED is specified, this is the default.

Notes

Scope of SET TRANSACTION: The SET TRANSACTION statement sets the isolation level for SQL statements for the current activation group of the process. If that activation group has commitment control scoped to the job, then the SET TRANSACTION statement sets the isolation level of all other activation groups with job commit scoping as well.

If an isolation clause is specified in an SQL statement, that isolation level overrides the transaction isolation level and is used for the SQL statement.

The scope of the SET TRANSACTION statement is based on the context in which it is run. If the SET TRANSACTION statement is run in a trigger, the isolation level specified applies to all subsequent SQL statements until another SET TRANSACTION statement occurs or until the trigger completes, whichever happens first. If the SET TRANSACTION statement is run outside a trigger, the isolation level specified applies to all subsequent SQL statements (except those statements within a trigger that are executed after a SET TRANSACTION statement in the trigger) until a COMMIT or ROLLBACK operation occurs.

For more information about isolation levels, see “Isolation Level” on page 21.

SET TRANSACTION Restrictions: The SET TRANSACTION statement can only be executed when it is the first SQL statement in a unit of work, unless it is executed in a trigger. In a trigger program, SET TRANSACTION with READ ONLY is allowed only on a COMMIT boundary. The SET TRANSACTION statement can be executed in a trigger at any time, but it is recommended that it be executed as the first statement in the trigger. The SET TRANSACTION statement is useful within triggers to set the isolation level for SQL statements in the trigger to the same level as the application which caused the trigger to be activated.

A SET TRANSACTION statement is not allowed if the current connection is to a remote server unless it is in a trigger at the current server. Once a SET TRANSACTION statement is executed, CONNECT and SET CONNECTION statements are not allowed until the unit of work is committed or rolled back.

The SET TRANSACTION statement has no effect on WITH HOLD cursors that are still open when the SET TRANSACTION statement is executed.

Keyword Synonym: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keywords NC or NONE can be used as synonyms for NO COMMIT.
- The keywords UR and CHG can be used as synonyms for READ UNCOMMITTED.
- The keyword CS can be used as a synonym for READ COMMITTED.
- The keywords RS or ALL can be used as synonyms for REPEATABLE READ.

SET TRANSACTION

- The keyword RR can be used as a synonym for SERIALIZABLE.

Examples

Example 1: The following SET TRANSACTION statement sets the isolation level to NONE (equivalent to specifying *NONE on the SQL precompiler command).

```
EXEC SQL SET TRANSACTION ISOLATION LEVEL NO COMMIT;
```

Example 2: The following SET TRANSACTION statement sets the isolation level to SERIALIZABLE.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

SET transition-variable

The SET transition-variable statement assigns values to new *transition-variables*.

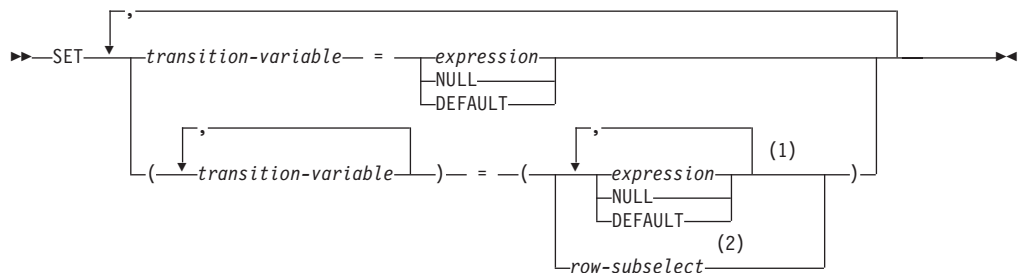
Invocation

This statement can only be used as an SQL statement in a BEFORE trigger. It is an executable statement that cannot be dynamically prepared.

Authorization

If a *row-subselect* is specified, see Chapter 4, "Queries" on page 329 for an explanation of the authorization required for each subselect.

Syntax



Notes:

- 1 The number of *expressions*, *NULLs*, and *DEFAULTs* must match the number of *transition-variables*.
- 2 The number of columns in the select list must match the number of *transition-variables*.

Description

transition-variable

Identifies the column to be updated in the new row. A *transition-variable* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value. An OLD *transition-variable* must not be identified.

A *transition-variable* must not be identified more than once in the same SET transition-variable statement.

The data type of each *transition-variable* must be compatible with its corresponding result column. Values are assigned to *transition-variables* according to the assignment rules to a column. For more information see "Assignments and Comparisons" on page 80.

expression

Specifies the new value of the *transition-variable*. The *expression* is any expression of the type described in "Expressions" on page 129. The expression cannot include a column function.

SET transition-variable

An *expression* may contain references to OLD and NEW *transition-variables*. If the CREATE TRIGGER statement contains both OLD and NEW clauses, references to *transition-variables* must be qualified by the *correlation-name* to specify which *transition-variable*.

NULL

Specifies the null value. NULL can only be specified for nullable columns.

DEFAULT

Specifies that the default value of the column associated with the *transition-variable* will be used. If the column is an IDENTITY column or has a ROWID data type, the value is generated by the database manager.

row-subselect

A subselect that returns a single result row. The result column values are assigned to each corresponding *transition-variable*. If the result of the subselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

Notes

Multiple Assignments

If more than one assignment is included in the same SET transition-variable statement, all *expressions* are evaluated before the assignments are performed. Thus, references to *transition-variables* in an expression are always the value of the *transition-variable* prior to any assignment in the single SET statement.

Examples

Example 1: Ensure that the salary column is never greater than 50000. If the new value is greater than 50000, set it to 50000.

```
CREATE TRIGGER LIMIT_SALARY
  BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AS NEW_VAR
  FOR EACH ROW MODE DB2SQL
  WHEN (NEW_VAR.SALARY > 50000)
  BEGIN ATOMIC
    SET NEW_VAR.SALARY = 50000;
  END
```

Example 2: When the job title is updated, increase the salary based on the new job title. Assign the years in the position to 0.

```
CREATE TRIGGER SET_SALARY
  BEFORE UPDATE OF JOB ON STAFF
  REFERENCING OLD AS OLD_VAR
                NEW AS NEW_VAR
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET (NEW_VAR.SALARY, NEW_VAR.YEARS) =
      (OLD_VAR.SALARY * CASE NEW_VAR.JOB
        WHEN 'Sales' THEN 1.1
        WHEN 'Mgr'   THEN 1.05
        ELSE 1 END ,0);
  END
```

SET variable

The SET variable statement produces a result table consisting of at most one row and assigns the values in that row to host variables.

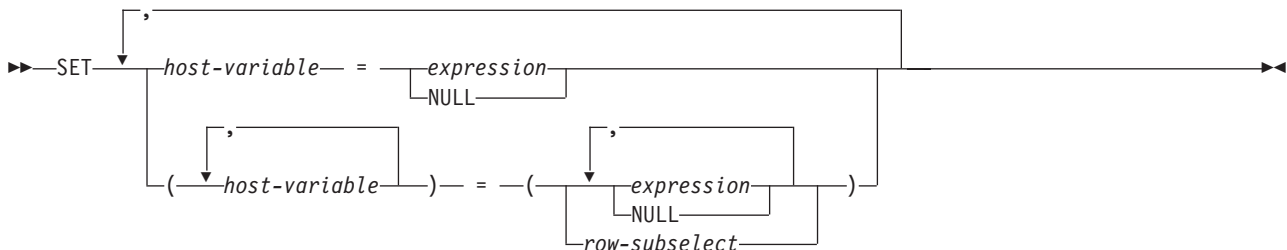
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

If a *row-subselect* is specified, see Chapter 4, “Queries” on page 329 for an explanation of the authorization required for each subselect.

Syntax



Description

host-variable, ...

Identifies one or more host variables or host structures that must be declared in accordance with the rules for declaring host variables (see “References to Host Variables” on page 114). A host structure is logically replaced by a list of host variables that represent each of the elements of the host structure.

The value to be assigned to each *host-variable* can be specified immediately following the *host-variable*, for example, *host-variable = expression, host-variable = expression*. Or, sets of parentheses can be used to specify all the *host-variables* and then all the values, for example, *(host-variable, host-variable) = (expression, expression)*.

The data type of each host variable must be compatible with its corresponding result column. Each assignment is made according to the rules described in “Assignments and Comparisons” on page 80. The number of *host-variables* specified to the left of the equal operator must equal the number of values in the corresponding result specified to the right of the equal operator. If the value is null, an indicator variable must be provided. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

If an error occurs as the result of an arithmetic expression in the *expression* or SELECT list of the subselect (division by zero, or overflow) or a character conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the host variable is undefined. In this case, however, the indicator variable is set to -2.

Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.) If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. It is possible that some values have already been assigned to host variables and will remain assigned when the error occurs.

expression

Specifies the new value of the host variable. The *expression* is any expression of the type described in “Expressions” on page 129. It must not include a column name.

NULL

Specifies that the new value for the host variable is the null value.

row-subselect

A subselect that returns a single result row. The result column values are assigned to each corresponding *host-variable*. If the result of the subselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

Notes

Host variable assignment: If the specified host variable is character and is not large enough to contain the result, 'W' is assigned to SQLWARN1 in the SQLCA. The actual length of the result is returned in the indicator variable associated with the host-variable, if an indicator variable is provided.

If the specified host variable is a C NUL-terminated host variable and is not large enough to contain the result and the NUL-terminator:

- If the *CNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the following occurs:
 - The result is truncated.
 - The last character is the NUL-terminator.
 - The value 'W' is assigned to SQLWARN1 in the SQLCA.
- If the *NOCNULRQD option on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) is specified, the following occurs:
 - The NUL-terminator is not returned.
 - The value 'N' is assigned to SQLWARN1 in the SQLCA.

Examples

Example 1: Assign the value of the CURRENT PATH special register to host variable HV1.

```
EXEC SQL SET :HV1 = CURRENT PATH;
```

Example 2: Assume that LOB locator LOB1 is associated with a CLOB value. Assign a portion of the CLOB value to host variable DETAILS using the LOB locator.

```
EXEC SQL SET :DETAILS = SUBSTR(:LOB1,1,35);
```

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.

There are two forms of this statement:

- The *Searched* UPDATE form is used to update one or more rows (optionally determined by a search condition).
- The *Positioned* UPDATE form is used to update exactly one row (as determined by the current position of a cursor).

Invocation

A Searched UPDATE statement can be embedded in an application program or issued interactively. A Positioned UPDATE must be embedded in an application program. Both forms are executable statements that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For the table or view identified in the statement:
 - The UPDATE privilege on the table or view, or
 - The UPDATE privilege on each column to be updated, or
 - Ownership of the table; and
 - The system authority *EXECUTE on the library containing the table or view
- Administrative authority

The authorization ID of the statement has the UPDATE privilege on a table (or the specified columns of a table) when:

- It is the owner of the table,
- It has been granted the UPDATE privilege on the table or on the columns of the table, or
- It has been granted the system authorities of *OBJOPR and *UPD on the table.

The authorization ID of the statement has the UPDATE privilege on a view (or the specified columns of a view) when:⁶⁹

- It has been granted the UPDATE privilege on the view or on the columns of the view, or
- It has been granted the system authorities of *OBJOPR and *UPD on the view and the system authority *UPD on the first table or view in the first FROM clause of the view definition; and if this is a view, then the system authority *UPD on the first table or view in the first FROM clause of that view definition; and so forth.

If the *expression* in the *assignment-clause* contains a reference to a column of the table or view, or if the *search-condition* in a Searched UPDATE contains a reference to a column of the table or view, then the privileges held by the authorization ID of the statement must also include one of the following:

69. When a view is created, the owner does not necessarily acquire the UPDATE privilege on the view. The owner only acquires the UPDATE privilege if the view allows updates and the owner also has the UPDATE privilege on the first table referenced in the subselect.

- The SELECT privilege on the table or view
- Administrative authority

The authorization ID of the statement has the SELECT privilege on a table when:

- It is the owner of the table,
- It has been granted the SELECT privilege on the table, or
- It has been granted the system authorities of *OBJOPR and *READ on the table.

The authorization ID of the statement has the SELECT privilege on a view when:

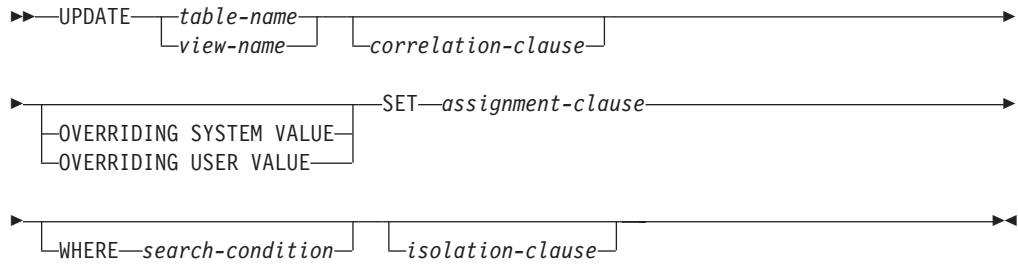
- It is the owner of the view,
- It has been granted the SELECT privilege on the view, or
- It has been granted the system authorities of *OBJOPR and *READ on the view and the system authority *READ on all tables and views that this view is directly or indirectly dependent on. That is, all tables and views referenced in the view definition, and if a view is referenced, all tables and views referenced in its definition, and so forth.

If the *search-condition* includes a subquery or if the *assignment-clause* includes a *scalar-subselect* or *row-subselect*, see Chapter 4, “Queries” on page 329 for an explanation of the authorization required for each subselect.

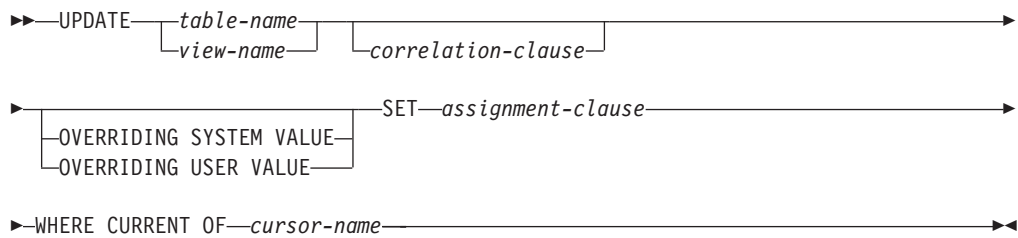
UPDATE

Syntax

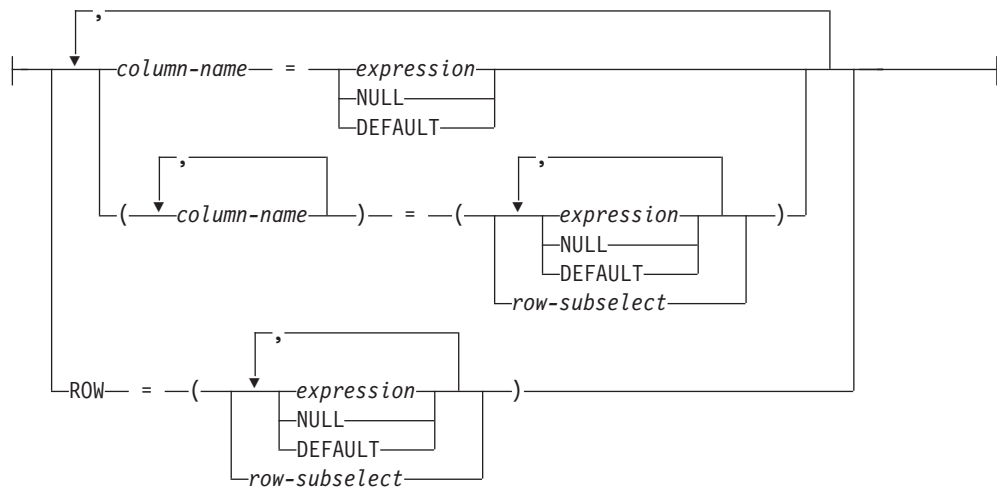
Searched UPDATE:



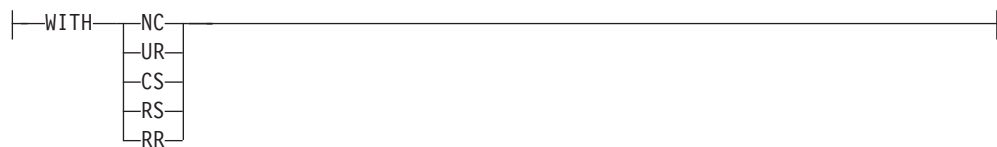
Positioned UPDATE:



assignment-clause::



isolation-clause:



Description

table-name or *view-name*

Identifies the table or view to be updated. The name must identify a table or view that exists at the current server, but it must not identify a catalog table, a view of a catalog table, or a read-only view. For an explanation of read-only views and updateable views, see “CREATE VIEW” on page 569.

correlation-clause

Can be used within *search-condition* or *assignment-clause* to designate the table or view. For an explanation of *correlation-clause*, see “table-reference” on page 335. For an explanation of *correlation-name*, see “Correlation Names” on page 108.

OVERRIDING SYSTEM VALUE or OVERRIDING USER VALUE

Specifies whether system-generated values or user-specified values for a ROWID or identity column are used. If OVERRIDING SYSTEM VALUE is specified, the implicit or explicit list of columns in the SET clause must contain a column defined as GENERATED ALWAYS. If OVERRIDING USER VALUE is specified, the implicit or explicit list of columns for the INSERT statement must contain a column defined as either GENERATED ALWAYS or GENERATED BY DEFAULT.

OVERRIDING SYSTEM VALUE

Specifies that the value specified in the SET clause for a column that is defined as GENERATED ALWAYS is used. A system-generated value is not used.

OVERRIDING USER VALUE

Specifies that the value specified in the SET clause for a column that is defined as either GENERATED ALWAYS or GENERATED BY DEFAULT is ignored. Instead, a system-generated value is used, overriding the user-specified value.

If neither OVERRIDING SYSTEM VALUE or OVERRIDING USER VALUE is specified:

- A value cannot be specified for a ROWID or identity column that is defined as GENERATED ALWAYS.
- A value can be specified for a ROWID or identity column that is defined as GENERATED BY DEFAULT. If a value is specified, that value is assigned to the column. However, a value in a ROWID column defined BY DEFAULT can be updated only if the specified value is a valid row ID value that was previously generated by DB2 UDB for OS/390 and z/OS or DB2 UDB for iSeries. When a value of an identity column defined BY DEFAULT is updated, the database manager does not verify that the specified value is a unique value for the column unless the identity column is the sole key in a unique constraint or unique index. Without a unique constraint or unique index, the database manager can guarantee unique values only among the set of system-generated values as long as NO CYCLE is in effect.

If a value is not specified the database manager generates a new value.

SET

Introduces the assignment of values to column names.

assignment-clause

column-name

Identifies a column to be updated. The *column-name* must identify a

UPDATE

column of the specified table or view, but must not identify a view column derived from a scalar function, constant, or expression. A column must not be specified more than once.

For a Positioned UPDATE:

- If the UPDATE clause was specified in the SELECT statement of the cursor, each column name in the SET list must also appear in the UPDATE clause.
- If the UPDATE clause was not specified in the SELECT statement of the cursor, the name of any updateable column may be specified.

For more information, see “update-clause” on page 354.

A view column derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same UPDATE statement.

If a list of *column-names* is specified, the number of *expressions*, NULLs, and DEFAULTS must match the number of *column-names*.

ROW

Identifies all the columns of the specified table or view. If a view is specified, none of the columns of the view may be derived from a scalar function, constant, or expression.

The number of *expressions*, NULLs, and DEFAULTS (or the number of result columns from a *row-subselect*) must match the number of columns in the row.

For a Positioned UPDATE, if the UPDATE clause was specified in the SELECT statement of the cursor, each column of the table or view must also appear in the UPDATE clause. For more information, see “update-clause” on page 351.

ROW may not be specified for a view that contains a view column derived from the same column as another column of the view, because both columns cannot be updated in the same UPDATE statement.

expression

Specifies the new value of the column. The *expression* is any expression of the type described in “Expressions” on page 129. It must not include a column function.

A *column-name* in an expression must name a column of the named table or view. For each row updated, the value of the column in the expression is the value of the column in the row before the row is updated.

NULL

Specifies the new value for a column is the null value. NULL should only be specified for nullable columns.

DEFAULT

Specifies that the default value is assigned to a column. The value that is used depends on how the column was defined, as follows:

- If the WITH DEFAULT clause is used, the default used is as defined for the column (see *default-clause* in *column-definition* in “CREATE TABLE” on page 525).
- If the WITH DEFAULT clause or the NOT NULL clause is not used, the value used is NULL.

- 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.

row-subselect

A subselect that returns a single result row. The number of result columns in the select list must match the number of *column-names* (or if ROW is specified, the number of columns in the row) specified for assignment. The result column values are assigned to each corresponding *column-name*. If the result of the subselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

The *row-subselect* may contain references to columns of the target table of the UPDATE statement. For each row updated, the value of such a column in the expression is the value of the column in the row before the row is updated.

WHERE

Specifies the rows to be updated. The clause can be omitted, or a *search-condition* or *cursor-name* can be specified. If the clause is omitted, all rows of the table or view are updated.

search-condition

Is any search described in “Search Conditions” on page 161. 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 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. The updated rows are those for which the results of the *search-condition* are 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 of that subquery used in applying the *search-condition*. In actuality, a subquery with no correlated references may be executed only once. 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 576.

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 576.

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

isolation-clause

Specifies the isolation level to be used for this statement.

WITH

Introduces the isolation level, which may be one of:

- RR Repeatable read
- RS Read stability

UPDATE

- CS Cursor stability
- UR Uncommitted read
- NC No commit

If *isolation-clause* is not specified the default isolation is used. See “*isolation-clause*” on page 357 for a description of how the default is determined.

UPDATE Rules

Assignment: Update values are assigned to columns in accordance with the storage assignment rules described in “Assignments and Comparisons” on page 80.

Validity: Updates must obey the following rules. If they do not, or if any other errors occur during the execution of the UPDATE statement, no rows are updated.

- *Subselects:* The *row-subselect* or *scalar-subselect* shall return no more than one row (SQLSTATE 21000).
- *Unique constraints and unique indexes:* If the identified table, or the base table of the identified view, has one or more unique indexes or unique constraints, each row update in the table must conform to the limitations imposed by those indexes and constraints (SQLSTATE 23505).

All uniqueness checks are effectively made at the end of the statement. In the case of a multiple-row update of a column involved in a unique index or unique constraint, this would occur after all rows were updated.

- *Check constraints:* If the identified table, or the base table of the identified view, has one or more check constraints, each check constraint must be true or unknown for each row updated in the table (SQLSTATE 23513).

All check constraints are effectively validated at the end of the statement. In the case of a multiple-row update, this would occur after all rows were updated.

- *Views and the WITH CHECK OPTION:* If a view is identified, the updated rows must conform to any applicable WITH CHECK OPTION (SQLSTATE 44000). For more information, see “CREATE VIEW” on page 569.

Triggers: If the identified table or the base table of the identified view has an update trigger, the trigger is activated. A trigger might cause other statements to be executed or raise error conditions based on the updated values.

Referential Integrity: The value of the parent key in a parent row must not be changed.

If the update values produce a foreign key that is nonnull, the foreign key must be equal to some value of the parent key of the parent table of the relationship.

The referential constraints (other than a referential constraint with a RESTRICT delete rule) are effectively checked at the end of the statement. In the case of a multiple-row update, this would occur after all rows were updated.

Notes

Update operation errors: If an update value violates any constraints, or if any other error occurs during the execution of the UPDATE statement and COMMIT(*NONE) was not specified, all changes made during the execution of the statement are backed out. However, other changes in the unit of work made prior to the error are not backed out. If COMMIT(*NONE) is specified, changes are not backed out.

It is possible for an error to occur that makes the state of the cursor unpredictable.

Number of rows updated: When an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows updated. For a description of the SQLCA, see Appendix B, “SQL Communication Area” on page 825.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful UPDATE statement. Until these locks are released by a commit or rollback operation, the updated rows can only be accessed by:

- The application process that performed the update.
- Another application process using COMMIT(*NONE) or COMMIT(*CHG) through a read-only cursor, SELECT INTO statement, or subquery.

The locks can prevent other application processes from performing operations on the table. For further information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements, and isolation levels in “Isolation Level” on page 21. Also, see the Database Programming book.

A maximum of 500 000 000 rows can be updated or changed in any single UPDATE statement when COMMIT(*RR), COMMIT(*ALL), COMMIT(*CS), or COMMIT(*CHG) has been specified. The number of rows changed includes any rows inserted, updated, or deleted under the same commitment definition as a result of a trigger.

REXX: Host variables cannot be used in the UPDATE statement within a REXX procedure. Instead, the UPDATE must be the object of a PREPARE and EXECUTE using parameter markers.

Datalinks: 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

Syntax alternatives: The following keywords are synonyms supported for compatibility to prior releases. These keywords are non-standard and should not be used:

- The keyword NONE can be used as a synonym for NC.
- The keyword CHG can be used as a synonym for UR.
- The keyword ALL can be used as a synonym for RS.

UPDATE

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 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'
```

Example 4: In a Java program display the rows from the EMPLOYEE table on the connection context 'ctx' and then, if requested to do so, change the job (JOB) of certain employees to the new job keyed in (NEWJOB).

```
#sql iterator empIterator implements sqlj.runtime.ForUpdate
with( updateColumns='JOB' )
( ... );
empIterator C1;

#sql [ctx] C1 = { SELECT * FROM EMPLOYEE };

#sql { FETCH :C1 INTO ... };
while ( !C1.endFetch() ) {
    System.out.println( ... );
    ...
    if ( condition for updating row ) {
        #sql [ctx] { UPDATE EMPLOYEE
                     SET JOB = :NEWJOB
                     WHERE CURRENT OF :C1 };
    }

    #sql { FETCH :C1 INTO ... };
}
C1.close();
```

VALUES

The VALUES statement provides a method for invoking a user-defined function from a trigger. Transition variables can be passed to the user-defined function.

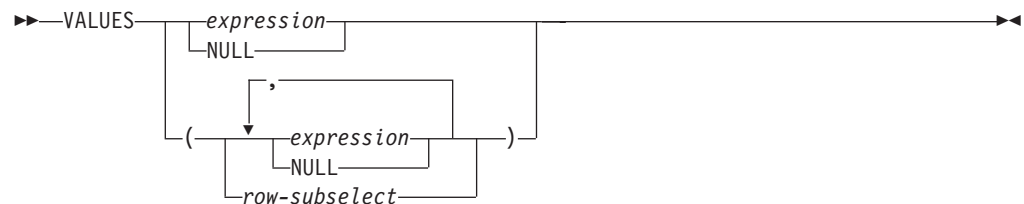
Invocation

This statement can only be used in the triggered action of a CREATE TRIGGER statement.

Authorization

If a *row-subselect* is specified, see Chapter 4, “Queries” on page 329 for an explanation of the authorization required for each subselect.

Syntax



Description

VALUES

Introduces a single row consisting of one or more columns.

expression

Any expression of the type described in “Expressions” on page 129.

NULL

Specifies the null value.

row-subselect

A subselect that returns a single result row. If the result of the subselect is no rows, then null values are returned. An error is returned if there is more than one row in the result.

Notes

Effects of the Statement: The statement is evaluated, but the resulting values are discarded and are not assigned to any output variables. If an error is returned, the database manager stops executing the trigger and rolls back any triggered actions that were performed (unless the trigger is running under isolation level *NONE).

Examples

Create an after trigger EMPISRT1 that invokes user-defined function NEWEMP when the trigger is activated. An insert operation on table EMP activates the trigger. Pass transition variables for the new employee number, last name, and first name to the user-defined function.

```

CREATE TRIGGER EMPISRT1
  AFTER INSERT ON EMPLOYEE
  REFERENCING NEW AS N
  FOR EACH ROW

```

VALUES

```
MODE DB2SQL
BEGIN ATOMIC
  VALUES( NEWEMP(N.EMPNO, N.LASTNAME, N.FIRSTNAME));
END
```


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.

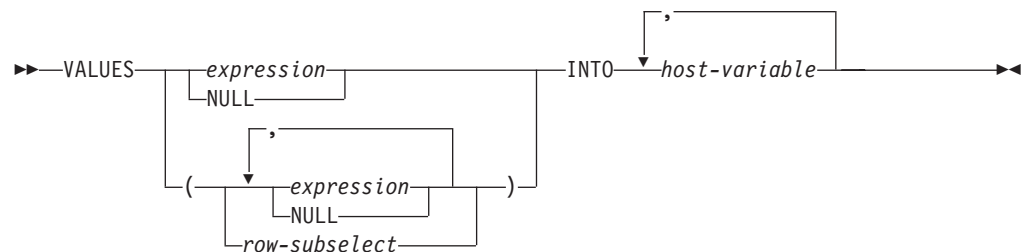
Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared, but cannot be issued interactively.

Authorization

If a *row-subselect* is specified, see Chapter 4, “Queries” on page 329 for an explanation of the authorization required for each subselect.

Syntax



Description

VALUES

Introduces a single row consisting of one or more columns.

expression

Specifies the new value of the host variable. The *expression* is any expression of the type described in “Expressions” on page 129. It must not include a column name. Host structures are not supported.

NULL

Specifies that the new value for the host variable is the null value.

row-subselect

A subselect that returns a single result row. The result column values are assigned to each corresponding *host-variable*. If the result of the subselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

INTO *host variable,...*

Identifies one or more host structures or variables that must be declared in the program in accordance with the rules for declaring host structures and variables. In the operational form of INTO, a reference to a host structure is replaced by a reference to each of its variables. The first value specified is assigned to the first host variable, the second value to the second host variable, and so on.

Notes

Host variable assignment: Each assignment to a host variable is performed according to the retrieval assignment rules described in “Assignments and

VALUES INTO

Comparisons” on page 80. Assignments are made in sequence through the list. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. See Appendix B, “SQL Communication Area” on page 825. Note that there is no warning if there are more variables than values. If a value is null, an indicator variable must be provided for that value.

If the specified host variable is character and is not large enough to contain the result, 'W' is assigned to SQLWARN1 in the SQLCA. The actual length of the result may be returned in the indicator variable associated with the host-variable, if an indicator variable is provided. For further information, see “References to Variables” on page 114.

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.

If the specified host variable is a C NUL-terminated host variable and is not large enough to contain the result and the NUL-terminator:

- If the *CNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*YES) on the SET OPTION statement), the following occurs:
 - The result is truncated.
 - The last character is the NUL-terminator.
 - The value 'W' is assigned to SQLWARN1 in the SQLCA.
- If the *NOCNULRQD option on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) is specified, the following occurs:
 - The NUL-terminator is not returned.
 - The value 'N' is assigned to SQLWARN1 in the SQLCA.

Result column evaluation considerations: If an error occurs while evaluating a result column in the expression list of a VALUES INTO statement as the result of an arithmetic expression (such as division by zero, or overflow) or a numeric or character conversion error, the result is the null value. As in any other case of a null value, an indicator variable must be provided. The value of the host variable is undefined. In this case, however, the indicator variable is set to the value of -2. Processing of the statement continues and a warning is returned. If an indicator variable is not provided, an error is returned and no more values are assigned to variables. It is possible that some values have already been assigned to host variables and will remain assigned when the error is returned.

When a datetime value is returned, the length of the variable must be large enough to store the complete value. Otherwise, depending on how much of the value would have to be truncated, a warning or error is returned. See “Datetime Assignments” on page 85 for details.

Examples

Example 1: Assign the value of the CURRENT PATH special register to host variable HV1.

```
EXEC SQL VALUES CURRENT PATH
      INTO :HV1;
```

VALUES INTO

Example 2: Assume that LOB locator LOB1 is associated with a CLOB value. Assign a portion of the CLOB value to host variable DETAILS using the LOB locator, and assign CURRENT TIMESTAMP to the host variable TIMETRACK.

```
EXEC SQL VALUES (SUBSTR(:LOB1,1,35), CURRENT TIMESTAMP)  
        INTO :DETAILS, :TIMETRACK;
```

WHENEVER

WHENEVER

The WHENEVER statement specifies the action to be taken when a specified exception condition occurs.

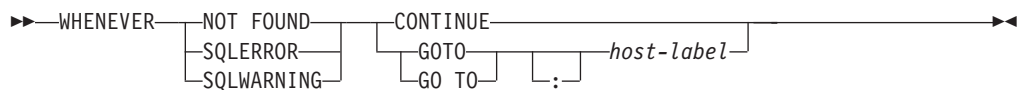
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX. See the SQL Programming with Host Languages book for information on handling errors 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 SQLSTATE of '02000' or an SQLCODE of +100.

SQLERROR

Identifies any condition that results in an SQLSTATE value where the first two characters are not '00', '01', or '02'.

SQLWARNING

Identifies any condition that results in an SQLSTATE value where the first two characters are '01', or a warning condition (SQLWARN0 is 'W').

The CONTINUE or GO TO clause are used to specify the next statement to be executed when the identified type of exception condition exists.

CONTINUE

Specifies the next sequential instruction of the source program.

GOTO or GO TO *host-label*

Specifies the statement identified by *host-label*. For *host-label*, substitute a single token, optionally preceded by a colon. The form of the token depends on the host language. In a COBOL program, for example, it can be a *section-name* or an unqualified *paragraph-name*.

Notes

WHENEVER statement types: There are three types of WHENEVER statements:

WHENEVER NOT FOUND

WHENEVER SQLERROR

WHENEVER SQLWARNING

WHENEVER statement scope: 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.

SQL does support nested programs in COBOL, C, and RPG. However, SQL does not honor normal COBOL, C, and RPG scoping rules. That is, the last WHENEVER statement specified in the program source prior to the nested procedure is still in effect for that nested program. The label referenced in the WHENEVER statement must be duplicated within that inner program. Alternatively, the inner program could specify a new WHENEVER statement.

In FORTRAN, the scope of a WHENEVER statement is limited to SQL statements within the same subprogram.

Example

The following statements can be embedded in a COBOL program.

Example 1: Go to the label HANDLER for any statement that produces an error.

```
EXEC SQL  WHENEVER SQLERROR GOTO HANDLER  END-EXEC.
```

Example 2: Continue processing for any statement that produces a warning.

```
EXEC SQL  WHENEVER SQLWARNING GOTO CONTINUE  END-EXEC.
```

Example 3: Go to the label ENDDATA for any statement that does not return data when expected to do so.

```
EXEC SQL  WHENEVER NOT FOUND GOTO ENDDATA  END-EXEC.
```

WHENEVER

Chapter 6. SQL Control Statements

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

SQL-control-statement:

<i>assignment-statement</i>
<i>CALL-statement</i>
<i>CASE-statement</i>
<i>compound-statement</i>
<i>FOR-statement</i>
<i>GET DIAGNOSTICS-statement</i>
<i>GOTO-statement</i>
<i>IF-statement</i>
<i>ITERATE-statement</i>
<i>LEAVE-statement</i>
<i>LOOP-statement</i>
<i>REPEAT-statement</i>
<i>RESIGNAL-statement</i>
<i>RETURN-statement</i>
<i>SIGNAL-statement</i>
<i>WHILE-statement</i>

Control statements are supported in SQL procedures, SQL functions, and SQL triggers.

SQL procedures are created by specifying LANGUAGE SQL and an SQL routine body on the CREATE PROCEDURE statement. SQL functions are created by specifying LANGUAGE SQL and an SQL routine body on the CREATE FUNCTION statement. SQL routines are SQL procedures or SQL functions. SQL triggers are created by specifying an SQL routine body on the CREATE TRIGGER statement.

An SQL routine body must be a single SQL statement which may be an SQL control statement.

The SQL routine body is the executable part of the procedure, function, or trigger that is transformed by the database manager into a program or service program. When an SQL routine or trigger is created, SQL creates a temporary source file (QTEMP/QSQLSRC) that will contain C source code with embedded SQL statements. If DBGVIEW(*SOURCE) is specified, SQL creates the root source for the routine or trigger in temporary source file QTEMP/QSQDSRC.

An SQL procedure or SQL trigger is created as a program (*PGM) object using the CRTPGM command. An SQL function is created as a service program (*SRVPGM) object using the CRTSRVPGM command. The program or service program is created in the library that is the implicit or explicit qualifier of the procedure, function, or trigger name.

SQL control statements

When the program or service program is created, the SQL statements other than control statements become embedded SQL statements in the program or service program.

The specified procedure or function is registered in the SYSROUTINES and SYSPARMS catalog tables, and an internal link is created from SYSROUTINES to the program. When the procedure is called using the SQL CALL statement or when the function is invoked in an SQL statement, the program associated with the routine is called. The specified SQL trigger is registered in the SYSTRIGGER catalog table.

The remainder of this chapter contains a description of the control statements including syntax diagrams, semantic descriptions, usage notes, and examples of the use of the statements that constitute the SQL routine body. There is also a section on referencing SQL parameters and variables found in “References to SQL Parameters and Variables” on page 781. There are two common elements that are used in describing specific SQL control statements. These are:

- SQL control statements as described above
- “SQL procedure statement” on page 782

For syntax and additional information on the SQL control statements see the following topics:

- “assignment-statement” on page 784
- “CALL statement” on page 786
- “CASE statement” on page 788
- “compound-statement” on page 790
- “IF statement” on page 804
- “FOR statement” on page 797
- “GET DIAGNOSTICS statement” on page 799
- “GOTO statement” on page 802
- “ITERATE statement” on page 806
- “LEAVE statement” on page 807
- “LOOP statement” on page 808
- “REPEAT statement” on page 809
- “RESIGNAL statement” on page 811
- “RETURN statement” on page 814
- “SIGNAL statement” on page 816
- “WHILE statement” on page 819

References to SQL Parameters and Variables

SQL parameters and SQL variables can be referenced anywhere in an SQL procedure statement where an expression or host variable can be specified. Host variables cannot be specified in SQL routines. SQL parameters can be referenced anywhere in the routine and can be qualified with the routine name. SQL variables can be referenced anywhere in the compound statement in which they are declared and can be qualified with the label name specified at the beginning of the compound statement.

All SQL parameters and SQL variables are considered nullable except SQL variables that are explicitly declared as NOT NULL. The name of an SQL parameter or SQL variable in an SQL routine can be the same as the name of a column in a table or view referenced in the routine. In this case, the name should be explicitly qualified to indicate whether it is a column, SQL variable, or SQL parameter.

If the name is not qualified, the following rules describe whether the name refers to the column or to the SQL variable or SQL parameter:

- If the tables and views specified in an SQL routine body exist at the time the routine is created, the name will first be checked as a column name. If not found as a column, it will then be checked as an SQL variable name in the compound, and then checked as an SQL parameter name.
- If the referenced tables or views do not exist at the time the routine is created, the name will first be checked as an SQL variable name and then as an SQL parameter name. If not found, it will be assumed to be a column.

The name of an SQL parameter or SQL variable in an SQL routine can be the same as the name of an identifier used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to the identifier or to the SQL parameter or SQL variable:

- In the SET PATH and SET SCHEMA statements, the name is checked as an SQL parameter name or SQL variable name. If not found as an SQL variable or SQL parameter name, it will then be used as an identifier.
- In the CONNECT statement, the name is used as an identifier.

SQL procedure statement

An SQL control statement may allow multiple SQL statements to be specified within the SQL control statement. These statements are defined as SQL procedure statements.

Syntax

<i>SQL-control-statement</i>	(1)
<i>ALTER-statement</i>	
<i>CLOSE-statement</i>	
<i>COMMENT-statement</i>	
<i>COMMIT-statement</i>	
<i>CONNECT-statement</i>	
<i>CREATE ALIAS-statement</i>	
<i>CREATE DISTINCT TYPE-statement</i>	
<i>CREATE FUNCTION (External Scalar)-statement</i>	
<i>CREATE FUNCTION (External Table)-statement</i>	
<i>CREATE FUNCTION (Sourced)-statement</i>	
<i>CREATE INDEX-statement</i>	
<i>CREATE PROCEDURE (External)-statement</i>	
<i>CREATE SCHEMA-statement</i>	
<i>CREATE TABLE-statement</i>	
<i>CREATE VIEW-statement</i>	
<i>DECLARE GLOBAL TEMPORARY TABLE-statement</i>	
<i>DELETE-statement</i>	
<i>DISCONNECT-statement</i>	
<i>DROP-statement</i>	
<i>EXECUTE-statement</i>	
<i>EXECUTE IMMEDIATE-statement</i>	
<i>FETCH-statement</i>	
<i>GRANT-statement</i>	
<i>INSERT-statement</i>	
<i>LABEL-statement</i>	
<i>LOCK TABLE-statement</i>	
<i>OPEN-statement</i>	
<i>PREPARE-statement</i>	
<i>RELEASE-statement</i>	
<i>RENAME-statement</i>	
<i>REVOKE-statement</i>	
<i>ROLLBACK-statement</i>	
<i>SELECT INTO-statement</i>	
<i>SET CONNECTION-statement</i>	
<i>SET PATH-statement</i>	
<i>SET RESULT SETS-statement</i>	
<i>SET SCHEMA-statement</i>	
<i>SET TRANSACTION-statement</i>	
<i>UPDATE-statement</i>	

Notes:

- 1 COMMIT, ROLLBACK, CONNECT, DISCONNECT, SET CONNECTION, and SET RESULT SETS statements are only allowed in SQL procedures. The SET TRANSACTION statement is allowed in SQL procedures and triggers.

Notes

Comments: Comments can be included within the body of an SQL procedure. In addition to the double-dash form of comments (--), a comment can begin with /* and end with */. The following rules apply to this form of a comment.

- The beginning characters /* must be on the same line.
- The ending characters */ must be on the same line.
- Comments can be started wherever a space is valid.
- Comments can be continued to the next line.

assignment-statement

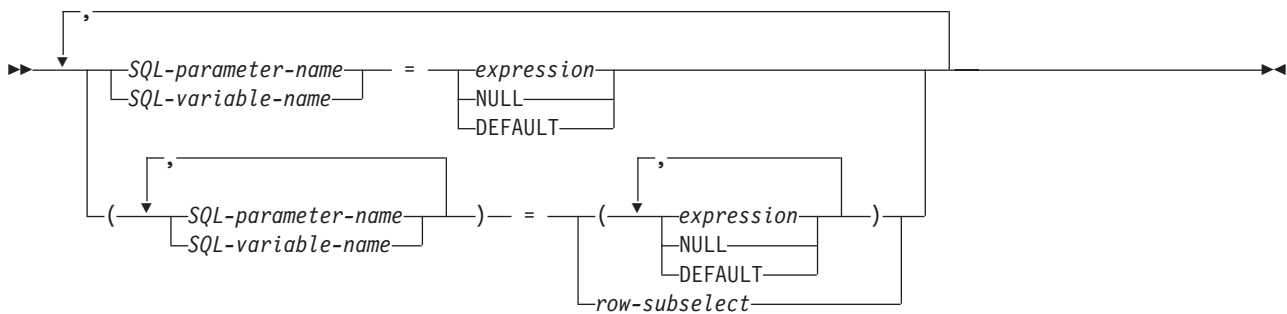
assignment-statement

The assignment-statement assigns a value to an SQL parameter or SQL variable.

Syntax



assignment-clause:



Description

label

Specifies the label for the *assignment-statement* statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

SQL-parameter-name

Identifies the SQL parameter that is the assignment target. The SQL parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE or CREATE FUNCTION statement.

SQL-variable-name

Identifies the SQL variable that is the assignment target. SQL variables can be defined in a compound statement or be a transition variable.

expression or NULL

Specifies the expression or value that is the source for the assignment.

DEFAULT

Specifies that the default value for the column associated with the transition variable will be used. This can only be specified in SQL triggers for transition variables.

row-subselect

A subselect that returns a single result row. The result column values are assigned to the corresponding SQL variable or parameter. If the result of the subselect is no rows, then null values are assigned. An error is returned if there is more than one row in the result.

Notes

Assignment rules: Assignments in the assignment statement must conform to the SQL assignment rules as described in “Assignments and Comparisons” on page 80. If assigning to a string variable, storage assignment rules apply.

Assignment rules for SQL parameters: An IN parameter can appear on the left or right side of an assignment-statement. When control returns to the caller, the original value of the IN parameter is retained. An OUT parameter can also appear on the left or right side of an assignment-statement. If used without first being assigned a value, the value is undefined. When control returns to the caller, the last value that is assigned to an OUT parameter is returned to the caller. For an INOUT parameter, the first value of the parameter is determined by the caller, and the last value that is assigned to the parameter is returned to the caller.

Special Registers: If a variable has been declared with an identifier that matches the name of a special register (such as PATH), then the variable must be delimited to distinguish it from assignment to the special register (for example, SET "PATH" = 1; for a variable called PATH declared as an integer).

SQLCODE and SQLSTATE: If the target of the assignment is a variable and source is a variable or constant, the assignment may be performed inline. In this case, the SQLCODE and SQLSTATE will not be reset.

Example

Increase the SQL variable p_salary by 10 percent.

```
SET p_salary = p_salary + (p_salary * .10)
```

Set SQL variable p_salary to the null value

```
SET p_salary = NULL
```

CALL statement

The CALL statement invokes a procedure. The syntax of CALL in an SQL procedure is a subset of what is supported as a CALL statement in other contexts. See "CALL" on page 396 for details.

Syntax

```
CALL procedure-name argument-list
```

Diagram showing the syntax: CALL *procedure-name* *argument-list*. A bracket labeled *label:* is positioned below the CALL keyword.

argument-list:

```
( SQL-variable-name
  SQL-parameter-name
  constant
  NULL
  special-register
  ,
  )
```

Diagram showing the argument list syntax: (*SQL-variable-name*, *SQL-parameter-name*, *constant*, NULL, *special-register*).

Description

label

Specifies the label for the CALL statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

procedure-name

Identifies the procedure to call. The *procedure-name* must identify a procedure that exists at the current server.

argument-list

Specifies the arguments of the procedure. The number of arguments specified must be the same as the number of parameters defined by that procedure.

SQL-variable-name

Specifies an SQL variable as an argument to the procedure.

SQL-parameter-name

Specifies an SQL parameter as an argument to the procedure.

constant

Specifies a constant as an argument to the procedure.

NULL

Specifies the null value as an argument to the procedure.

special-register

Specifies a special register as an argument to the procedure.

Notes

Rules for arguments to OUT and INOUT parameters: Each OUT or INOUT parameter must be specified as an SQL parameter or SQL variable.

Special registers: The initial value of a special register in a procedure is inherited from the caller of the procedure. A value assigned to a special register within the procedure is used for the entire SQL procedure and will be inherited by any procedure subsequently called from that procedure. When a procedure returns to its caller, the special registers are restored to the original values of the caller.

Related information: See “CALL” on page 396 for more information.

Example

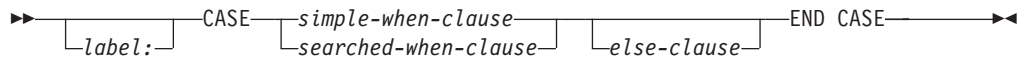
Call procedure *proc1* and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

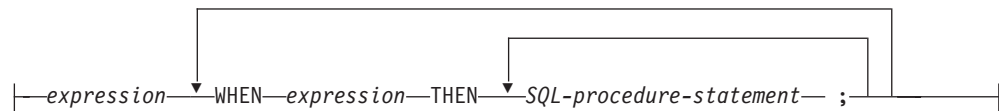
CASE statement

The CASE statement selects an execution path based on multiple conditions.

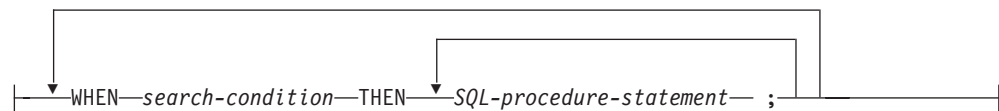
Syntax



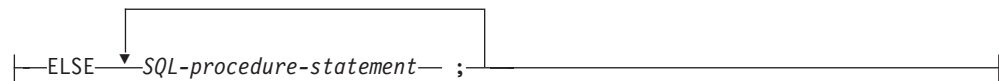
simple-when-clause:



searched-when-clause:



else-clause:



Description

label

Specifies the label for the CASE statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

simple-when-clause

The value of the *expression* prior to the first WHEN keyword is tested for equality with the value of each *expression* that follows the WHEN keyword. If the comparison is true, the THEN statement is executed. If the result is unknown or false, processing continues to the next comparison. If the result does not match any of the comparisons, and an ELSE clause is present, the statements in the ELSE clause are processed.

searched-when-clause

The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are processed. If it evaluates to false, or unknown, the next *search-condition* is evaluated. If no *search-condition* evaluates to true and an ELSE clause is present, the statements in the ELSE clause are processed.

else-clause

If none of the conditions specified in the *simple-when-clause* or *searched-when-clause* are true, then the statements in the *else-clause* are executed.

If none of the conditions specified in the WHEN are true, and an ELSE clause is not specified, an error is issued at runtime, and the execution of the CASE statement is terminated (SQLSTATE 20000).

SQL-procedure-statement

Specifies a statement that should be executed. See “SQL procedure statement” on page 782.

Notes

Nesting of CASE statements: CASE statements that use a *simple-when-clause* can be nested up to three levels. CASE statements that use a *searched-when-clause* have no limit to the number of nesting levels.

Examples

Example 1: Depending on the value of SQL variable `v_workdept`, update column `DEPTNAME` in table `DEPARTMENT` with the appropriate name.

The following example shows how to do this using the syntax for a *simple-when-clause*.

```
CASE v_workdept
  WHEN 'A00'
    THEN UPDATE department SET
      deptname = 'DATA ACCESS 1';
  WHEN 'B01'
    THEN UPDATE department SET
      deptname = 'DATA ACCESS 2';
  ELSE UPDATE department SET
      deptname = 'DATA ACCESS 3';
END CASE
```

Example 2: The following example shows how to do this using the syntax for a *searched-when-clause*:

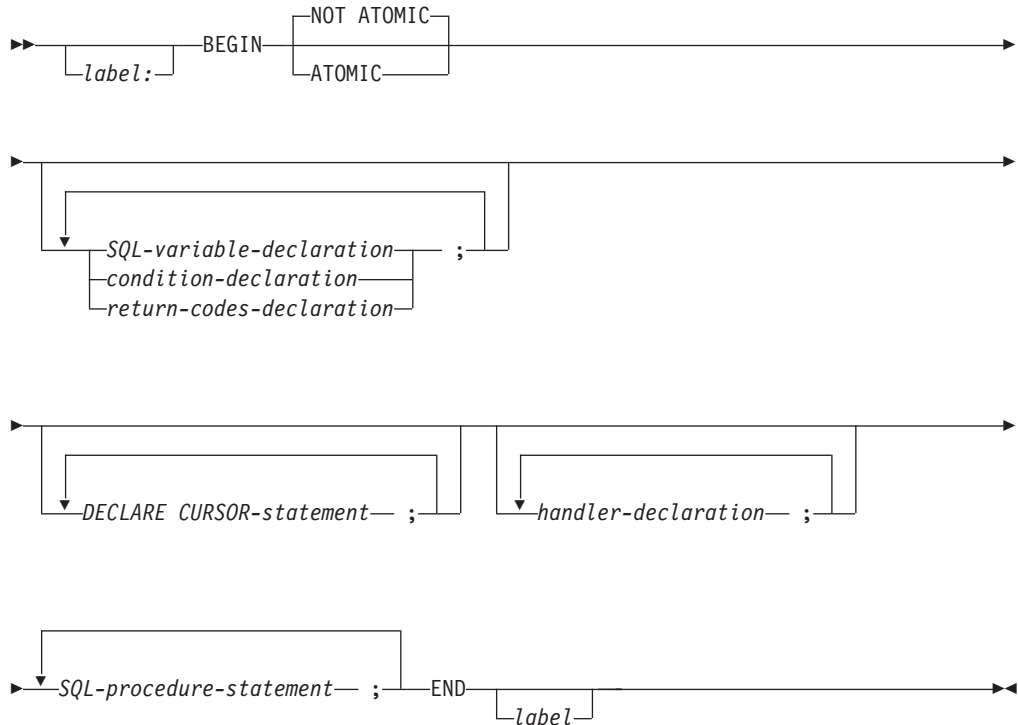
```
CASE
  WHEN v_workdept = 'A00'
    THEN UPDATE department SET
      deptname = 'DATA ACCESS 1';
  WHEN v_workdept = 'B01'
    THEN UPDATE department SET
      deptname = 'DATA ACCESS 2';
  ELSE UPDATE department SET
      deptname = 'DATA ACCESS 3';
END CASE
```

compound-statement

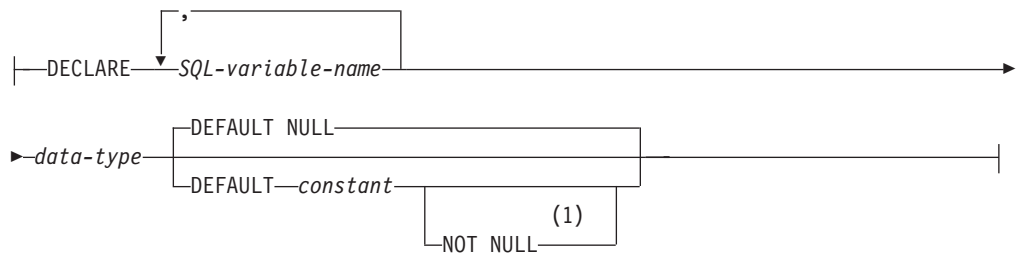
compound-statement

A compound statement groups other statements together in an SQL procedure. A compound statement allows the declaration of SQL variables, cursors, and condition handlers.

Syntax



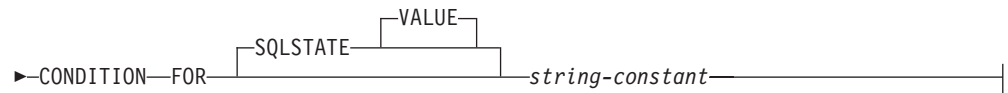
SQL-variable-declaration:



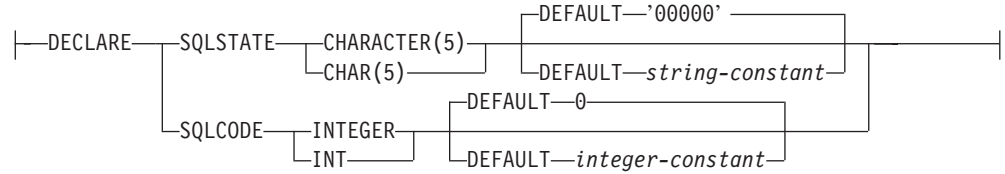
condition-declaration:



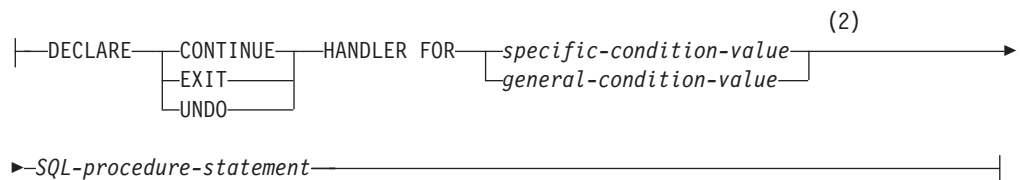
compound-statement



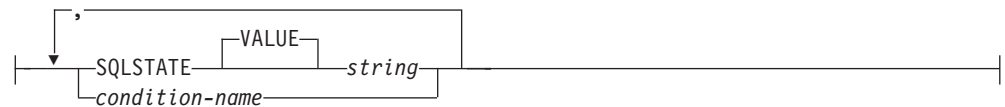
return-codes-declaration:



handler-declaration:



specific-condition-value:



general-condition-value:



data-type:

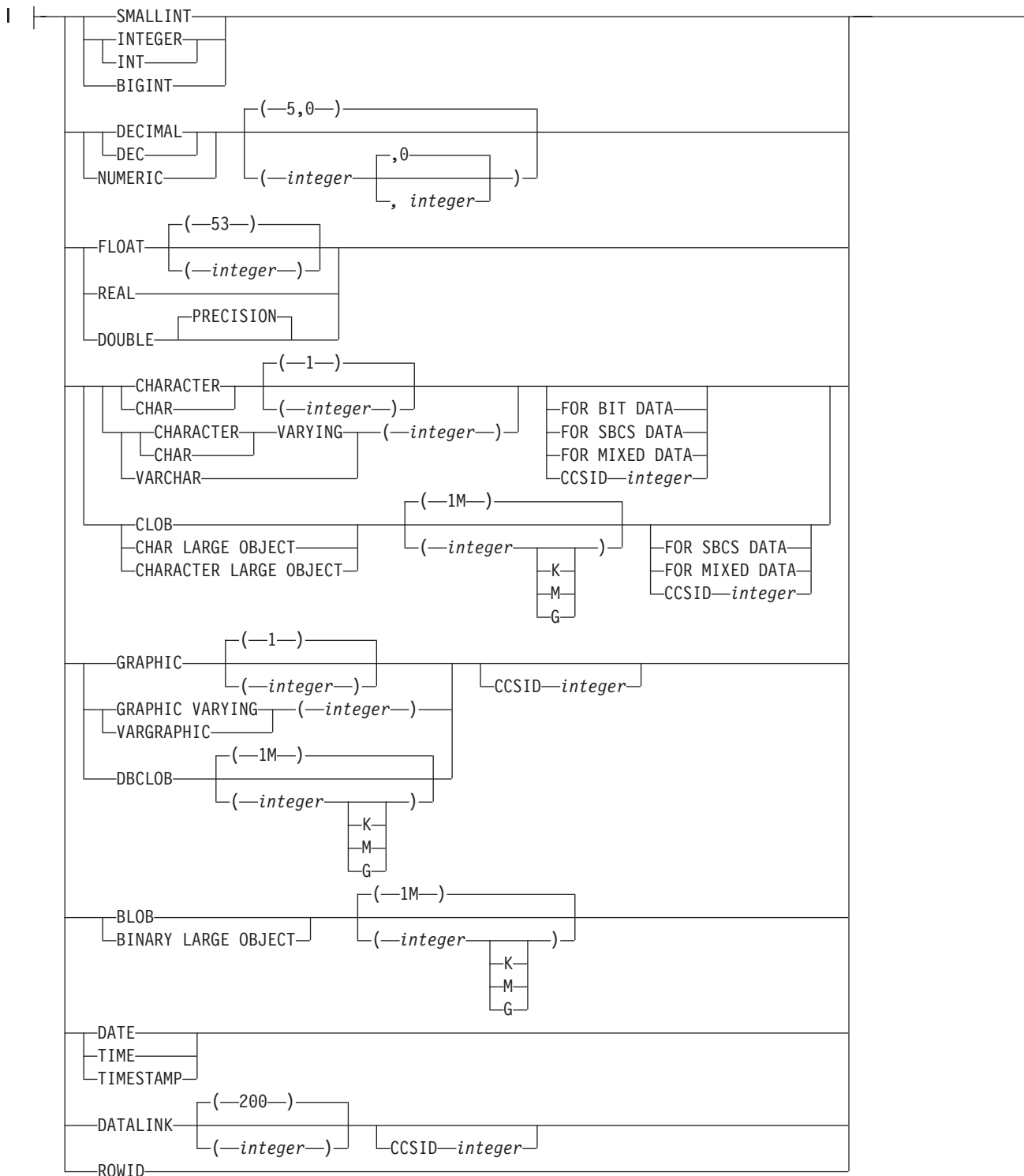


Notes:

- 1 The DEFAULT and NOT NULL clauses can be specified in either order.
- 2 *specific-condition-value* and *general-condition-value* cannot be specified in the same handler declaration.

compound-statement

built-in-type:



Description

label

Specifies the label for the *compound-statement* statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

ATOMIC

ATOMIC indicates that an unhandled within the *compound-statement* causes the *compound-statement* to be rolled back. If ATOMIC is specified, COMMIT or ROLLBACK statements cannot be specified in the compound statement (ROLLBACK TO SAVEPOINT may be specified).

NOT ATOMIC

NOT ATOMIC indicates that an unhandled within the *compound-statement* does not causes the *compound-statement* to be rolled back. If NOT ATOMIC is specified in the outermost compound statement of an SQL trigger, it is treated as ATOMIC.

SQL-variable-declaration

Declares a variable that is local to the compound statement.

SQL-variable-name

Defines the name of a local variable. The database manager converts all un delimited SQL variable names to uppercase. The *SQL-variable-name* must be unique within the *compound-statement* (excluding any declarations in *compound-statements* nested within the *compound-statement*). SQL variable names should not be the same as column names or SQL parameter names. See “References to SQL Parameters and Variables” on page 781 for how SQL variable names are resolved when there are columns with the same name involved in a statement. Variable names should not begin with ‘SQL’.

An *SQL-variable-name* can only be referenced within the *compound-statement* in which it is declared (including any *compound-statements* nested within the *compound-statement*).

data-type

Specifies the data type of the variable. See “CREATE TABLE” on page 525 for a description of data type.

If the *data-type* is a graphic string data type, consider specifying CCSID 13488 to indicate UCS-2 data. If a CCSID is not specified, the CCSID of the graphic string variable will be the associated DBCS CCSID for the job.

DEFAULT *constant* or NULL

Defines the default for the SQL variable. The variable will be initialized when the SQL procedure, SQL function, or SQL trigger is invoked. If a default value is not specified, the SQL variable is initialized to NULL.

NOT NULL

Prevents the SQL variable from containing the NULL value. Omission of NOT NULL implies that the column can be null.

condition-declaration

Declares a condition name and corresponding SQLSTATE value.

condition-name

Specifies the name of the condition. The condition name must be unique

compound-statement

within the *compound-statement* (excluding any declarations in *compound-statements* nested within the *compound-statement*).

A *condition-name* can only be referenced within the *compound-statement* in which it is declared (including any *compound-statements* nested within the *compound-statement*).

FOR SQLSTATE *string-constant*

Specifies the SQLSTATE associated with this condition. The string constant must be specified as 5 characters, and cannot be '00000'.

return-codes-declaration

Declares special variables called SQLSTATE and SQLCODE that are set automatically to the SQL return codes returned after executing an SQL statement. Both the SQLSTATE and SQLCODE variables can only be declared in the outermost *compound-statement* of an SQL procedure, SQL function, or SQL trigger.

Assignment to these variables is not prohibited. However, the assignment will not be useful since the next SQL statement will replace the assigned value. The SQLCODE and SQLSTATE variables cannot be set to NULL.

SQLCODE and SQLSTATE variables should be saved immediately to another SQL variable if there is any intention to use the values. If a handler exists for the SQLSTATE, this assignment must be the first statement in the handler to avoid having the value replaced by the next SQL procedure statement.

DECLARE CURSOR-*statement*

Declares a cursor in the routine body. The cursor name must be unique within the *compound-statement* (excluding any declarations in *compound-statements* nested within the *compound-statement*).

A *cursor-name* can only be referenced within the *compound-statement* in which it is declared (including any *compound-statements* nested within the *compound-statement*).

Use an OPEN statement to open the cursor, and a FETCH statement to read rows using the cursor. If the cursor in an SQL procedure and is intended for use as a result set:

- specify WITH RETURN when declaring the cursor
- create the procedure using the DYNAMIC RESULT SETS clause with a non-zero value
- do not specify a CLOSE statement in the *compound-statement*.

Any open cursor that does not meet these criteria is closed at the end of the *compound-statement*.

For more information on declaring a cursor, refer to "DECLARE CURSOR" on page 576.

handler-declaration

Specifies a *handler*, an *SQL-procedure-statement* to execute when an exception or completion condition occurs in the *compound-statement*.

A handler is active for the set of *SQL-procedure-statements* that follow the *handler-declarations* within the *compound-statement* in which it is declared.

There are three types of condition handlers:

CONTINUE

After the handler is invoked successfully, control is returned to the SQL statement following the one that raised the exception. If the error occurs

compound-statement

while executing a comparison as in an IF, CASE, FOR, WHILE, or REPEAT, control returns to the statement following the corresponding END IF, END CASE, END FOR, END WHILE, or END REPEAT.

EXIT

Once the handler is invoked successfully, control is returned to the end of the compound statement that declared the handler.

UNDO

ROLLBACK the changes made by the *compound-statement* and invoke the handler. Once the handler is invoked successfully, control is returned to the end of the *compound-statement*. If UNDO is specified, then ATOMIC must be specified.

UNDO cannot be specified in the outermost *compound-statement* of an SQL function or SQL trigger.

The conditions under which the handler is activated are:

SQLSTATE *string*

Specifies that the handler is invoked when the specific SQLSTATE condition occurs. The first two characters of the SQLSTATE value cannot be '00'.

condition-name

Specifies that the handler is invoked when the condition occurs. The condition name must be previously defined in a *condition-declaration*.

SQLException

Specifies that the handler is invoked when an exception condition occurs. An exception condition is represented by an SQLSTATE value where the first two characters are not '00', '01', or '02'.

SQLWARNING

Specifies that the handler is invoked when a warning condition occurs. A warning condition is represented by an SQLSTATE value where the first two characters are '01'.

NOT FOUND

Specifies that the handler is invoked when a NOT FOUND condition occurs. A NOT FOUND condition is represented by an SQLSTATE value where the first two characters are '02'.

The same condition cannot be specified more than once in the *handler-declaration*.

Notes

Nesting compound statements: Compound statements can be nested.

Rules for *handler-declaration*:

- Handler declarations within the same compound statement cannot contain duplicate conditions.
- A handler declaration cannot contain the same condition value or SQLSTATE value more than once, and cannot contain a SQLSTATE value and a condition name that represents the same SQLSTATE value. For a list of SQLSTATE values as well as more information, see the SQL Programming Concepts book.
- A handler is activated when it is the most appropriate handler for an exception or completion condition. The most appropriate handler is a handler (for the exception or completion condition) that is defined in the *compound-statement*

compound-statement

| which most closely matches the SQLSTATE of the exception or completion
| condition. For example, if a handler exists for SQLSTATE 22001 as well as a
| handler for SQLEXCEPTION, the handler for SQLSTATE 22001 would be the
| most appropriate handler when an SQLSTATE 22001 is signalled. If an exception
| occurs for which there is no handler, execution of the *compound-statement* is
| terminated. If a warning or not found condition occurs for which there is no
| handler, processing continues with the next statement.

Examples

Create a procedure body with a compound statement that performs the following actions.

1. Declares SQL variables.
2. Declares a cursor to return the salary of employees in a department determined by an IN parameter.
3. Declares an EXIT handler for the condition NOT FOUND (end of file) which assigns the value 6666 to the OUT parameter medianSalary.
4. Select the number of employees in the given department into the SQL variable v_numRecords.
5. Fetch rows from the cursor in a WHILE loop until 50% + 1 of the employees have been retrieved.
6. Return the median salary.

```
CREATE PROCEDURE DEPT_MEDIAN
(IN deptNumber SMALLINT,
 OUT medianSalary DOUBLE)
LANGUAGE SQL
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE c1 CURSOR FOR
    SELECT salary FROM staff
      WHERE DEPT = deptNumber
      ORDER BY salary;
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET medianSalary = 6666;
  /* initialize OUT parameter */
  SET medianSalary = 0;
  SELECT COUNT(*) INTO v_numRecords FROM staff
    WHERE DEPT = deptNumber;
  OPEN c1;
  WHILE v_counter < (v_numRecords / 2 + 1) DO
    FETCH c1 INTO medianSalary;
    SET v_counter = v_counter + 1;
  END WHILE;
  CLOSE c1;
END
```


FOR

Notes

FOR statement rules: The FOR statement executes one or multiple statements for each row in a table. The cursor is defined by specifying a select list that describes the columns and rows selected. The statements within the FOR statement are executed for each row selected.

The select list must consist of unique column names and the table specified in the select list must exist when the function, procedure, or trigger is created.

The cursor specified in a FOR statement cannot be referenced outside the FOR statement and cannot be specified on an OPEN, FETCH, or CLOSE statement.

Example

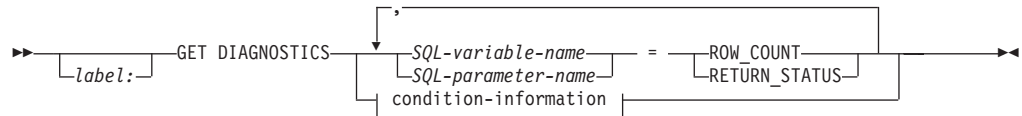
In this example, the FOR statement is used to specify a cursor that selects 3 columns from the employee table. For every row selected, SQL variable *fullname* is set to the last name followed by a comma, the first name, a blank, and the middle initial. Each value for *fullname* is inserted into table T NAMES.

```
BEGIN
  DECLARE fullname CHAR(40);
  FOR v1 AS
    c1 CURSOR FOR
    SELECT firstnme, midinit, lastname FROM employee
  DO
    SET fullname =
      lastname || ', ' || firstnme || ' ' || midinit;
    INSERT INTO T NAMES VALUE ( fullname );
  END FOR;
END;
```

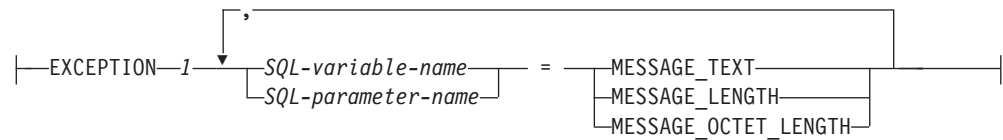
GET DIAGNOSTICS statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

Syntax



condition-information:



Description

label

Specifies the label for the GET DIAGNOSTICS statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

SQL-variable-name

Identifies the SQL variable that is the assignment target. If MESSAGE_TEXT is specified, the SQL variable must have a CHAR or VARCHAR data type. Otherwise, the SQL variable must be an integer variable.

SQL-parameter-name

Identifies the SQL parameter that is the assignment target. If MESSAGE_TEXT is specified, the SQL parameter must have a CHAR or VARCHAR data type. Otherwise, the SQL parameter must be an integer variable.

ROW_COUNT

Identifies the number of rows associated with the previous SQL statement that was executed. If the previous SQL statement is a DELETE, INSERT, or UPDATE statement, ROW_COUNT identifies the number of rows deleted, inserted, or updated by that statement, excluding rows affected by either triggers or referential integrity constraints. If the previous statement is a PREPARE statement, ROW_COUNT identifies the estimated number of result rows in the prepared statement.

RETURN_STATUS

Identifies the status value returned from the previous SQL CALL statement. If the previous statement is not a CALL statement, the value returned has no meaning and is unpredictable. For more information, see “RETURN statement” on page 814.

GET DIAGNOSTICS

condition-information

Specifies that error or warning information will be returned about the previous SQL statement.

If information is desired about an error, the GET DIAGNOSTICS statement must be the first statement specified in the handler that will handle the error.

If information is desired about a warning,

- If a handler will get control for the warning condition, the GET DIAGNOSTICS statement must be the first statement specified in that handler.
- If a handler will not get control for the warning condition, the GET DIAGNOSTICS statement must be the next statement executed after that previous statement.

MESSAGE_TEXT

Identifies the message text of the error or warning returned from the previous SQL statement that was executed. If the previous SQL statement completes with an SQLCODE equal to zero, an empty string or blanks is returned. If the message text is longer than the length attribute of the *SQL-variable-name*, no warning is returned.

MESSAGE_LENGTH or MESSAGE_OCTET_LENGTH

Identifies the length of the message text of the error or warning returned from the previous SQL statement that was executed. If the previous SQL statement completes with an SQLCODE equal to zero, a length of 0 is returned.

Notes

Effect of statement: The GET DIAGNOSTICS statement does not change the contents of the diagnostics area (SQLCA). If an SQLSTATE or SQLCODE special variable is declared in an SQL procedure, SQL function, or SQL trigger, these are set to the SQLSTATE or SQLCODE returned from issuing the GET DIAGNOSTICS statement.

Example

In an SQL procedure, execute a GET DIAGNOSTICS statement to determine how many rows were updated.

```
CREATE PROCEDURE sqlprocg (IN deptnbr VARCHAR(3))
LANGUAGE SQL
BEGIN
  DECLARE SQLSTATE CHAR(5);
  DECLARE rcount INTEGER;
  UPDATE CORPDATA.PROJECT
    SET PRSTAFF = PRSTAFF + 1.5
    WHERE DEPTNO = deptnbr;
  GET DIAGNOSTICS rcount = ROW_COUNT;
  /* At this point, rcount contains the number of rows that were updated. */
END
```

Within an SQL procedure, handle the returned status value from the invocation of a stored procedure called TRYIT. TRYIT could use the RETURN statement to explicitly return a status value or a status value could be implicitly returned by the database manager. If the procedure is successful, it returns a value of zero.

```
CREATE PROCEDURE TESTIT ()
LANGUAGE SQL
A1: BEGIN
  DECLARE RETVAL INTEGER DEFAULT 0;
  ...
```

```

|          CALL TRYIT
|          GET DIAGNOSTICS RETVAL = RETURN_STATUS;
|          IF RETVAL <> 0 THEN
|              ...
|              LEAVE A1;
|          ELSE
|              ...
|          END IF;
|      END A1

```

In an SQL procedure, execute a GET DIAGNOSTICS statement to retrieve the message text for an error.

```

|      CREATE PROCEDURE divide2 ( IN numerator INTEGER,
|                                IN denominator INTEGER,
|                                OUT divide_result INTEGER,
|                                OUT divide_error VARCHAR(70) )
|
|      LANGUAGE SQL
|      BEGIN
|          DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
|              GET DIAGNOSTICS EXCEPTION 1
|                  divide_error = MESSAGE_TEXT;
|          SET divide_result = numerator / denominator;
|      END;

```

GOTO statement

The GOTO statement branches to a user-defined label within an SQL routine or SQL trigger.

Syntax

```

  ───────────┬────────── GOTO ─label2──────────────────────────────────────────▶
  └──label1:──┘

```

Description

label1

Specifies the label for the GOTO statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

label2

Specifies the labelled statement where processing is to continue. The labelled statement and the GOTO statement must both be in the same scope:

- If the GOTO statement is defined in a FOR statement, *label* must be defined inside the same FOR statement, excluding a nested FOR statement.
- If the GOTO statement is defined outside a FOR statement, *label* must not be defined within a FOR statement.
- If the GOTO statement is defined in a handler, *label* must be defined inside the same handler.
- If the GOTO statement is defined outside a handler, *label* must not be defined within a handler.

If *label2* is not defined within a scope that the GOTO statement can reach, an error is returned.

Notes

Using a GOTO statement: Use the GOTO statement sparingly. This statement interferes with the normal sequence of processing, thus making a routine more difficult to read and maintain. Often, another statement, such as IF or LEAVE, can eliminate the need for a GOTO statement.

Example

In the following statement, the parameters *rating* and *v_empno* are passed in to the procedure. The time in service is returned as a date duration in output parameter *return_parm*. If the time in service with the company is less than 6 months, the GOTO statement transfers control to the end of the procedure and *new_salary* is left unchanged.

```

CREATE PROCEDURE adjust_salary
  (IN v_empno CHAR(6),
  IN rating INTEGER,
  OUT return_parm DECIMAL(8,2))
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
  DECLARE new_salary DECIMAL(9,2);

```

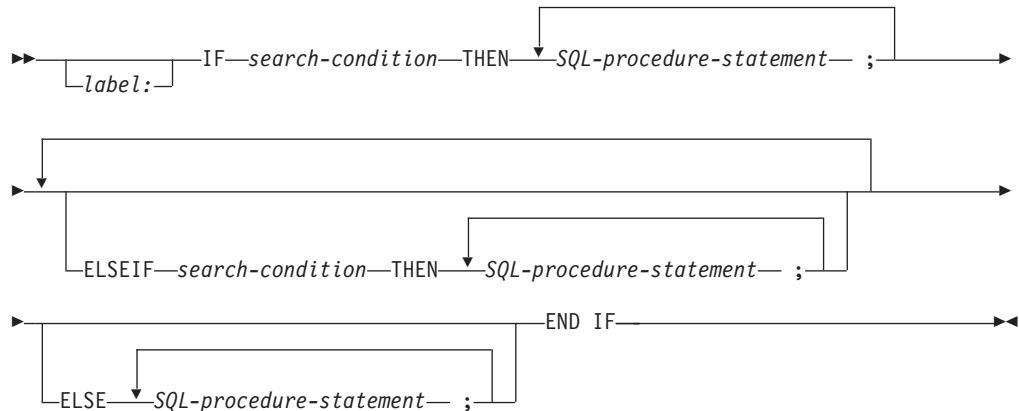
```
DECLARE service DECIMAL(8,2);
SELECT salary, current_date - hiredate
  INTO new_salary, service
  FROM employee
  WHERE empno = v_empno;
IF service < 600
  THEN GOTO exit1;
END IF;
IF rating = 1
  THEN SET new_salary = new_salary + (new_salary * .10);
  ELSEIF rating = 2
  THEN SET new_salary = new_salary + (new_salary * .05);
END IF;
UPDATE employee
  SET salary = new_salary
  WHERE empno = v_empno;

exit1: SET return_parm = service;
END
```

IF statement

The IF statement executes different sets of SQL statements based on the result of search conditions.

Syntax



Description

label

Specifies the label for the IF statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

search-condition

Specifies the *search-condition* for which an SQL statement should be executed. If the condition is unknown or false, processing continues to the next search condition, until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement

Specifies an SQL statement that should be executed if the preceding *search-condition* is true.

Example

The following SQL procedure accepts two IN parameters: an employee number and an employee rating. Depending on the value of *rating*, the employee table is updated with new values in the salary and bonus columns.

```

CREATE PROCEDURE UPDATE_SALARY_IF
  (IN employee_number CHAR(6), INOUT rating SMALLINT)
  LANGUAGE SQL
  MODIFIES SQL DATA
  BEGIN
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE EXIT HANDLER FOR not_found
      SET rating = -1;
    IF rating = 1
      THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
  
```



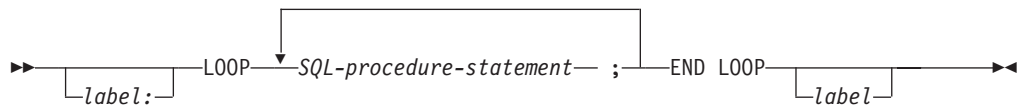
```
        WHERE empno = employee_number;
ELSEIF rating = 2
    THEN UPDATE employee
        SET salary = salary * 1.05, bonus = 500
        WHERE empno = employee_number;
ELSE UPDATE employee
    SET salary = salary * 1.03, bonus = 0
    WHERE empno = employee_number;
END IF;
END
```


LOOP

LOOP statement

The LOOP statement repeats the execution of a statement or a group of statements.

Syntax



Description

label

Specifies the label for the LOOP statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

SQL-procedure statement

Specifies an SQL statement to be executed in the loop

Examples

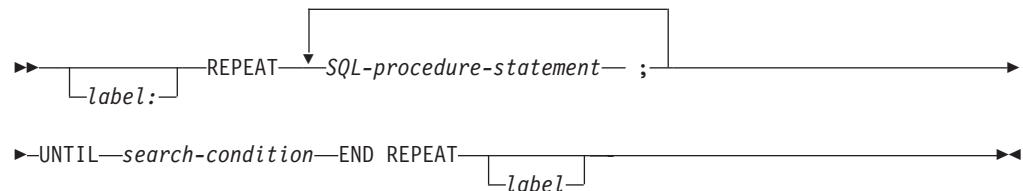
This procedure uses a LOOP statement to fetch values from the employee table. Each time the loop iterates, the OUT parameter *counter* is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

```
CREATE PROCEDURE LOOP_UNTIL_SPACE (OUT COUNTER INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE v_firstname VARCHAR(12);
  DECLARE v_midinit CHAR(1);
  DECLARE v_lastname VARCHAR(15);
  DECLARE c1 CURSOR FOR
    SELECT firstname, midinit, lastname
    FROM employee;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET counter = -1;
  OPEN c1;
  fetch_loop:
  LOOP
    FETCH c1 INTO v_firstname, v_midinit, v_lastname;
    IF v_midinit = ' ' THEN
      LEAVE fetch_loop;
    END IF;
    SET v_counter = v_counter + 1;
  END LOOP fetch_loop;
  SET counter = v_counter;
  CLOSE c1;
END
```

REPEAT statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Description

label

Specifies the label for the REPEAT statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

SQL-procedure-statement

Specifies an SQL statement to be executed in the REPEAT loop.

search-condition

The *search-condition* is evaluated after each execution of the REPEAT loop. If the condition is true, the REPEAT loop will exit. If the condition is unknown or false, the looping continues.

Example

A REPEAT statement fetches rows from a table until the *not_found* condition handler is invoked.

```

CREATE PROCEDURE REPEAT_STMT (OUT COUNTER INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE v_firstname VARCHAR(12);
  DECLARE v_midinit CHAR(1);
  DECLARE v_lastname VARCHAR(15);
  DECLARE at_end SMALLINT DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE c1 CURSOR FOR
    SELECT firstame, midinit, lastname
    FROM employee;
  DECLARE CONTINUE HANDLER FOR not_found
    SET at_end = 1;
  OPEN c1;
  fetch_loop:
  REPEAT
    FETCH c1 INTO v_firstname, v_midinit, v_lastname;
    SET v_counter = v_counter + 1;
  UNTIL at_end > 0
  
```

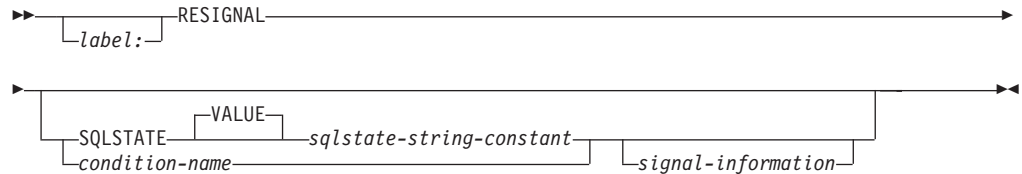
REPEAT

```
END REPEAT fetch_loop;  
SET counter = v_counter;  
CLOSE c1;  
END
```

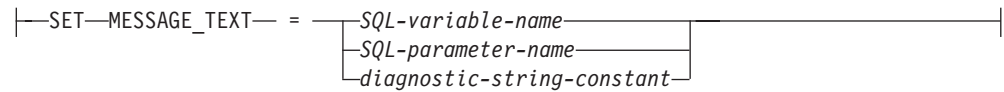
RESIGNAL statement

The RESIGNAL statement is used within a handler to return an error or warning condition.

Syntax



signal-information:



Description

label

Specifies the label for the RESIGNAL statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

SQLSTATE VALUE *sqlstate-string-constant*

Specifies the SQLSTATE error code that will be returned. The *sqlstate-string-constant* must be a character string constant with exactly 5 characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00' since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is returned.

condition-name

Specifies the name of the condition that will be returned. The *condition-name* must be declared within the compound statement.

MESSAGE_TEXT

Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA. If the actual length of the string is longer than 70 bytes, it is truncated without a warning.

SQL-variable-name

Identifies an SQL variable that must be declared within the *compound-statement*. The SQL variable must be defined as CHAR or VARCHAR data type.

RESIGNAL

SQL-parameter-name

Identifies an SQL parameter that must be declared within the *compound-statement*. The SQL parameter must be defined as CHAR or VARCHAR data type.

diagnostic-string-constant

Specifies a character string constant that contains the message text.

Notes

SQLSTATE values: Any valid SQLSTATE value can be used in the RESIGNAL statement. However, it is recommended that programmers define new SQLSTATEs based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

For more information about SQLSTATEs, see the SQL Messages and Codes book in the iSeries Information Center.

SQLSTATE values:

- If the RESIGNAL statement is specified without a SQLSTATE clause or a *condition-name*, the RESIGNAL statement must be in a handler. The SQL routine returns to the caller with the identical condition that invoked the handler.
- When a RESIGNAL statement is issued and an SQLSTATE or *condition-name* is specified, the SQLCODE returned in the SQLCA is based on the SQLSTATE value as follows:
 - If the specified SQLSTATE class is either '01' or '02', a warning or not found is signalled and the SQLCODE is set to +438.
 - Otherwise, an exception is returned and the SQLCODE is set to -438.
- When a RESIGNAL statement is issued and neither an SQLSTATE value or *condition-name* is specified, the SQLCODE is not changed.

If the SQLSTATE or condition indicates that an exception is signalled (SQLSTATE class other than '01' or '02');

- If a handler exists in the same compound statement as the RESIGNAL statement, and the *compound-statement* contains a handler for SQLEXCEPTION or the specified SQLSTATE or condition; the exception is handled and control is transferred to that handler.
- If the *compound-statement* is nested and an outer level *compound-statement* has a handler for SQLEXCEPTION or the specified SQLSTATE or condition; the exception is handled and control is transferred to that handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If the SQLSTATE or condition indicates that a warning (SQLSTATE class '01') or not found (SQLSTATE class '02') is signalled:

- If a handler exists in the same compound statement as the RESIGNAL statement, and the *compound-statement* contains a handler for SQLWARNING (if the SQLSTATE class is '01'), NOT FOUND (if the SQLSTATE class is '02'), or the specified SQLSTATE or condition; the warning or not found condition is handled and control is transferred to that handler.
- If the *compound-statement* is nested and an outer level compound statement contains a handler for SQLWARNING (if the SQLSTATE class is '01'), NOT

| FOUND (if the SQLSTATE class is '02'), or the specified SQLSTATE or condition;
| the warning or not found condition is handled and the exception is handled and
| control is returned to that handler.

- Otherwise, the warning is not handled and processing continues with the next statement.

SQLSTATE values are comprised of a two-character class code value, followed by a three-character subclass code value. Class code values represent classes of successful and unsuccessful execution conditions.

Example

This example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the overflow condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE value to the client application.

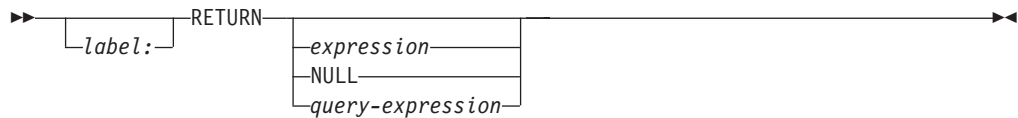
```
CREATE PROCEDURE divide ( IN numerator INTEGER,  
                          IN denominator INTEGER,  
                          OUT divide_result INTEGER )  
  
LANGUAGE SQL  
BEGIN  
    DECLARE overflow CONDITION FOR '22003';  
    DECLARE CONTINUE HANDLER FOR overflow  
        RESIGNAL SQLSTATE '22375';  
    IF denominator = 0 THEN  
        SIGNAL overflow;  
    ELSE  
        SET divide_result = numerator / denominator;  
    END IF;  
END
```

RETURN

RETURN statement

The RETURN statement returns from a routine. For SQL functions, it returns the result of the function. For an SQL procedure, it optionally returns an integer status value. For SQL table functions, it returns a table as the result of the function.

Syntax



Description

label

Specifies the label for the RETURN statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

expression

Specifies a value that is returned from the routine:

- If the routine is a function, *expression* must be specified and the value and the value of *expression* must conform to the SQL assignment rules as described in “Assignments and Comparisons” on page 80. If assigning to a string variable, storage assignment rules apply.
- If the routine is a procedure, the data type of *expression* must be INTEGER. If the *expression* evaluates to the null value, a value of 0 is returned.

NULL

The null value is returned from the SQL function. NULL is not allowed in SQL procedures.

query-expression

Specifies a *query-expression* value that is returned from the routine. The *query-expression* is a *common-table-expression* or *fullselect*. A *query-expression* is only allowed in an SQL trigger.

Notes

Returning from a procedure:

- If a RETURN statement with a specified return value is used to return from a procedure then the SQLCODE, SQLSTATE, and message length in the SQLCA are initialized to zeros, and message text is set to blanks. An error is not returned to the caller.
- If a RETURN statement is not used to return from a procedure or if a value is not specified on the RETURN statement,
 - if the procedure returns with an SQLCODE that is greater or equal to zero, the specified target for RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of 0
 - if the procedure returns with an SQLCODE that is less than zero, the specified target for RETURN_STATUS in a GET DIAGNOSTICS statement will be set to a value of -1.

RETURN

- When a value is returned from a procedure, the caller may access the value using:
 - the GET DIAGNOSTICS statement to retrieve the RETURN_STATUS when the SQL procedure was called from another SQL procedure
 - the parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI application
 - directly from the SQLCA returned from processing the CALL of an SQL procedure by retrieving the value of sqlerrd[0] when the SQLCODE is not less than zero. When the SQLCODE is less than zero, the sqlerrd[0] value is not set and the application should assume a return status value of -1.

RETURN restrictions:

- RETURN is not allowed in SQL triggers.
- Only one RETURN statement is allowed in an SQL table function statement *routine-body*.

Example

Use a RETURN statement to return from an SQL procedure with a status value of zero if successful, and -200 if not.

```
BEGIN
...
GOTO fail;
...
success: RETURN 0
failure: RETURN -200
...
END
```

Define a scalar function that returns the tangent of a value using the existing sine and cosine functions.

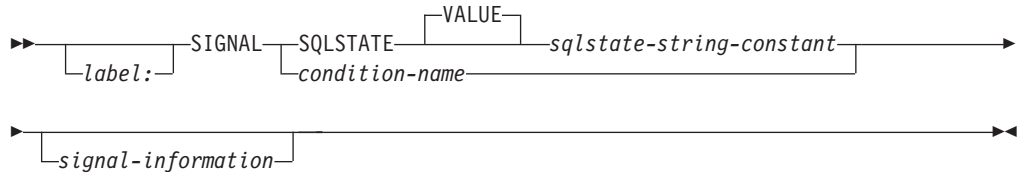
```
CREATE FUNCTION mytan (x DOUBLE)
RETURNS DOUBLE
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
RETURN SIN(x)/COS(x)
```

SIGNAL

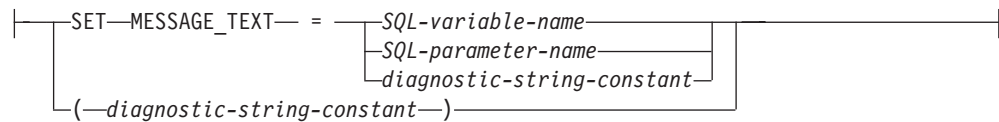
SIGNAL statement

The SIGNAL statement signals an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE and optional message text.

Syntax



signal-information:



Description

label

Specifies the label for the SIGNAL statement. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

SQLSTATE VALUE *sqlstate-string-constant*

Specifies the SQLSTATE that will be signalled. The *sqlstate-string-constant* must be a character string constant with exactly 5 characters that follow the rules for SQLSTATES:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z').
- The SQLSTATE class (first two characters) cannot be '00' since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is returned.

condition-name

Specifies the name of the condition that will be signalled. The *condition-name* must be declared within the *compound-statement*.

SET MESSAGE_TEXT

Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA. If the actual length of the string is longer than 70 bytes, it is truncated without a warning.

SQL-variable-name

Identifies an SQL variable declared within the *compound-statement*, that contains the message text. The SQL variable must be defined as CHAR or VARCHAR data type.

SQL-parameter-name

Identifies an SQL parameter declared within the *compound-statement*, that contains the message text. The SQL parameter must be defined as CHAR or VARCHAR data type.

diagnostic-string-constant

Specifies a character string constant that contains the message text. If the string is longer than 70 bytes, it will be truncated without warning.

(diagnostic-string-constant)

Specifies a character string constant that contains the message text. If the string is longer than 70 bytes, it will be truncated without warning. Within the triggered action of a CREATE TRIGGER statement, the message text can only be specified using this syntax:

```
SIGNAL SQLSTATE sqlstate-string-constant (diagnostic-string-constant);
```

To conform with the ANS and ISO standards, this form should not be used. It is provided for compatibility with other products.

Notes

SQLSTATE values: Any valid SQLSTATE value can be used in the SIGNAL statement. However, it is recommended that programmers define new SQLSTATES based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

For more information about SQLSTATES, see the SQL Messages and Codes book in the iSeries Information Center.

Processing a SIGNAL statement: When a SIGNAL statement is issued, the SQLCODE returned in the SQLCA is based on the SQLSTATE value as follows:

- If the specified SQLSTATE class is either '01' or '02', a warning or not found is signalled and the SQLCODE is set to +438.
- Otherwise, an exception is signalled and the SQLCODE is set to -438.

If the SQLSTATE or condition indicates that an exception (SQLSTATE class other than '01' or '02') is signalled,

- If a handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a handler for SQLEXCEPTION or the specified SQLSTATE or condition; the exception is handled and control is transferred to that handler.
- Otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.

If the SQLSTATE or condition indicates that a warning (SQLSTATE class '01') or not found (SQLSTATE class '02') is signalled,

- If a handler exists in the same compound statement as the SIGNAL statement, and the compound statement contains a handler for SQLWARNING (if the SQLSTATE class is '01'), NOT FOUND (if the SQLSTATE class is '02'), or the specified SQLSTATE or condition; the warning or not found condition is handled and control is transferred to that handler.
- Otherwise, the warning is not handled and processing continues with the next statement.

SIGNAL

SQLSTATE values are comprised of a two-character class code value, followed by a three-character subclass code value. Class code values represent classes of successful and unsuccessful execution conditions.

Any valid SQLSTATE value can be used in the SIGNAL statement. However, it is recommended that programmers define new SQLSTATEs based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.

- SQLSTATE classes that begin with the characters '7' through '9' or 'I' through 'Z' may be defined. Within these classes, any subclass may be defined.
- SQLSTATE classes that begin with the characters '0' through '6' or 'A' through 'H' are reserved for the database manager. Within these classes, subclasses that begin with the characters '0' through 'H' are reserved for the database manager. Subclasses that begin with the characters 'I' through 'Z' may be defined.

For more information about SQLSTATEs, see the SQL Messages and Codes book in the iSeries Information Center.

Example

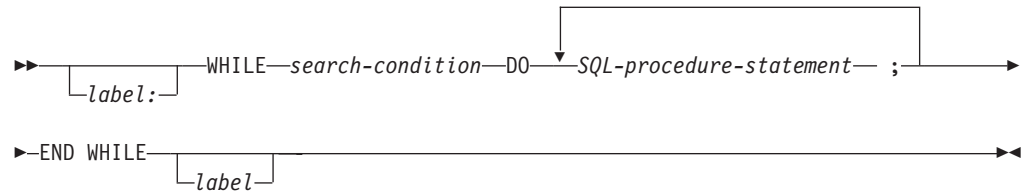
An SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
  (IN ONUM INTEGER, IN CNUM INTEGER,
   IN PNUM INTEGER, IN QNUM INTEGER)
  LANGUAGE SQL
  MODIFIES SQL DATA
  BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
      SIGNAL SQLSTATE '75002'
      SET MESSAGE_TEXT = 'Customer number is not known';
    INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
      VALUES (ONUM, CNUM, PNUM, QNUM);
  END
```

WHILE statement

The WHILE statement repeats the execution of a statement while a specified condition is true.

Syntax



Description

label

Specifies the label for the WHILE statement. If the ending label is specified, it must be the same as the beginning label. The label name cannot be the same as another label in the same *compound-statement* and it cannot be the same as the name of the SQL function, SQL procedure, or SQL trigger in which the label is used.

search-condition

Specifies a condition that is evaluated before each execution of the WHILE loop. If the condition is true, the *SQL-procedure-statements* in the WHILE loop are executed.

SQL-procedure-statement

Specifies an SQL statement or statements to execute within the WHILE loop.

Example

This example uses a WHILE statement to iterate through FETCH and SET statements. While the value of SQL variable *v_counter* is less than half of number of employees in the department identified by the IN parameter *deptNumber*, the WHILE statement continues to perform the FETCH and SET statements. When the condition is no longer true, the flow of control leaves the WHILE statement and closes the cursor.

```

CREATE PROCEDURE dept_median
(IN deptNumber SMALLINT,
OUT medianSalary DECIMAL(7,2))
LANGUAGE SQL
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE c1 CURSOR FOR
    SELECT salary
    FROM staff
    WHERE dept = deptNumber
    ORDER BY salary;
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET medianSalary = 6666;
  SET medianSalary = 0;
  SELECT COUNT(*) INTO v_numRecords

```

WHILE

```
FROM staff
WHERE dept = deptNumber;
OPEN c1;
WHILE v_counter < (v_numRecords/2 + 1) DO
    FETCH c1 INTO medianSalary;
    SET v_counter = v_counter +1;
END WHILE;
CLOSE c1;
END
```

Appendix A. SQL Limits

The following tables describe certain limits imposed by the DB2 UDB for iSeries database manager.

Table 59. Identifier Length Limits

Identifier Limits	DB2 UDB for iSeries Limit
Longest authorization name	10
Longest condition name	128
Longest correlation name	128
Longest cursor name	18
Longest external program name (unqualified form)	10
Longest external program name (string form) ⁷⁰	279
Longest host identifier ⁷¹	64
Longest savepoint name	128
Longest schema name	10
Longest server name	18
Longest statement name	18
Longest unqualified alias name	128
Longest unqualified column name	30
Longest unqualified constraint name	128
Longest unqualified distinct type name	128
Longest unqualified function name	128
Longest unqualified index name	128
Longest unqualified nodegroup name	10
Longest unqualified package name	10
Longest unqualified procedure name	128
Longest unqualified specific name	128
Longest unqualified SQL parameter name	128
Longest unqualified SQL variable name	128
Longest unqualified system column name	10
Longest unqualified system table, view, and index name	10
Longest unqualified table and view name	128
Longest unqualified trigger name	128

70. For REXX procedures, the limit is 33.

71. For a C program, the limit is 128.

SQL Limits

Table 60. Numeric Limits

Numeric Limits	DB2 UDB for iSeries Limit
Smallest SMALLINT value	-32 768
Largest SMALLINT value	+32 767
Smallest INTEGER value	-2 147 483 648
Largest INTEGER value	+2 147 483 647
Smallest BIGINT value	-9 223 372 036 854 775 808
Largest BIGINT value	+9 223 372 036 854 775 807
Largest decimal precision	31
Smallest DOUBLE value ⁷³	-1.79x10 ³⁰⁸
Largest DOUBLE value ⁷³	+1.79x10 ³⁰⁸
Smallest positive DOUBLE value ⁷³	+2.23x10 ⁻³⁰⁸
Largest negative DOUBLE value ⁷³	-2.23x10 ⁻³⁰⁸
Smallest REAL value ⁷³	-3.4x10 ³⁸
Largest REAL value ⁷³	+3.4x10 ³⁸
Smallest positive REAL value ⁷³	+1.18x10 ⁻³⁸
Largest negative REAL value ⁷³	-1.18x10 ⁻³⁸

Table 61. String Limits

String Limits	DB2 UDB for iSeries Limit
Maximum length of CHAR (in bytes) ⁷⁴	32765
Maximum length of VARCHAR (in bytes) ⁷⁴	32739
Maximum length of CLOB (in bytes)	2 147 483 647
Maximum length of GRAPHIC (in double-byte characters) ⁷⁴	16382
Maximum length of VARGRAPHIC (in double-byte characters) ⁷⁴	16369
Maximum length of DBCLOB (in double-byte characters)	1 073 741 823
Maximum length of BLOB (in bytes)	2 147 483 647
Maximum length of character constant	32740
Maximum length of a graphic constant	16370
Maximum length of binary constant	32740
Maximum length of concatenated character string ⁷⁴	2 147 483 647
Maximum length of concatenated graphic string ⁷⁴	1 073 741 823
Maximum length of concatenated binary string ⁷⁴	2 147 483 647
Maximum number of hex constant digits	65 480
Maximum length of catalog comments	2000
Maximum length of column label	60
Longest SQL routine label	128
Longest table, package, or alias label	50
Maximum length of C NUL-terminated ⁷⁴	32739
Maximum length of C NUL-terminated graphic ⁷⁴	16369

Table 62. Datetime Limits

Datetime Limits	DB2 UDB for iSeries Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000

Table 63. DataLink Limits

Datalink Limits	DB2 UDB for iSeries Limit
Maximum length of DATALINK	32718
Maximum length of DATALINK comment	254

Table 64. Database Manager Limits

Database Manager Limits	DB2 UDB for iSeries Limit
Most columns in a table	8000
Most columns in a view	8000
Maximum length of a row without LOBs including all overhead	32766
Maximum length of a row with LOBs including all overhead	3 758 096 383
Maximum number of parameters in a function	90
Maximum number of parameters in a procedure	254 ⁷⁶
Maximum size of a table	1 terabyte
Maximum size of an index	1 terabyte
Most rows in a table	4 294 967 288
Longest index key	2000
Most columns in an index key	120
Most indexes on a table	approximately 4000
Most tables referenced in an SQL statement	256
Most tables referenced in a view	32
Most host variable declarations in a precompiled program ⁷²	storage
Most host variables and constants in an SQL statement	4096 ⁷⁷
Longest host variable used for insert or update (in bytes)	2 147 483 647
Longest SQL statement	65535
Longest CHECK constraint (in bytes)	statement
Most elements in a select list ⁷⁵	approximately 8000
Most predicates in a WHERE or HAVING clause	statement
Maximum number of columns in a GROUP BY clause	120
Maximum total length of columns in a GROUP BY clause	2000
Maximum number of columns in an ORDER BY clause	10000

SQL Limits

Table 64. Database Manager Limits (continued)

Database Manager Limits	DB2 UDB for iSeries Limit
Maximum total length of columns in an ORDER BY clause	10000
Maximum size of an SQLDA	16 777 215
Maximum number of prepared statements	storage
Most declared cursors in a program	storage
Maximum number of cursors opened at one time	storage
Most tables in a relational database	storage
Maximum number of triggers on a table	300
Maximum number of nested trigger invocations	200
Maximum length of a password	128
Maximum number of constraints on a table	300
Maximum length of a path ⁷⁹	3483
Maximum number of schemas in a path	268
Maximum levels allowed for a subselect	32
Maximum number of rows changed in a unit of work	500 000 000
Maximum procedures with result sets waiting to be fetched	100
Maximum number of locators in a transaction	250 000
Maximum number of savepoints active at one time	storage
Maximum number of simultaneously allocated CLI handles in a process	80 000

72. In RPG/400 and PL/I programs when the old parameter passing technique is used, the limit is approximately 4000. The limit is based on the number of pointers allowed in the program. In all other cases, the limit is based on architectural constraints within the operating system.

73. The values shown are approximate.

74. If the column is NOT NULL, the maximum is one more.

75. The limit is based on the size of internal structures generated for the parsed SQL statement.

76. Procedures with PARAMETER STYLE SQL are limited to 90 parameters. SQL procedures with PARAMETER STYLE GENERAL are limited to 253. Procedures with PARAMETER STYLE GENERAL WITH NULLS are limited to 254. External procedures with PARAMETER STYLE GENERAL are limited to 255. The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program used to compile the external program.

77. If the statement is not read-only, the limit is 2048. The limit is approximate and may be less if very large string constants or string variables are used.

78. The maximum number of allocated handles per DRDA connection is 500.

79. The maximum length of a path in DRDA is 255.

Appendix B. SQL Communication Area

An SQLCA is a set of variables that is updated at the end of the execution of every SQL statement. A program that contains executable SQL statements must provide exactly one SQLCA (unless a stand-alone SQLCODE or a stand-alone SQLSTATE variable is used instead), except in Java, where the SQLCA is not applicable.

The SQL INCLUDE statement can be used to provide the declaration of the SQLCA in all host languages except Java, RPG, or REXX. For information on the use of the SQLCA in a REXX procedure, see the SQL Programming with Host Languages book. For information on how to access the information regarding errors and warnings in Java, refer to the Developer Kit for Java.

In C, COBOL, FORTRAN, and PL/I, the name of the storage area must be SQLCA. Every SQL statement must be within the scope of its declaration.

If a stand-alone SQLCODE or SQLSTATE is specified in the program, the SQLCA must not be included. For more information, see "SQL Return Codes" on page 367.

Field Descriptions

The names in the following table are those provided by the SQL INCLUDE statement. For the most part, C (and C++), COBOL, FORTRAN and PL/I use the same names. RPG names are different, because in RPG/400, they are limited to 6 characters. Note one instance where PL/I names differ from the COBOL names.

Table 65. Names Provided by the SQL INCLUDE Statement

C Name	COBOL Name	FORTRAN ¹ Name	RPG Name	Field Data Type	Field Value								
SQLCAID sqlcaid	Not used	SQLCAID	SQLAID	CHAR(8)	An "eye catcher" for storage dumps, containing 'SQLCA'.								
SQLCABC sqlcab	Not used	SQLCABC	SQLABC	INTEGER	Contains the length of the SQLCA, 136.								
SQLCODE sqlcode	SQLCOD	SQLCOD	SQLCOD	INTEGER	Contains an SQL return code. <table border="0"> <tr> <td>Code</td> <td>Meaning</td> </tr> <tr> <td>0</td> <td>Successful execution although SQLWARN indicators might have been 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> </table>	Code	Meaning	0	Successful execution although SQLWARN indicators might have been set.	positive	Successful execution, but with a warning condition.	negative	Error condition.
Code	Meaning												
0	Successful execution although SQLWARN indicators might have been set.												
positive	Successful execution, but with a warning condition.												
negative	Error condition.												
SQLERRML ² sqlerrml	SQLTXL	SQLERRML	SQLERL	SMALLINT	Length indicator for SQLERRMC, in the range 0 through 70. 0 means that the value of SQLERRMC is not pertinent.								

SQLCA

Table 65. Names Provided by the SQL INCLUDE Statement (continued)

C Name	COBOL Name	FORTTRAN ¹	RPG Name	Field Data Type	Field Value
SQLERRMC ² sqlerrmc	SQLTXT SQLERRMC		SQLERM	CHAR (70)	Contains message replacement text associated with the SQLCODE. For CONNECT and SET CONNECTION, the SQLERRMC field contains information about the connection, see Table 68 on page 830 for a description of the replacement text.
SQLERRP sqlerrp	SQLERP SQLERRP		SQLERP	CHAR(8)	Contains the name of the product and module returning the error. The first three characters identify the product: ARI for DB2 for VM and VSE DSN for DB2 UDB for OS/390 and z/OS QSQ for DB2 UDB for iSeries SQL for all other DB2 products See "CONNECT (Type 1)" on page 416 or "CONNECT (Type 2)" on page 421 for additional information.
SQLERRD sqlerrd	SQLERR SQLERRD		SQLERR ³	Array	Contains six INTEGER variables that provide diagnostic information, see Table 67 on page 828 for a description of the diagnostic information.
SQLWARN sqlwarn	SQLWRN SQLWARN		SQLWRN ⁴	CHAR(11)	A set of 11 CHAR(1) warning indicators, each containing blank or 'W' or 'N'.
SQLSTATE sqlstate	SQLSTT SQLSTATE		SQLSTT	CHAR(5)	A return code that indicates the outcome of the most recently executed SQL statement.

Notes:

- ¹ The first name indicates the IBM SQL SQLCA names for the FORTRAN SQLCA. The second name indicates an alternative name that is available due to the DB2 UDB for iSeries implementation of the SQLCA in FORTRAN.
- ² In COBOL, SQLERRM includes SQLERRML and SQLERRMC. In PL/I, the varying-length string SQLERRM is equivalent to SQLERRML prefixed to SQLERRMC.
- ³ In RPG/400 and ILE RPG/400, SQLERR is defined as 24 characters (not an array) that are redefined by the fields SQLER1 through SQLER6. The fields are full-word binary. In ILE RPG/400, SQLERR is also redefined as an array. The name of the array is SQLERRD.
- ⁴ Defined as 11 characters (not an array).

Table 66. SQLWARN Diagnostic Information

C Name COBOL Name PL/I Name	FORTRAN ¹ Name	RPG Name	Field Value
SQLWARN0 sqlwarn[0]	SQLWRN(0) SQLWARN(1:1)	SQLWN0	Contains 'W' if at least one other indicator contains 'W' or 'N', it is blank if all other indicators are blank.
SQLWARN1 sqlwarn[1]	SQLWRN(1) SQLWARN(2:2)	SQLWN1	Contains 'W' if the value of a string column was truncated when assigned to a host variable. Contains 'N' if *NOCNULRQD was specified on the CRTSQLCI or CRTSQLCPPI command (or CNULRQD(*NO) on the SET OPTION statement) and if the value of a string column was assigned to a C NUL-terminated host variable and if the host variable was large enough to contain the result but not large enough to contain the NUL-terminator.
SQLWARN2 sqlwarn[2]	SQLWRN(2) SQLWARN(3:3)	SQLWN2	Contains 'W' if the null values were eliminated from the argument of a function; not necessarily set to 'W' for the MIN function because its results are not dependent on the elimination of null values.
SQLWARN3 sqlwarn[3]	SQLWRN(3) SQLWARN(4:4)	SQLWN3	Contains 'W' if the number of columns is larger than the number of host variables.
SQLWARN4 sqlwarn[4]	SQLWRN(4) SQLWARN(5:5)	SQLWN4	Contains 'W' if a prepared UPDATE or DELETE statement does not include a WHERE clause.
SQLWARN5 sqlwarn[5]	SQLWRN(5) SQLWARN(6:6)	SQLWN5	Reserved
SQLWARN6 sqlwarn[6]	SQLWRN(6) SQLWARN(7:7)	SQLWN6	Contains 'W' if date arithmetic results in an end-of-month adjustment.
SQLWARN7 sqlwarn[7]	SQLWRN(7) SQLWARN(8:8)	SQLWN7	Reserved
SQLWARN8 sqlwarn[8]	SQLWRX(1) SQLWARN(9:9)	SQLWN8	Contains 'W' if the result of a character conversion contains the substitution character.
SQLWARN9 sqlwarn[9]	SQLWRX(2) SQLWARN(10:10)	SQLWN9	Reserved
SQLWARNA sqlwarn[10]	SQLWRX(3) SQLWARN(11:11)	SQLWNA	Reserved

SQLCA

Table 67. *SQLERRD Diagnostic Information*

C Name COBOL Name PL/I Name	FORTTRAN ¹ Name	RPG Name	Field Value
SQLERRD(1) sqlerrd[0]	SQLERR(1)	SQLER1	<p>Contains the last four characters of the CPF escape message if SQLCODE is less than 0. For example, if the message is CPF5715, X'F5F7F1F5' is placed in SQLERRD(1).¹</p> <p>For a call to a procedure, contains the return status value specified on the RETURN statement. If a return status value is not specified on the RETURN statement,</p> <ul style="list-style-type: none">• 0 is returned if the call statement is successful, or• -200 is returned if the call statement is not successful.
SQLERRD(2) sqlerrd[1]	SQLERR(2)	SQLER2	<p>Contains the last four characters of a CPD diagnostic message if the SQL code is less than 0.¹</p> <p>For a CALL statement, SQLERRD(2) contains the number of result sets.</p>
SQLERRD(3) sqlerrd[2]	SQLERR(3)	SQLER3	<p>For a CONNECT for status statement, SQLERRD(3) contains information on the connection status. See "CONNECT (Type 2)" on page 421 for more information.</p> <p>For INSERT, UPDATE, and DELETE, shows the number of rows affected.</p> <p>For a FETCH statement, SQLERRD(3) contains the number of rows fetched.</p> <p>For the PREPARE statement, contains the estimated number of rows selected. If the number of rows is greater than 2 147 483 647, then 2 147 483 647 is returned.</p>

Table 67. SQLERRD Diagnostic Information (continued)

C Name	COBOL Name	FORTRAN ¹	RPG Name	Field Value
PL/I Name	Name	Name		
SQLERRD(4) sqlerrd[3]	SQLERR(4)	SQLERR(4)	SQLER4	<p>For the PREPARE statement, contains a relative number estimate of the resources required for every execution. This number varies depending on the current availability of indexes, file sizes, CPU model, etc. It is an estimated cost for the access plan chosen by the DB2 UDB for iSeries Query Optimizer.</p> <p>For a CONNECT and SET CONNECTION statement, SQLERRD(4) contains the type of conversation used and whether or not committable updates can be performed. See "CONNECT (Type 2)" on page 421 for more information.</p> <p>For a CALL statement, SQLERRD(4) contains the message key of the error that caused the procedure to fail. The QMHRTVPM API can be used to return the message description for the message key.</p> <p>For a trigger error in a DELETE, INSERT or UPDATE statement, SQLERRD(4) contains the message key of the error that was signaled from the trigger program. The QMHRTVPM API can be used to return the message description for the message key.</p> <p>For a FETCH statement, SQLERRD(4) contains the length of the row retrieved.</p>
SQLERRD(5) sqlerrd[4]	SQLERR(5)	SQLERR(5)	SQLER5	<p>For a CALL statement, SQLERRD(5) contains the number of result sets returned from the procedure.</p> <p>For a CONNECT or SET CONNECTION statement, SQLERRD(5) contains:</p> <ul style="list-style-type: none"> • -1 if the connection is unconnected • 0 if the connection is local • 1 if the connection is remote <p>For a DELETE statement, shows the number of rows affected by referential constraints.</p> <p>For an EXECUTE IMMEDIATE or PREPARE statement, may contain the position of a syntax error.</p> <p>For a multiple-row FETCH statement, SQLERRD(5) contains +100 if the last row currently in the table has been fetched.</p> <p>For a PREPARE statement, SQLERRD(5) contains the number of parameter markers in the prepared statement.</p>
SQLERRD(6) sqlerrd[5]	SQLERR(6)	SQLERR(6)	SQLER6	<p>Contains the SQL completion message identifier when the SQLCODE is 0.</p> <p>In all other cases, it is undefined.</p>

Note:

¹ SQLERRD(1) and SQLERRD(2) are set only if appropriate and only if the current server is DB2 UDB for iSeries.

SQLCA

Table 68. *SQLERRMC Replacement Text for CONNECT and SET CONNECTION*

Description	Data type
Relational Database Name	CHAR(18)
Product Identification (same as SQLERRP)	CHAR(8)
User ID of the server job	CHAR(10)
Connection method (*DUW or *RUW)	CHAR(10)
DDM server class name	CHAR(10)
QAS	DB2 UDB for iSeries
QDB2	DB2 UDB for OS/390 and z/OS
QDB2/2	DB2 for OS/2
QDB2/6000	DB2 for AIX/6000
QDB2/6000 PE	DB2 UDB for AIX Parallel Edition
QDB2/AIX64	DB2 UDB for AIX 64-bit
QDB2/HPUX	DB2 for HP-UX**
QDB2/HP64	DB2 UDB for HP-UX** 64-bit
QDB2/LINUX	DB2 UDB for Linux
QDB2/LINUX390	DB2 UDB for Linux
QDB2/LINUXIA64	DB2 UDB for Linux
QDB2/LINUXPPC	DB2 UDB for Linux
QDB2/LINUXPPC64	DB2 UDB for Linux
QDB2/LINUXZ64	DB2 UDB for Linux
QDB2/NT	DB2 for Windows** NT
QDB2/NT64	DB2 UDB for Windows** NT 64-bit
QDB2/PTX	DB2 UDB for NUMA-Q**
QDB2/SCO	DB2 UDB for SCO** UnixWare
QDB2/SGI	DB2 UDB for Silicon Graphics**
QDB2/SNI	DB2 UDB for Siemens Nixdorf**
QDB2/SUN	DB2 for SUN** Solaris**
QDB2/SUN64	DB2 UDB for SUN** 64-bit
QDB2/Windows 95	DB2 UDB for Windows** 95 or Windows** 98
QSQLDS/VM	DB2 for VM and VSE
QSQLDS/VSE	DB2 for VM and VSE
Connection type (same as SQLERRD(4))	SMALLINT

INCLUDE SQLCA Declarations

In C and C++, INCLUDE SQLCA declarations are equivalent to the following:

```
#ifndef    SQLCODE
struct    sqlca
{
        unsigned char  sqlcaid[8];
        long           sqlcabc;
        long           sqlcode;
        short          sqlerrml;
        unsigned char  sqlerrmc[70];
        unsigned char  sqlerrp[8];
        long           sqlerrd[6];
        unsigned char  sqlwarn[11];
        unsigned char  sqlstate[5];
};
#define    SQLCODE    sqlca.sqlcode
#define    SQLWARN0   sqlca.sqlwarn[0]
#define    SQLWARN1   sqlca.sqlwarn[1]
#define    SQLWARN2   sqlca.sqlwarn[2]
#define    SQLWARN3   sqlca.sqlwarn[3]
#define    SQLWARN4   sqlca.sqlwarn[4]
#define    SQLWARN5   sqlca.sqlwarn[5]
#define    SQLWARN6   sqlca.sqlwarn[6]
#define    SQLWARN7   sqlca.sqlwarn[7]
#define    SQLWARN8   sqlca.sqlwarn[8]
#define    SQLWARN9   sqlca.sqlwarn[9]
#define    SQLWARNA   sqlca.sqlwarn[10]
#define    SQLSTATE   sqlca.sqlstate
#endif
struct sqlca sqlca;
```

In COBOL, INCLUDE SQLCA declarations are equivalent to the following:

```
01 SQLCA.
   05 SQLCAID      PIC X(8).
   05 SQLCABC      PIC S9(9) BINARY.
   05 SQLCODE      PIC S9(9) BINARY.
   05 SQLERRM.
       49 SQLERRML PIC S9(4) BINARY.
       49 SQLERRMC PIC X(70).
   05 SQLERRP      PIC X(8).
   05 SQLERRD      OCCURS 6 TIMES
                   PIC S9(9) BINARY.

   05 SQLWARN.
       10 SQLWARN0 PIC X(1).
       10 SQLWARN1 PIC X(1).
       10 SQLWARN2 PIC X(1).
       10 SQLWARN3 PIC X(1).
       10 SQLWARN4 PIC X(1).
       10 SQLWARN5 PIC X(1).
       10 SQLWARN6 PIC X(1).
       10 SQLWARN7 PIC X(1).
       10 SQLWARN8 PIC X(1).
       10 SQLWARN9 PIC X(1).
       10 SQLWARNA PIC X(1).
   05 SQLSTATE     PIC X(5).
```

Note: In COBOL, INCLUDE SQLCA must not be specified outside the Working Storage Section.

SQLCA

In **FORTRAN**, **INCLUDE SQLCA** declarations are equivalent to the following:

```
CHARACTER SQLCA(136)
CHARACTER SQLCAID*8
INTEGER*4 SQLCABC
INTEGER*4 SQLCODE
INTEGER*2 SQLERRML
CHARACTER SQLERRMC*70
CHARACTER SQLERRP*8
INTEGER*4 SQLERRD(6)
CHARACTER SQLWARN*11
CHARACTER SQLSTOTE*5
EQUIVALENCE (SQLCA( 1), SQLCAID)
EQUIVALENCE (SQLCA( 9), SQLCABC)
EQUIVALENCE (SQLCA( 13), SQLCODE)
EQUIVALENCE (SQLCA( 17), SQLERRML)
EQUIVALENCE (SQLCA( 19), SQLERRMC)
EQUIVALENCE (SQLCA( 89), SQLERRP)
EQUIVALENCE (SQLCA( 97), SQLERRD)
EQUIVALENCE (SQLCA(121), SQLWARN)
EQUIVALENCE (SQLCA(132), SQLSTOTE)

INTEGER*4 SQLCOD,
C      SQLERR(6)
INTEGER*2 SQLTXL
CHARACTER SQLERP*8,
C      SQLWRN(0:7)*1,
C      SQLWRX(1:3)*1,
C      SQLTXT*70,
C      SQLSTT*5,
C      SQLWRNWK*8,
C      SQLWRXWK*3,
C      SQLERRWK*24,
C      SQLERRDWK*24
EQUIVALENCE (SQLWRN(1), SQLWRNWK)
EQUIVALENCE (SQLWRX(1), SQLWRXWK)
EQUIVALENCE (SQLCA(97), SQLERRDWK)
EQUIVALENCE (SQLERR(1), SQLERRWK)
COMMON /SQLCA1/SQLCOD,SQLERR,SQLTXL
COMMON /SQLCA2/SQLERP,SQLWRN,SQLTXT,SQLWRX,SQLSTT
```

In **PL/I**, **INCLUDE SQLCA** declarations are equivalent to the following:

```
DCL 1 SQLCA,
    2 SQLCAID      CHAR(8),
    2 SQLCABC     BIN FIXED(31),
    2 SQLCODE     BIN FIXED(31),
    2 SQLERRM     CHAR(70) VAR,
    2 SQLERRP     CHAR(8),
    2 SQLERRD(6)  BIN FIXED(31),
    2 SQLWARN,
    3 SQLWARN0    CHAR(1),
    3 SQLWARN1    CHAR(1),
    3 SQLWARN2    CHAR(1),
    3 SQLWARN3    CHAR(1),
    3 SQLWARN4    CHAR(1),
    3 SQLWARN5    CHAR(1),
    3 SQLWARN6    CHAR(1),
    3 SQLWARN7    CHAR(1),
    3 SQLWARN8    CHAR(1),
    3 SQLWARN9    CHAR(1),
    3 SQLWARNA    CHAR(1),
    2 SQLSTATE    CHAR(5);
```

In **RPG/400**; SQLCA declarations are equivalent to the following:

ISQLCA	DS			
I		1	8	SQLAID
I		B 9	120	SQLABC
I		B 13	160	SQLCOD
I		B 17	180	SQLERL
I		19	88	SQLERM
I		89	96	SQLERP
I		97	120	SQLERR
I		B 97	1000	SQLER1
I		B 101	1040	SQLER2
I		B 105	1080	SQLER3
I		B 109	1120	SQLER4
I		B 113	1160	SQLER5
I		B 117	1200	SQLER6
I		121	131	SQLWRN
I		121	121	SQLWN0
I		122	122	SQLWN1
I		123	123	SQLWN2
I		124	124	SQLWN3
I		125	125	SQLWN4
I		126	126	SQLWN5
I		127	127	SQLWN6
I		128	128	SQLWN7
I		129	129	SQLWN8
I		130	130	SQLWN9
I		131	131	SQLWNA
I		132	136	SQLSTT

In **ILE RPG/400**; SQLCA declarations are equivalent to the following:

```

D*      SQL Communications area
D SQLCA      DS
D  SQLAID      1      8A
D  SQLABC      9      12B 0
D  SQLCOD     13      16B 0
D  SQLERL     17      18B 0
D  SQLERM     19      88A
D  SQLERP     89      96A
D  SQLERRD    97      120B 0 DIM(6)
D  SQLERR     97      120A
D  SQLER1     97      100B 0
D  SQLER2    101      104B 0
D  SQLER3    105      108B 0
D  SQLER4    109      112B 0
D  SQLER5    113      116B 0
D  SQLER6    117      120B 0
D  SQLWRN    121      131A
D  SQLWN0    121      121A
D  SQLWN1    122      122A
D  SQLWN2    123      123A
D  SQLWN3    124      124A
D  SQLWN4    125      125A
D  SQLWN5    126      126A
D  SQLWN6    127      127A
D  SQLWN7    128      128A
D  SQLWN8    129      129A
D  SQLWN9    130      130A
D  SQLWNA    131      131A
D  SQLSTT    132      136A
D*      End of SQLCA

```

SQLCA

Appendix C. Characteristics of SQL Statements

| This appendix contains information on the characteristics of SQL statements
| pertaining to various places where they are used.

- | • “Actions Allowed on SQL Statements” on page 836 shows whether an SQL
| statement can be executed, prepared interactively or dynamically, and whether
| the statement is processed by the requestor, the server, or the precompiler.
- | • “SQL Statement Data Access Indication in Routines” on page 838 shows the level
| of SQL data access that must be specified to use the SQL statement in a routine.
- | • “Considerations for Using Distributed Relational Database” on page 840 provides
| information about the use of SQL statements when the application server is not
| the same as the application requestor.

Actions Allowed on SQL Statements

Table 69 shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the requester, the server, or the precompiler. The letter **Y** means *yes*.

Table 69. Actions allowed on SQL statements

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
ALTER	Y	Y		Y	
BEGIN DECLARE SECTION ^{4 5}					Y
CALL	Y	Y		Y	
CLOSE ⁴	Y			Y	
COMMENT	Y	Y		Y	
COMMIT	Y	Y		Y	
CONNECT (Type 1 and Type 2) ⁴ ⁵	Y		Y		
CREATE	Y	Y		Y	
DECLARE CURSOR ⁴					Y
DECLARE GLOBAL TEMPORARY TABLE	Y	Y		Y	
DECLARE PROCEDURE ^{4 5}					Y
DECLARE STATEMENT ^{4 5}					Y
DECLARE VARIABLE ^{4 5}					Y
DELETE	Y	Y		Y	
DESCRIBE ⁴	Y			Y	
DESCRIBE TABLE ⁴	Y			Y	
DISCONNECT ^{4 5}	Y		Y		
DROP	Y	Y		Y	
END DECLARE SECTION ^{4 5}					Y
EXECUTE ⁴	Y			Y	
EXECUTE IMMEDIATE ⁴	Y			Y	
FETCH	Y			Y	
FREE LOCATOR ^{4 5}	Y	Y		Y	
GET DIAGNOSTICS ²	Y			Y	
GRANT	Y	Y		Y	
HOLD LOCATOR ^{4 5}	Y	Y		Y	
INCLUDE ^{4 5}					Y
INSERT	Y	Y		Y	
LABEL	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
OPEN ⁴	Y			Y	
PREPARE ⁴	Y			Y	

Table 69. Actions allowed on SQL statements (continued)

SQL statement	Executable	Interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
RELEASE connection ^{4 5}	Y		Y		
RELEASE SAVEPOINT	Y	Y		Y	
RENAME	Y	Y		Y	
REVOKE	Y	Y		Y	
ROLLBACK	Y	Y		Y	
SAVEPOINT	Y	Y		Y	
SELECT INTO ⁵	Y			Y	
SET CONNECTION ^{4 5}	Y		Y		
SET OPTION ^{4 5}					Y
SET PATH	Y	Y		Y	
SET RESULT SETS ³	Y			Y	
SET SCHEMA	Y	Y		Y	
SET TRANSACTION	Y	Y		Y	
SET transition-variable ¹	Y			Y	
SET variable	Y		Y		
SQL-control-statement ²	Y			Y	
UPDATE	Y	Y		Y	
VALUES ¹	Y			Y	
VALUES INTO ⁵	Y	Y		Y	
WHENEVER ^{4 5}					Y

Notes:

1. This statement can only be used in the triggered action of a trigger.
2. This statement can only be used in an SQL function, SQL procedure, or SQL trigger.
3. This statement can only be used in a procedure.
4. This statement is not applicable in a Java program.
5. This statement is not supported in a REXX program.

SQL Statement Data Access Indication in Routines

The following table indicates whether or not an SQL statement (specified in the first column) is allowed to execute in a function or procedure with the specified SQL data access indication. If an executable SQL statement is encountered in a function or procedure defined with NO SQL, SQLSTATE 38001 is returned. For other executions contexts, SQL statements that are not supported in any context return SQLSTATE 38003. For other SQL statements not allowed in a CONTAINS SQL context, SQLSTATE 38004 is returned and in a READS SQL DATA context, SQLSTATE 38002 is returned. During creation of an SQL function or SQL procedure, a statement that does not match the SQL data access indication will cause SQLSTATE 42895 to be returned.

Table 70. SQL Statement and SQL Data Access Indication

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALTER TABLE				Y
BEGIN DECLARE SECTION	Y ¹	Y	Y	Y
CALL		Y	Y	Y
CLOSE			Y	Y
COMMENT				Y
COMMIT ³		Y	Y	Y
CONNECT (Type 1 and Type 2) ³				
CREATE ...				Y
DECLARE CURSOR	Y ¹	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE				Y
DECLARE PROCEDURE	Y ¹	Y	Y	Y
DECLARE STATEMENT	Y ¹	Y	Y	Y
DECLARE VARIABLE	Y ¹	Y	Y	Y
DELETE				Y
DESCRIBE			Y	Y
DESCRIBE TABLE			Y	Y
DISCONNECT ³				
DROP ...				Y
END DECLARE SECTION	Y ¹	Y	Y	Y
EXECUTE		Y ²	Y ²	Y
EXECUTE IMMEDIATE		Y ²	Y ²	Y
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GRANT ...				Y
HOLD LOCATOR		Y	Y	Y
INCLUDE	Y ¹	Y	Y	Y
INSERT				Y
LABEL				Y

Table 70. SQL Statement and SQL Data Access Indication (continued)

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
LOCK TABLE		Y	Y	Y
OPEN			Y	Y
PREPARE		Y	Y	Y
RELEASE CONNECTION ³				
RELEASE SAVEPOINT				Y
RENAME				Y
REVOKE ...				Y
ROLLBACK ³		Y	Y	Y
ROLLBACK TO SAVEPOINT				Y
SAVEPOINT				Y
SELECT INTO			Y	Y
SET CONNECTION ³				
SET OPTION	Y ¹	Y	Y	Y
SET PATH		Y	Y	Y
SET RESULT SETS		Y	Y	Y
SET SCHEMA			Y	Y
SET TRANSACTION		Y	Y	Y
SET variable		Y	Y	Y
UPDATE				Y
VALUES				
VALUES INTO			Y	Y
WHENEVER	Y ¹	Y	Y	Y

Notes:

1. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. It depends on the statement being executed. The statement specified for the EXECUTE statement must be a statement that is allowed in the context of the particular SQL access level in effect. For example, if the SQL access level in effect is READS SQL DATA, the statement must not be an INSERT, UPDATE, or DELETE.
3. Connection management and transaction statements are not allowed in a procedure running on a remote server. COMMIT and ROLLBACK are not allowed in an ATOMIC SQL procedure.

Considerations for Using Distributed Relational Database

This section contains information that may be useful in developing applications that use servers which are not the same product as their application requesters.

All DB2 Universal Database products support extensions to IBM SQL. Some of these extensions are product-specific, but many are already shared by more than one product or support is planned but not yet generally available.

For the most part, an application can use the statements and clauses that are supported by the database manager of the current server, even though that application might be running through the application requester of a database manager that does not support some of those statements and clauses. Restrictions to this general rule are identified by application requester:

- for DB2 UDB for OS/390 and z/OS Server application requester, see Table 71 on page 841
- for DB2 for VM and VSE Server application requester, see Table 72 on page 842
- for DB2 UDB for iSeries Server application requester, see Table 73 on page 843
- for DB2 UDB UWO application requester, see Table 74 on page 844.

Note that an 'R' in the table indicates that this SQL function is not supported in the specified environment. An 'R' in every column of the same row means that the function is available only if the current server and requester are the same product or that the statement is blocked by the application requester from being processed at the application server.

Table 71. DB2 UDB for OS/390 and z/OS Application Requester

SQL Statement or Function	DB2 UDB for OS/390 and z/OS Server	DB2 for VM and VSE Server	DB2 UDB for iSeries Server	DB2 UDB UWO Server
COMMIT HOLD	R	R	R	R
COMMIT RELEASE	R	R	R	R
CONNECT (Type 2)		Note 1		
DECLARE CURSOR WITH HOLD		R		
DECLARE STATEMENT				
DECLARE TABLE				
DECLARE VARIABLE				
DESCRIBE TABLE		R		R
DESCRIBE with USING clause				R
DISCONNECT	R	R	R	R
Extended Dynamic Statements	R	R	R	R
Large Object (LOB) Data Types		R		
BIGINT Data Types	R	R	Note 3	Note 3
ROWID Data Types		R		R
DATALINK Data Types	R	R	R	R
Distinct Data Types		R		Note 4
Non-IBM SQL host declarations		Note 2	Note 2	Note 2
PREPARE with INTO clause				
PREPARE with USING clause				R
PUT	R	R	R	R
RELEASE				
ROLLBACK HOLD	R	R	R	R
ROLLBACK RELEASE	R	R	R	R
SET CONNECTION				
SET CURRENT PACKAGESET				
SET host variable		R	R	R
SET TRANSACTION	R	R	R	R
Scrollable Cursor statements	R	R	R	R
UPDATE cursor - FOR UPDATE OF clause not specified				
WHENEVER with STOP	R	R	R	R

Notes:

- 1 The server will only be allowed read-only operations unless all the other connections are already read-only. If the server is DB2 for VM and VSE this is only a restriction if a TCP/IP connection is used.
- 2 The statement is supported if the application requester understands it.
- 3 The DB2 UDB for OS/390 and z/OS Server application requestor will process a BIGINT data type at the application server using the compatible DECIMAL(19,0) data type.
- 4 The DB2 UDB UWO application server returns the source type of the distinct type but the distinct type name is not returned.

Table 72. DB2 for VM and VSE Application Requester

SQL Statement or Function	DB2 UDB for OS/390 and z/OS Server	DB2 for VM and VSE Server	DB2 UDB for iSeries Server	DB2 UDB UWO Server
COMMIT HOLD	R	R	R	R
COMMIT RELEASE				
CONNECT (Type 2)	R	R	R	R
DECLARE CURSOR WITH HOLD		R		
DECLARE STATEMENT	R	R	R	R
DECLARE TABLE	R	R	R	R
DECLARE VARIABLE	R	R	R	R
DESCRIBE TABLE	R	R	R	R
DESCRIBE with USING clause			R	
DISCONNECT	R	R	R	R
Extended Dynamic Statements	R Note 2		R Note 2	R Note 2
Host declarations not documented in IBM SQL	R	R	R	R
Large Object (LOB) Data Types	R	R	R	R
BIGINT Data Types	R	R		
ROWID Data Types	R	R	R	R
DATALINK Data Types	R	R	R	R
Distinct Data Types	R	R	R	R
Non-IBM SQL host declarations	Note 1		Note 1	Note 1
PREPARE with INTO clause	R	R	R	R
PREPARE with USING clause	R	R	R	R
PUT				
RELEASE	R	R	R	R
ROLLBACK HOLD	R	R	R	R
ROLLBACK RELEASE				
SET CONNECTION	R	R	R	R
SET CURRENT PACKAGESET	R	R	R	R
SET host variable	R	R	R	R
SET TRANSACTION	R	R	R	R
Scrollable Cursor statements	R	R	R	R
UPDATE cursor - FOR UPDATE OF clause not specified				
WHENEVER with STOP				

Notes:

- 1 The statement is supported if the application requester understands it.
- 2 Most of the extended dynamic statements involving non-modifiable packages will work. See the DB2 for VM and VSE product library for more information.

Table 73. DB2 UDB for iSeries Application Requester

SQL Statement or Function	DB2 UDB for OS/390 and z/OS Server	DB2 for VM and VSE Server	DB2 UDB for iSeries Server	DB2 UDB UWO Server
COMMIT HOLD	R	R		R
COMMIT RELEASE	R	R	R	R
CONNECT (Type 2)		Note 1		
DECLARE CURSOR WITH HOLD		R		
DECLARE STATEMENT				
DECLARE TABLE				
DECLARE VARIABLE				
DESCRIBE TABLE		R		R
DESCRIBE with USING clause				R
DISCONNECT				
Extended Dynamic Statements	R	R	R	R
Host Variables - optional colon	R	R	R	R
Large Object (LOB) Data Types		R		R
BIGINT Data Types	R	R		
I ROWID Data Types		R		R
DATALINK Data Types	R	R		R
Distinct Data Types		R		Note 3
Non-IBM SQL host declarations	Note 2	Note 2		Note 2
PREPARE with INTO clause				
PREPARE with USING clause				R
PUT	R	R	R	R
RELEASE				
ROLLBACK HOLD	R	R		R
ROLLBACK RELEASE	R	R	R	R
SET CONNECTION				
SET CURRENT PACKAGESET	R	R	R	R
SET host variable	R	R	R	R
SET TRANSACTION	R	R		R
Scrollable Cursor statements	R	R		R
UPDATE cursor - FOR UPDATE OF clause not specified	R	R		
WHENEVER with STOP	R	R	R	R
Notes:				
1	The server will only be allowed read-only operations unless all the other connections are already read-only. If the server is DB2 for VM and VSE this is only a restriction if a TCP/IP connection is used.			
2	The statement is supported if the application requester understands it.			
3	The DB2 UDB UWO application server returns the source type of the distinct type but the distinct type name is not returned.			

Table 74. DB2 UDB UWO Application Requester

SQL Statement or Function	DB2 UDB for OS/390 and z/OS Server	DB2 for VM and VSE Server	DB2 UDB for iSeries Server	DB2 UDB UWO Server
COMMIT HOLD	R	R	R	R
COMMIT RELEASE	R	R	R	R
CONNECT (Type 2)		Note 1		
DECLARE CURSOR WITH HOLD		R		
DECLARE STATEMENT	R	R	R	R
DECLARE TABLE	R	R	R	R
DECLARE VARIABLE	R	R	R	R
DESCRIBE TABLE	R	R	R	R
DESCRIBE with USING clause	R	R	R	R
DISCONNECT				
Extended Dynamic Statements	R	R	R	R
Host Variables - optional colon	R	R	R	R
Large Object (LOB) Data Types		R		
BIGINT Data Types	R	R		
ROWID Data Types	R	R	R	R
DATALINK Data Types	R	R	R	R
Distinct Data Types		R		Note 3
Non-IBM SQL host declarations	Note 2	Note 2	Note 2	
PREPARE with INTO clause				
PREPARE with USING clause	R	R	R	R
PUT	R	R	R	R
RELEASE				
ROLLBACK HOLD	R	R	R	R
ROLLBACK RELEASE	R	R	R	R
SET CONNECTION				
SET CURRENT PACKAGESET				
SET host variable	R	R	R	R
SET TRANSACTION	R	R	R	R
Scrollable Cursor statements	R	R	R	R
UPDATE cursor - FOR UPDATE OF clause not specified	R	R		
WHENEVER with STOP	R	R	R	R

Notes:

- 1 The server will only be allowed read-only operations unless all the other connections are already read-only. If the server is DB2 for VM and VSE, this is only a restriction if a TCP/IP connection is used.
- 2 The statement is supported if the application requester understands it.
- 3 The DB2 UDB UWO application server returns the source type of the distinct type but the distinct type name is not returned.

CONNECT (Type 1) and CONNECT (Type 2) Differences

There are two types of CONNECT statements. They have the same syntax, but they have different semantics:

- CONNECT (Type 1) is used for remote unit of work. See “Remote Unit of Work” on page 27.
- CONNECT (Type 2) is used for distributed unit of work. See “CONNECT (Type 2)” on page 421.

The following table summarizes the differences between CONNECT (Type 1) and CONNECT (Type 2) rules:

Table 75. CONNECT (Type 1) and CONNECT (Type 2) Differences

Type 1 Rules	Type 2 Rules
CONNECT statements can only be executed when the activation group is in the connectable state. No more than one CONNECT statement can be executed within the same unit of work.	There are no rules about the connectable state. More than one CONNECT statement can be executed within the same unit of work.
If the CONNECT statement fails because the server name is not listed in the local directory, the connection state of the activation group is unchanged.	If a CONNECT statement fails, the current SQL connection is unchanged and any subsequent SQL statements are executed by the current server.
If a CONNECT statement fails because the activation group is not in the connectable state, the SQL connection status of the activation group is unchanged.	
If a CONNECT statement fails for any other reason, the activation group is placed in the unconnected state.	
CONNECT ends all existing connections of the activation group. Accordingly, CONNECT also closes any open cursors for that activation group.	CONNECT does not end connections and does not close cursors.
A CONNECT to the current server will succeed if the application group is the connectable state.	A CONNECT to an existing SQL connection of the activation group is an error. Thus, a CONNECT to the current server is an error.

Determining the CONNECT rules that apply

A program preparation option is used to specify the type of CONNECT that will be performed by a program. The program preparation option is specified using the RDBCNNMTH parameter on the CRTSQLxxx command.

Connecting to Servers That Only Support Remote Unit of Work

CONNECT (Type 2) connections to servers that only support remote unit of work might result in connections that are read-only.

If a CONNECT (Type 2) is performed to a server that only supports remote unit of work⁸⁰:

⁸⁰. DB2 UDB for iSeries using the initial DRDA support for native TCP/IP is an example of a server that supports only remote unit of work.

- The connection allows read-only operations if, at the time of the connect, there are any dormant connections that allow updates. In this case, the connection does not allow updates.
- Otherwise, the connection allows updates.

If a CONNECT (Type 2) is performed to a server that supports distributed unit of work:

- The connection allows read-only operations when there are dormant connections that allow updates to servers that only support remote unit of work. In this case, the connection allows updates as soon as the dormant connection is ended.
- Otherwise, the connection allows updates.

Appendix D. SQL Descriptor Area (SQLDA)

An SQLDA is a set of variables that is required for execution of the SQL DESCRIBE statement, and it may optionally be used by the PREPARE, OPEN, CALL, FETCH, and EXECUTE statements. An SQLDA can be used in a DESCRIBE or PREPARE statement, altered with the addresses of host variables, and then used again in a FETCH statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C (and C++), COBOL, ILE RPG/400, PL/I, and REXX. In REXX, the SQLDA is somewhat different than in the other languages; for information on the use of SQLDAs in REXX, see the SQL Programming with Host Languages book.

The meaning of the information in an SQLDA depends on its use.

- When an SQLDA is used in a DESCRIBE or PREPARE statement, an SQLDA provides information to an application program about a prepared *select-statement*. Each column of the result table is described in an SQLVAR occurrence or set of related SQLVAR occurrences.
- In OPEN, EXECUTE, CALL, and FETCH, an SQLDA provides information to the database manager about storage areas for input or output data. Each storage area is described in the SQLVARs.
 - For OPEN and EXECUTE, each SQLVAR occurrence or set of related SQLVAR occurrences describes a storage area that is used to contain an input value which is substituted for a parameter marker in the associated SQL statement that was previously prepared.
 - For FETCH, each SQLVAR occurrence or set of related SQLVAR occurrences describes a storage area that is used to contain an output value from a row of the result table.
 - For CALL, each SQLVAR occurrence or set of related SQLVAR occurrences describes a storage area that is used to contain an input or output value (or both) that corresponds to an argument in the argument list for the procedure.

An SQLDA consists of four variables in a header followed by an arbitrary number of occurrences of a *base SQLVAR*. When the SQLDA describes either LOBs or distinct types the base SQLVARs are followed by the same number of occurrences of an *extended SQLVAR*.

Base SQLVAR entry

The base SQLVAR entry is always present. The fields of this entry contain the base information about the column or host variable such as data type code, length attribute (except for LOBs), column name (or label), CCSID, host variable address, and indicator variable address.

Extended SQLVAR entry

The extended SQLVAR entry is needed (for each column) if the result includes any LOB or distinct type columns. For distinct types, the extended SQLVAR contains the distinct type name. For LOBs, the extended SQLVAR contains the length attribute of the host variable and a pointer to the buffer that contains the actual length. If locators or file reference variables are used to represent LOBs, an extended SQLVAR is not necessary.

The extended SQLVAR entry is also needed for each column when:

SQLDA

- USING BOTH is specified, which indicates that column names and labels are returned.
- USING ALL is specified, which indicates that column names, labels, and system column names are returned.

The fields in the extended SQLVAR that return LOB and distinct type information do not overlap, and the fields that return LOB and label information do not overlap. Depending on the combination of labels, LOBs and distinct types, more than one extended SQLVAR entry per column may be required to return the information. See “Determining How Many SQLVAR Occurrences are Needed” on page 850.

Field Descriptions in an SQLDA Header

An SQLDA consists of four variables in a header structure followed by an arbitrary number of occurrences of a sequence of five variables collectively named SQLVAR. In OPEN, CALL, FETCH, and EXECUTE, each occurrence of SQLVAR describes a host variable. In PREPARE and DESCRIBE, each occurrence describes a column of a result table.

The SQL INCLUDE statement provides the following field names:

Table 76. Field Descriptions for an SQLDA Header

C Name ⁸¹ PL/I Name COBOL Name	Field Data Type	Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
sqldaid SQLDAID	CHAR(8)	An 'eye catcher' for storage dumps, containing 'SQLDA ' . The 7th byte of the SQLDAID can be used to determine whether more than one SQLVAR entry is needed for each column. For details, see "Determining How Many SQLVAR Occurrences are Needed" on page 850.	A '2' in the 7th byte indicates that two SQLVAR entries were allocated for each column. A '3' in the 7th byte indicates that three SQLVAR entries were allocated for each column. A '4' in the 7th byte indicates that four SQLVAR entries were allocated for each column.
sqldabc SQLDABC	INTEGER	Length of the SQLDA.	Number of bytes of storage allocated for the SQLDA. Enough storage must be allocated to contain SQLN occurrences. SQLDABC must be set to a value greater than or equal to $16 + \text{SQLN} * (80)$, where 80 is the length of an SQLVAR occurrence. If LOBs or distinct types are specified, there must be two SQLVAR entries for each parameter marker.
sqln SQLN	SMALLINT	Unchanged by the database manager. Must be set to a value greater than or equal to zero before the PREPARE or DESCRIBE statement is executed. It should be set to a value that is greater than or equal to the number of columns in the result or a multiple of the number of columns in the result when multiple sets of SQLVAR entries are necessary. 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. If LOBs or distinct types are specified, there must be two SQLVAR entries for each parameter marker and SQLN must be set to two times the number of parameter markers.
sqld SQLD	SMALLINT	The number of columns described by occurrences of SQLVAR (zero if the statement being described is not a select-statement).	Number of host variables described by SQLVAR to be used in the SQLDA when executing this statement. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

81. In this column, the lowercase name is the C Name. The uppercase name is the COBOL, PL/I, or RPG Name.

Determining How Many SQLVAR Occurrences are Needed

The number of SQLVAR occurrences needed depends on the statement that the SQLDA was provided for and the data types of the columns or parameters being described. See the tables above for more information.

The 7th byte of SQLDAID is always set to the number of sets of SQLVARs necessary.

If SQLD is not set to a sufficient number of SQLVAR occurrences:

- SQLD is set to the total number of SQLVAR occurrences needed for all sets.
- A warning (SQLSTATE 01594) is returned in the SQLCODE field of the SQLCA if at least enough SQLVARs were specified for the Base SQLVAR Entries. The Base SQLVAR entries are returned, but no extended SQLVARs are returned.
- A warning (SQLSTATE 01005) is returned in the SQLCODE field of the SQLCA if enough SQLVARs were not specified for even the Base SQLVAR Entries. No SQLVAR entries are returned.

Table 77 on page 851, Table 78 on page 851, and Table 79 on page 851 show how to map the base and extended SQLVAR entries. For an SQLDA that contains both base and extended SQLVAR entries, the base SQLVAR entries are in the first block, followed by a block of extended SQLVAR entries, which if necessary, are followed by a second or third block of extended SQLVAR entries. In each block, the number of occurrences of the SQLVAR entry is equal to the value in SQLD even though many of the extended SQLVAR entries might be unused.

Table 77. Contents of SQLVAR Arrays for USING NAMES, USING SYSTEM NAMES, USING LABELS or USING ANY

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)	Third Set (Extended)	Fourth Set (Extended)
No	No	Blank	n	Column names, system column names, or labels	Not used	Not used	Not used
Yes	No	2	2n	Column names, system column names, or labels	LOBs	Not used	Not used
No	Yes	2	2n	Column names, system column names, or labels	Distinct types	Not used	Not used
Yes	Yes	2	2n	Column names, system column names, or labels	LOBs and distinct types	Not used	Not used

Table 78. Contents of SQLVAR Arrays for USING BOTH

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)	Third Set (Extended)	Fourth Set (Extended)
No	No	2	2n	Column names	Labels	Not used	Not used
Yes	No	2	2n	Column names	LOBs and labels	Not used	Not used
No	Yes	3	3n	Column names	Distinct types	Labels	Not used
Yes	Yes	3	3n	Column names	LOBs and distinct types	Labels	Not used

Table 79. Contents of SQLVAR Arrays for USING ALL

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)	Third Set (Extended)	Fourth Set (Extended)
No	No	3	3n	System column names	Labels	Column names	Not used
Yes	No	3	3n	System column names	LOBs and labels	Column names	Not used
No	Yes	4	4n	System column names	Distinct types	Labels	Column names
Yes	Yes	4	4n	System column names	LOBs and distinct types	Labels	Column names

SQLDA

Field Descriptions in an Occurrence of SQLVAR

Table 80. Field Descriptions for an SQLVAR

C Name ⁸²			
PL/I Name	Field Data Type	Usage in DESCRIBE and PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, and EXECUTE (set by the user prior to executing the statement)
sqltype SQLTYPE	SMALLINT	Indicates the data type of the column and whether it can contain nulls. For a description of the type codes, see Table 82 on page 854. For a distinct type, the data type on which the distinct type is based is placed in this field. The base SQLVAR contains no indication that this is part of the description of a distinct type.	Indicates the data type of the host variable and whether an indicator variable is provided. For a description of the type codes, see Table 82 on page 854.
sqllen SQLLEN	SMALLINT	The length attribute of the column. For datetime columns, the length of the string representation of the values. See Table 82 on page 854. For a LOB, the value is 0 regardless of the length attribute of the LOB. Field SQLLONGLEN in the extended SQLVAR entry contains the length attribute of the LOB.	The length attribute of the host variable. See Table 82 on page 854. For a LOB, the value is 0 regardless of the length attribute of the LOB. Field SQLLONGLEN in the extended SQLVAR entry contains the length attribute of the LOB.
sqlres SQLRES	CHAR(12)	Reserved. Provides boundary alignment for SQLDATA.	Reserved. Provides boundary alignment for SQLDATA.
sqldata SQLDATA	pointer	The CCSID of a string column as described in Table 83 on page 856.	Contains the address of the host variable. For LOB host variables, if the SQLDATALEN field in the extended SQLVAR is null, this points to the four-byte LOB length, followed immediately by the LOB data. If the SQLDATALEN field in the extended SQLVAR is not null, this points to the LOB data and the SQLDATALEN field points to the four-byte LOB length.
sqlind SQLIND	pointer	Reserved	Contains the address of the indicator variable. Not used if there is no indicator variable (as indicated by an even value of SQLTYPE).
sqlname SQLNAME	VARCHAR(30)	The unqualified name of the column. If the column does not have a name, a string is constructed from the expression and returned. The name is case sensitive and does not contain surrounding delimiters.	Contains the CCSID of the host variable as described in Table 83 on page 856.

82. In this column, the lowercase name is the C Name. The uppercase name is the PL/I, COBOL, and RPG Name.

Table 81. Field Descriptions for an Extended SQLVAR

C Name ⁸³	Field Data Type	Usage in DESCRIBE and PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, and EXECUTE (set by the user prior to executing the statement)
len.sqllonglen SQLLONGL SQLLONGLEN	INTEGER	The length attribute of a LOB column.	The length attribute of a LOB host variable. The database manager ignores the SQLLEN field in the base SQLVAR for these data types. The length attribute indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB.
*	CHAR(12)	Reserved. Provides boundary alignment for SQLDATALEN.	Reserved. Provides boundary alignment for SQLDATALEN.
*	pointer	Reserved.	Reserved.
sqldataen SQLDATAL SQLDATALEN	pointer	Not used.	Used only for LOB host variables. If the value of this field is not null, this field points to a four-byte long buffer that contains the actual length of the LOB in bytes (even for DBCLOBs). The SQLDATA field in the matching base SQLVAR then points to the LOB data. If the value of this field is null, the actual length of the LOB is stored in the first four bytes pointed to by the SQLDATA field in the matching base SQLVAR, and the LOB data immediately follows the four-byte length. The actual length indicates the number of bytes for a BLOB or CLOB and the number of double-byte characters for a DBCLOB. Regardless of whether this field is used, field SQLLONGLEN must be set.
sqldatatype_name SQLTNAME SQLDATATYPE-NAME	VARCHAR (30)	The SQLTNAME field of the extended SQLVAR is set to one of the following: <ul style="list-style-type: none"> For a distinct type column, the database manager sets this to the fully qualified distinct type name. If the qualified name is longer than 30 bytes, it is truncated. For a label, the database manager sets this to the first 20 bytes of the label. For a column name, the database manager sets this to the column name. 	Not used.

83. In this column, the lowercase name is the C Name. The first uppercase name is the PL/I and RPG Name. The second uppercase name is the COBOL Name.

SQLTYPE and SQLLEN

The following table shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In PREPARE and DESCRIBE, an even value of SQLTYPE means the column does not allow nulls, and an odd value means the column does allow nulls.

Note: In an SQLDA used in DESCRIBE or PREPARE statements, an odd value is returned for an expression if one operand is nullable or if the expression may result in a -2 mapping-error null value.

In FETCH, OPEN, CALL, and EXECUTE, 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 82. SQLTYPE and SQLLEN values for PREPARE, DESCRIBE, FETCH, OPEN, CALL, or EXECUTE

SQLTYPE	For PREPARE and DESCRIBE		For FETCH, OPEN, CALL, and EXECUTE	
	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 time stamp	Length attribute of the host variable
396/397	DataLink	Length attribute of the column	DataLink	Length attribute of the host variable
400/401	Not Applicable	Not Applicable	NUL-terminated graphic string	Length attribute of the host variable
404/405	BLOB	0 ⁸⁵	BLOB	Not used. ⁸⁵
408/409	CLOB	0 ⁸⁵	CLOB	Not used. ⁸⁵
412/413	DBCLOB	0 ⁸⁵	DBCLOB	Not used. ⁸⁵
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	Not Applicable	Not Applicable	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

Table 82. SQLTYPE and SQLLEN values for PREPARE, DESCRIBE, FETCH, OPEN, CALL, or EXECUTE (continued)

SQLTYPE	For PREPARE and DESCRIBE		For FETCH, OPEN, CALL, and EXECUTE	
	COLUMN DATA TYPE	SQLLEN	HOST VARIABLE DATA TYPE	SQLLEN
476/477	Not Applicable	Not Applicable	PASCAL L-string	Length attribute of the host variable
480/481	Floating point	4 for single precision, 8 for double precision	Floating point	4 for single precision, 8 for double precision
484/485	Packed decimal	Precision in byte 1; scale in byte 2	Packed decimal	Precision in byte 1; scale in byte 2
488/489	Zoned decimal	Precision in byte 1; scale in byte 2	Zoned decimal	Precision in byte 1; scale in byte 2
492/493	Big integer	8 ⁸⁴	Big integer	8
496/497	Large integer	4 ⁸⁴	Large integer	4
500/501	Small integer	2 ⁸⁴	Small integer	2
504/505	Not Applicable	Not Applicable	DISPLAY SIGN LEADING SEPARATE	Precision in byte 1; scale in byte 2
I 904/905	ROWID	40	ROWID	40
916/917	Not Applicable	Not Applicable	BLOB file reference variable	267
920/921	Not Applicable	Not Applicable	CLOB file reference variable	267
924/925	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

84. Binary numbers can be represented in the SQLDA with a length of 2, 4, or 8, or with the precision in byte 1 and the scale in byte 2. If the first byte is greater than x'00', it indicates precision and scale.

85. Field SQLLONGLEN in the extended SQLVAR contains the length attribute of the column.

CCSID Values in SQLDATA or SQLNAME

In the OPEN, FETCH, CALL, and EXECUTE statements, the SQLNAME field of the SQLVAR element can be used to specify a CCSID for string host variables. If the SQLNAME field is used to specify a CCSID, the SQLNAME length must be set to 8. In addition, the first 4 bytes of SQLNAME must be set as described in the table below. If no CCSID is specified, the job CCSID is used.

In the DESCRIBE, DESCRIBE TABLE, and PREPARE statements, the SQLDATA field of the SQLVAR element contains the CCSID of the column of the result table if that column is a string column. The CCSID is located in bytes 3 and 4 as described in Table 83.

Table 83. CCSID values for SQLDATA or SQLNAME

Data Type	Encoding Scheme	Bytes 1 & 2	Bytes 3 & 4
Character	SBCS data	X'0000'	ccsid
Character	Mixed data	X'0000'	ccsid
Character	Bit data	X'0000'	65535
Graphic	Not Applicable	X'0000'	ccsid
Any other data type	Not Applicable	Not Applicable	Not Applicable

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 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 receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Table 84. Compatible Data Types for Unsupported Data Types

Data Type	Compatible Data Type
BIGINT	DECIMAL(19,0)
ROWID	VARCHAR(40) FOR BIT DATA

INCLUDE SQLDA Declarations

For C and C++

In C and C++, INCLUDE SQLDA declarations are equivalent to the following:

```

#ifndef SQLDASIZE
struct sqlda
{
    unsigned char  sqldaid[8];
    long          sqldabc;
    short         sqln;
    short         sqld;
    struct sqlvar
    {
        short      sqltype;
        short      sqllen;
        unsigned char *sqldata;
        short      *sqlind;
        struct sqlname
        {
            short      length;
            unsigned char data[30];
        } sqlname;
    } sqlvar[1];
};

struct sqlvar2
{ struct
    { long          sqllonglen;
      char         reserve1[28];
    } len;
  char *sqldataen;
  struct sqldistinct_type
  { short          length;
    unsigned char data[30];
  } sqldatatype_name;
};

#define SQLDASIZE(n) (sizeof(struct sqlda)+(n-1) * sizeof(struct sqlvar))
#endif

```

Figure 11. INCLUDE SQLDA Declarations for C and C++ (Part 1 of 3)

SQLDA

```

/*****
/* Macros for using the sqlvar2 fields. */
/*****

/*****
/* '2' in the 7th byte of sqldaid indicates a doubled number of */
/* sqlvar entries. */
/* '3' in the 7th byte of sqldaid indicates a tripled number of */
/* sqlvar entries. */
/*****
#define SQLDOUBLED '2'
#define SQLSINGLED ' '

/*****
/* GETSQLDOUBLED(daptr) returns 1 if the SQLDA pointed to by */
/* daptr has been doubled, or 0 if it has not been doubled. */
/*****
#define GETSQLDOUBLED(daptr) (((daptr)->sqldaid[6]== \
(char) SQLDOUBLED) ? \
(1) : \
(0))

/*****
/* SETSQLDOUBLED(daptr, SQLDOUBLED) sets the 7th byte of sqldaid */
/* to '2'. */
/* SETSQLDOUBLED(daptr, SQLSINGLED) sets the 7th byte of sqldaid */
/* to be a ' '. */
/*****
#define SETSQLDOUBLED(daptr, newvalue) \
(((daptr)->sqldaid[6] =(newvalue)))

/*****
/* GETSQLDALONGLEN(daptr,n) returns the data length of the nth */
/* entry in the sqlda pointed to by daptr. Use this only if the */
/* sqlda was doubled or tripled and the nth SQLVAR entry has a */
/* LOB datatype. */
/*****
#define GETSQLDALONGLEN(daptr,n) ((long) (((struct sqlvar2 *) \
&((daptr)->sqlvar[(n) +((daptr)->sqld)])) ->len.sqllonglen))

/*****
/* SETSQLDALONGLEN(daptr,n,len) sets the sqllonglen field of the */
/* sqlda pointed to by daptr to len for the nth entry. Use this only */
/* if the sqlda was doubled or tripled and the nth SQLVAR entry has */
/* a LOB datatype. */
/*****
#define SETSQLDALONGLEN(daptr,n,length) { \
struct sqlvar2 *var2ptr; \
var2ptr = (struct sqlvar2 *) &((daptr)->sqlvar[(n)+ \
((daptr)->sqld)]); \
var2ptr->len.sqllonglen = (long) (length); \
}

/*****
/* SETSQLDALENPTR(daptr,n,ptr) sets a pointer to the data length for */
/* the nth entry in the sqlda pointed to by daptr. */
/* Use this only if the sqlda has been doubled or tripled. */
/*****
#define SETSQLDALENPTR(daptr,n,ptr) { \
struct sqlvar2 *var2ptr; \
var2ptr = (struct sqlvar2 *) &((daptr)->sqlvar[(n)+ \
((daptr)->sqld)]); \
var2ptr->sqldatalen = (char *) ptr; \
}

```

Figure 11. INCLUDE SQLDA Declarations for C and C++ (Part 2 of 3)

```

/*****
/* GETSQLDALENPTR(daptr,n) returns a pointer to the data length for */
/* the nth entry in the sqlda pointed to by daptr. Unlike the inline */
/* value (union sql8bytelen len), which is 8 bytes, the sqldatalen */
/* pointer field returns a pointer to a long (4 byte) integer. */
/* If the SQLDATALEN pointer is zero, a NULL pointer is be returned. */
/* */
/* NOTE: Use this only if the sqlda has been doubled or tripled. */
/*****
#define GETSQLDALENPTR(daptr,n) ( \
    ((struct sqlvar2 *) &(daptr)->sqlvar[(n) + \
    (daptr)->sqld]->sqldatalen == NULL) ? \
    ((long *) NULL) : ((long *) ((struct sqlvar2 *) \
    &(daptr)->sqlvar[(n) + (daptr) ->sqld])->sqldatalen))

```

Figure 11. INCLUDE SQLDA Declarations for C and C++ (Part 3 of 3)

SQLDA

For COBOL

In COBOL, INCLUDE SQLDA declarations are equivalent to the following:

```
1 SQLDA.  
  05 SQLDAID      PIC X(8).  
  05 SQLDABC      PIC S9(9) BINARY.  
  05 SQLN         PIC S9(4) BINARY.  
  05 SQLD         PIC S9(4) BINARY.  
  05 SQLVAR OCCURS 0 TO 409 TIMES DEPENDING ON SQLD.  
    10 SQLTYPE    PIC S9(4) BINARY.  
    10 SQLLEN     PIC S9(4) BINARY.  
    10 FILLER     REDEFINES SQLLEN.  
      15 SQLPRECISION PIC X.  
      15 SQLSCALE    PIC X.  
    10 SQLRES     PIC X(12).  
    10 SQLDATA    POINTER.  
    10 SQLIND     POINTER.  
    10 SQLNAME.  
      49 SQLNAMEL PIC S9(4) BINARY.  
      49 SQLNAMEC PIC X(30).
```

Figure 12. INCLUDE SQLDA Declarations for COBOL

For ILE COBOL

In ILE COBOL, INCLUDE SQLDA declarations are equivalent to the following:

```
1 SQLDA.  
  05 SQLDAID      PIC X(8).  
  05 SQLDABC      PIC S9(9) BINARY.  
  05 SQLN         PIC S9(4) BINARY.  
  05 SQLD         PIC S9(4) BINARY.  
  05 SQLVAR OCCURS 0 TO 409 TIMES DEPENDING ON SQLD.  
    10 SQLVAR1.  
      15 SQLTYPE    PIC S9(4) BINARY.  
      15 SQLLEN     PIC S9(4) BINARY.  
      15 FILLER     REDEFINES SQLLEN.  
        20 SQLPRECISION PIC X.  
        20 SQLSCALE    PIC X.  
      15 SQLRES     PIC X(12).  
      15 SQLDATA    POINTER.  
      15 SQLIND     POINTER.  
      15 SQLNAME.  
        49 SQLNAMEL PIC S9(4) BINARY.  
        49 SQLNAMEC PIC X(30).  
    10 SQLVAR2 REDEFINES SQLVAR1.  
      15 SQLVAR2-RESERVED-1 PIC S9(9) BINARY.  
      15 SQLLONGLEN      REDEFINES SQLVAR2-RESERVED-1  
                          PIC S9(9) BINARY.  
      15 SQLVAR2-RESERVED-2 PIC X(28).  
      15 SQLDATALEN      POINTER.  
      15 SQLDATATYPE-NAME.  
        49 SQLDATATYPE-NAMEL PIC S9(4) BINARY.  
        49 SQLDATATYPE-NAMEC PIC X(30).
```

Figure 13. INCLUDE SQLDA Declarations for ILE COBOL

For PL/I

In PL/I, INCLUDE SQLDA declarations are equivalent to the following:

```

DCL 1 SQLDA BASED(SQLDAPTR),
    2 SQLDAID    CHAR(8),
    2 SQLDABC    BIN FIXED(31),
    2 SQLN       BIN FIXED,
    2 SQLD       BIN FIXED,
    2 SQLVAR     (99),
    3 SQLTYPE    BIN FIXED,
    3 SQLLEN     BIN FIXED,
    3 SQLRES     CHAR(12),
    3 SQLDATA    PTR,
    3 SQLIND     PTR,
    3 SQLNAME    CHAR(30) VAR,

1 SQLDA2 BASED(SQLDAPTR),
    2 SQLDAID2   CHAR(8),
    2 SQLDABC2   FIXED(31) BINARY,
    2 SQLN2      FIXED(15) BINARY,
    2 SQLD2      FIXED(15) BINARY,
    2 SQLVAR2    (99),
    3 SQLBIGLEN,
    4 SQLLONGL  FIXED(31) BINARY,
    4 SQLRSVDL  FIXED(31) BINARY,
    3 SQLDATAL  POINTER,
    3 SQLTNAME  CHAR(30) VAR;

DECLARE SQLSIZE    FIXED(15) BINARY;
DECLARE SQLDAPTR  PTR;
DECLARE SQLDOUBLED CHAR(1)  INITIAL('2') STATIC;
DECLARE SQLSINGLED CHAR(1)  INITIAL(' ') STATIC;

```

Figure 14. INCLUDE SQLDA Declarations for PL/I

SQLDA

For ILE RPG/400

In ILE RPG/400, INCLUDE SQLDA declarations are equivalent to the following:

```
D*      SQL Descriptor area
D SQLDA      DS
D  SQLDAID      1      8A
D  SQLDABC      9      12B 0
D  SQLN        13      14B 0
D  SQLD        15      16B 0
D  SQL_VAR     80A    DIM(SQL_NUM)
D          17      18B 0
D          19      20B 0
D          21      32A
D          33      48*
D          49      64*
D          65      66B 0
D          67      96A
D*
D SQLVAR      DS
D  SQLTYPE      1      2B 0
D  SQLLEN      3      4B 0
D  SQLRES      5      16A
D  SQLDATA     17      32*
D  SQLIND      33      48*
D  SQLNAMELEN  49      50B 0
D  SQLNAME     51      80A
D*
D SQLVAR2     DS
D  SQLLONGL    1      4B 0
D  SQLRSVDL    5      32A
D  SQLDATAL    33      48*
D  SQLTNAMELN  49      50B 0
D  SQLTNAME    51      80A
D* End of SQLDA
```

Figure 15. INCLUDE SQLDA Declarations for ILE RPG/400

The user is responsible for the definition of SQL_NUM. SQL_NUM must be defined as a numeric constant with the dimension required for SQL_VAR.

Since RPG does not support structures within arrays, the SQLDA generates three data structures. The second and third data structures are used to setup/reference the part of the SQLDA which contains the field descriptions.

To set the field descriptions of the SQLDA the program sets up the field description in the subfields of SQLVAR (or SQLVAR2) and then does a MOVEA of SQLVAR (or SQLVAR2) to SQL_VAR, n where n is the number of the field in the SQLDA. This is repeated until all the field descriptions are set.

When the SQLDA field descriptions are to be referenced the user does a MOVEA of SQL_VAR, n to SQLVAR (or SQLVAR2) where n is the number of the field description to be processed.

Appendix E. CCSID Values

The following tables describe the CCSIDs and conversions provided by the IBM relational database products. For more information, see “Character Conversion” on page 32.

The following list defines the symbols used in the DB2 UDB product column in the following tables:

X	Indicates that the conversion tables exist to convert from or to that CCSID. This also implies that this CCSID can be used to tag local data.
C	Indicates that conversion tables exist to convert from that CCSID to another CCSID. This also implies that this CCSID cannot be used to tag local data, because the CCSID is in a foreign encoding scheme (for example, a PC-Data CCSID such as 850 cannot be used to tag local data in DB2 UDB for iSeries).
blank	Indicates that the specific product does not support the CCSID at all. Such a CCSID must not be used unless interoperability with the specific product is not necessary.

This information is current as of the publishing date of this book for the CCSIDs listed. Additional CCSIDs may have been added since the publishing date and are not in the lists below. Alexis would like to eliminate all the separate UWO columns and merge them into one. He says that every DB2 supports every CCSID. The Hieplaplis The DB2 UDB UWO books, however, only show those CCSIDs that fit well with the particular operating system. The way to view this is that right now, these tables for DB2 UDB UWO show the preferred CCSIDs for a operating system, but not all. Revisit this decision next time when we have more time. The key to the DB2 UDB for iSeries values in the chart is that any CCSID that has a conversion supported in the DB2 UDB for iSeries books can be tagged with that CCSID, with the exception of the Far East CCSIDs where there is a specific list in the Installation Guide.

CCSID Values

Table 85. Universal Character Set (UTF-8, UTF-16, and UCS-2)

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
1200	UTF-16	X	C	X	X	X	X	X	X	X	X
1208	UTF-8 Level 3	X	C	X	X	X	X	X	X	X	X
13488	UCS-2 Level 1	C	X	C *	C *	C *	C *	C *	C *	C *	C *

Note: * In DB2 UDB UWO, 13488 is only used to tag the GRAPHIC column of eucJP and eucTW databases. In DB2 UDB UWO Unicode database, 1200 is used to tag the GRAPHIC column and 1208 to tag the CHAR column. DB2 UDB UWO is UTF-16 tolerant but not enabled, meaning customers can store and retrieve UTF-16 data in a Unicode GRAPHIC column in a lossless manner, however it is the customer's responsibility to properly parse the mixed 16-/32-bit UTF-16 data. DB2 UDB UWO currently assumes the GRAPHIC data is in UCS-2. When Unicode data is flowed on the wire via DRDA, 1208 is used for both SBCS and mixed, and 1200 for DBCS.

Table 86. CCSIDs for EBCDIC Group 1 (Latin-1) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
37	USA, Canada, Netherlands, Portugal, Brazil, Australia, New Zealand	X	X	C	C	C	C	C	C	C	C
256	Word Processing, Netherlands	X	X								
273	Austria, Germany	X	X	C	C	C	C	C	C	C	C
274	Belgium	X		C	C	C	C	C	C	C	C
277	Denmark, Norway	X	X	C	C	C	C	C	C	C	C
278	Finland, Sweden	X	X	C	C	C	C	C	C	C	C
280	Italy	X	X	C	C	C	C	C	C	C	C
284	Spain, Latin America (Spanish)	X	X	C	C	C	C	C	C	C	C
285	United Kingdom	X	X	C	C	C	C	C	C	C	C
297	France	X	X	C	C	C	C	C	C	C	C
500	Belgium, Canada, Switzerland, International Latin-1	X	X	C	C	C	C	C	C	C	C
871	Iceland	X	X	C	C	C	C	C	C	C	C
924	Latin-0	X	X								
1047	Latin-0 (with Euro)	X									
1140	USA, Canada, Netherlands, Portugal, Brazil, Australia, New Zealand	X	X	C	C	C	C	C	C	C	C
1141	Austria, Germany	X	X	C	C	C	C	C	C	C	C
1142	Denmark, Norway	X	X	C	C	C	C	C	C	C	C
1143	Finland, Sweden	X	X	C	C	C	C	C	C	C	C
1144	Italy	X	X	C	C	C	C	C	C	C	C
1145	Spain, Latin America (Spanish)	X	X	C	C	C	C	C	C	C	C
1146	United Kingdom	X	X	C	C	C	C	C	C	C	C
1147	France	X	X	C	C	C	C	C	C	C	C
1148	Belgium, Canada, Switzerland, International Latin-1	X	X	C	C	C	C	C	C	C	C
1149	Iceland	X	X	C	C	C	C	C	C	C	C

CCSID Values

Table 87. CCSIDs for PC-Data and ISO Group 1 (Latin-1) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
437	USA	X	C	X	C	C	C	C	C	C	C
819	Latin-1 countries (ISO 8859-1)	X	C	C	X	X	X	C	X	X	X
850	Latin Alphabet Number 1; Latin-1 countries	X	C	X	X	C	C	C	C	C	C
858	Latin Alphabet Number 1; Latin-1 countries (with Euro)	X	C								
860	Portugal (850 subset)	X	C	X	C	C	C	C	C	C	C
861	Iceland	X	C								
863	Canada (850 subset)	X	C	X	C	C	C	C	C	C	C
865	Denmark, Norway, Finland, Sweden	X	C								
923	Latin-0	X	C	C	X	X	X	C	C	C	X
1009	IRV 7-bit	X	C								
1010	France 7-bit	X	C								
1011	Germany 7-bit	X	C								
1012	Italy 7-bit	X	C								
1013	United Kingdom 7-bit	X	C								
1014	Spain 7-bit	X	C								
1015	Portugal 7-bit	X	C								
1016	Norway 7-bit	X	C								
1017	Denmark 7-bit	X	C								
1018	Finland and Sweden 7-bit	X	C								
1019	Belgium and Netherlands 7-bit	X	C								
1051	HP Emulation	X	C	C	C	X	C	C	C	C	C
1252	Windows** Latin-1	X	C	C	C	C	C	X	C	C	C
1275	Macintosh** Latin-1	X	C								
5348	Windows Latin-1(with Euro)	X									

Table 88. CCSIDs for EBCDIC Group 1a (Non-Latin-1 SBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
420	Arabic (Type 4)Visual LTR	X	X	C	C	C	C	C	C	C	C
423	Greek	X	X	C	C	C	C	C	C	C	C
424	Hebrew(Type 4)	X	X	C	C	C	C	C	C	C	C
425	Arabic (Type 5)			V8	V8	V8	V8	V8	V8	V8	V8
870	Latin-2 Multilingual	X	X	C	C	C	C	C	C	C	C
875	Greek	X	X	C	C	C	C	C	C	C	C
880	Cyrillic Multilingual	X	X								
905	Turkey Latin-3 Multilingual	X	X								
918	Urdu	X	X								
1025	Cyrillic Multilingual	X	X	C	C	C	C	C	C	C	C
1026	Turkey Latin-5	X	X	C	C	C	C	C	C	C	C
1097	Farsi	X	X								
1112	Baltic Multilingual	X	X	C	C	C	C	C	C	C	C
1122	Estonia	X	X	C	C	C	C	C	C	C	C
1123	Ukraine	X	X	C	C	C	C	C	C	C	C
1137	Devanagari	X	X	C	C	C	C	C	C	C	C
1153	Latin-2 (with Euro)	X	X	C	C	C	C	C	C	C	C
1154	Cyrillic (with Euro)	X	X	C	C	C	C	C	C	C	C
1155	Turkey Latin-5 (with Euro)	X	X	C	C	C	C	C	C	C	C
1156	Balitic (with Euro)	X	X	C	C	C	C	C	C	C	C
1157	Estonia (with Euro)	X	X	C	C	C	C	C	C	C	C
1158	Ukraine (with Euro)	X	X	C	C	C	C	C	C	C	C
4971	Greek (with Euro)	X	X								
8612	Arabic (Type 5)	X	X								
8616	Hebrew (Type 6)		X								
12708	Arabic (Type 7)		X								
62211	Hebrew (Type 5)		X	C	C	C	C	C	C	C	C
62224	Arabic (Type 6)		X	C	C	C	C	C	C	C	C
62229	Hebrew (Type 8)			C	C	C	C	C	C	C	C
62233	Arabic (Type 8)			C	C	C	C	C	C	C	C
62234	Arabic (Type 9)			C	C	C	C	C	C	C	C
62235	Hebrew (Type 6)		X	C	C	C	C	C	C	C	C
62240	Hebrew (Type 11)			C	C	C	C	C	C	C	C
62245	Hebrew (Type 10)		X	C	C	C	C	C	C	C	C
62250	Arabic (Type 12)			C	C	C	C	C	C	C	C
62251	Arabic (Type 6)			V8	V8	V8	V8	V8	V8	V8	V8

CCSID Values

Table 88. CCSIDs for EBCDIC Group 1a (Non-Latin-1 SBCS) Countries (continued)

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
String Types:											
4	Visual / Left-to-Right / Shaped / Symmetrical Swapping Off										
5	Implicit / Left-to-Right / Unshaped / Symmetrical Swapping On										
6	Implicit / Right-to-Left / Unshaped / Symmetrical Swapping On										
7	Visual / Contextual / Unshaped / Symmetrical Swapping Off										
8	Visual / Right-to-Left / Shaped / Symmetrical Swapping Off										
9	Visual / Right-to-Left / Shaped / Symmetrical Swapping On										
10	Implicit / Contextual-Left / Unshaped / Symmetrical Swapping On										
11	Implicit / Contextual-Right / Unshaped / Symmetrical Swapping On										
12	Implicit / Right-to-Left / Shaped / Symmetrical Swapping On										

Table 89. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
720	Arabic (MS-Dos)	X	C								
737	Greek (MS-Dos)	X	C	C	C	C	C	X	C	C	C
775	Baltic (MS-Dos)	X	C								
808	Cyrillic (with Euro)	X									
813	Greek/Latin (ISO 8859-7)	X	C	X	X	X	C	C	X	C	X
848	Ukraine (with Euro)	X									
849	Belarus (with Euro)	X									
851	Greek	X	C								
852	Latin-2 Multilingual	X	C	X	C	C	C	C	C	C	C
855	Cyrillic Multilingual	X	C	X	C	C	C	C	C	C	C
856	Arabic (Type 5)	X	C	C	X	C	C	C	C	C	C
857	Turkey Latin-5	X	C	X	C	C	C	C	C	C	C
862	Hebrew (Type 4)	X	C	X	C	C	C	C	C	C	C
864	Arabic (Type 5)	X	C	X	C	C	C	C	C	C	C
866	Cyrillic	X	C	X	C	C	C	C	C	C	C
867	Hebrew (with Euro)(Type 10)	X									
868	Urdu	X	C								
869	Greek	X	C	X	C	C	C	C	C	C	C
872	Cyrillic Multilingual (with Euro)	X									
878	Russian Internet	X	C								
901	Baltic 8-bit (with Euro)	X									
902	Estonia 8-bit (with Euro)	X									
912	Latin-2 (ISO 8859-2)	X	C	C	X	X	C	C	X	C	X
914	Latin-4 (ISO 8859-4)	X	C								
915	Cyrillic Multilingual (ISO 8859-5)	X	C	X	X	X	C	C	X	C	X
916	Hebrew/Latin (ISO 8859-8) (Type 5)	X	C	C	X	C	C	C	C	C	X
920	Turkey Latin-5 (ISO 8859-9)	X	C	C	X	X	C	C	X	C	X
921	Baltic 8-bit (ISO 8859-13)	X	C	X	X	C	C	C	C	C	C
922	Estonia 8-bit	X	C	X	X	C	C	C	C	C	C
1008	Arabic 8-bit ISO	X	C								
1046	Arabic (Type 5)	X	C	C	X	C	C	C	C	C	C
1089	Arabic (ISO 8859-6) (Type 5)	X	C	C	X	X	C	C	C	C	C
1098	Farsi	X	C								
1124	Ukraine 8-bit ISO	X	C	C	X	C	C	C	C	C	C
1125	Ukraine	X	C	X	C	C	C	C	C	C	C

CCSID Values

Table 89. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries (continued)

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
1131	Belarus	X	C	X	C	C	C	C	C	C	C
1250	Windows Latin-2	X	C	C	C	C	C	X	C	C	C
1251	Windows Cyrillic	X	C	C	C	C	C	X	C	C	C
1253	Windows Greek	X	C	C	C	C	C	X	C	C	C
1254	Windows Turkey	X	C	C	C	C	C	X	C	C	C
1255	Windows Hebrew (Type 5)	X	C	C	C	C	C	X	C	C	C
1256	Windows Arabic (Type 5)	X	C	C	C	C	C	X	C	C	C
1257	Windows Baltic	X	C	C	C	C	C	X	C	C	C
1280	Macintosh** Greek	X	C	C							
1281	Macintosh** Turkish	X	C								
1282	Macintosh** Latin-2	X	C								
1283	Macintosh** Cyrillic	X	C								
4909	ISO 8859-7 Greek/Latin (with Euro)	X									
4948	Latin-2 Multilingual	X	C								
4951	Cyrillic Multilingual	X	C								
4952	Hebrew	X	C								
4953	Turkey Latin-5	X	C								
4960	Arabic	X	C								
4965	Greek		C								
5346	Windows Latin-2 (with Euro)	X	C								
5347	Windows Cyrillic (with Euro)	X	C								
5349	Windows Greek (with Euro)	X	C								
5350	Windows Turkey (with Euro)	X	C								
5351	Windows Hebrew (with Euro)	X	C								
5352	Windows Arabic (with Euro)	X	C								
5353	Windows Baltic Rim (with Euro)	X	C								
5354	Windows Vietnam (with Euro)	T									
9044	Latin-2 Multilingual (with Euro)	T									
9048	Hebrew ASCII (with Euro) (Type 5)										
9049	Turkey Latin-5 (with Euro)	T									
9056	Arabic (Storage Interchange)	X	C								

Table 89. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries (continued)

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
9061	Greek (with Euro)	T									
9238	Arabic (with Euro) (Type 5)										
17248	Arabic (with Euro) (Type 5)										
62208	Hebrew (Type 4)			X	X	X	X	X	X	X	X
62209	Hebrew (Type 10)		C	X	C	C	C	C	C	C	C
62210	Hebrew/Latin (ISO 8859-8) (Type 4)		C	C	X	X	C	C	C	C	C
62213	Hebrew (Type 5)		C	X	C	C	C	C	C	C	C
62215	Windows Hebrew (Type 4)		C	C	C	C	C	X	C	C	C
62218	Arabic (Type 4)		C	X	C	C	C	C	C	C	C
62220	Hebrew (Type 6)			X	X	X	X	X	X	C	C
62221	Hebrew (Type 6)		C	X	C	C	C	C	C	C	C
62222	Hebrew/Latin (ISO 8859-8) (Type 6)		C	C	X	X	C	C	C	C	C
62223	Windows Hebrew (Type 6)		C	C	C	C	C	X	C	C	C
62225	Arabic (Type 6)			X	C	C	C	C	C	C	C
62226	Arabic (Type 6)			C	X	C	C	C	C	C	C
62227	Arabic (ISO 8859-6) (Type 6)			C	X	X	C	C	C	C	C
62228	Windows Arabic (Type 6)		C	C	C	C	C	X	C	C	C
62230	Hebrew (Type 8)			X	X	X	X	X	X	C	C
62231	Hebrew (Type 8)			X	C	C	C	C	C	C	C
62232	Hebrew/Latin (ISO 8859-8) (Type 8)			C	X	X	C	C	C	C	C
62236	Hebrew (Type 10)			X	X	X	X	X	X	X	X
62237	Hebrew (Type 8)										
62238	ISO 8859-8 Hebrew/Latin (Type 10)		C	C	C	C	C	X	C	C	C
62239	Windows Hebrew (Type 10)		C	C	C	C	C	X	C	C	C
62241	Hebrew (Type 11)			X	X	X	X	X	X	X	X
62242	Hebrew (Type 11)			X	C	C	C	C	C	C	C
62243	Hebrew/Latin (ISO 8859-8) (Type 11)			C	X	X	C	C	C	C	C
62244	Windows Hebrew (Type 11)			C	C	C	C	X	C	C	C
62246	Arabic (Type 8)										
62247	Arabic (Type 9)										
62248	Arabic (Type 4)										
62249	Arabic (Type 12)										

CCSID Values

Table 89. CCSIDs for PC-Data and ISO Group 1a (Non-Latin-1 SBCS) Countries (continued)

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
String Types:											
4	Visual / Left-to-Right / Shaped / Symmetrical Swapping Off										
5	Implicit / Left-to-Right / Unshaped / Symmetrical Swapping On										
6	Implicit / Right-to-Left / Unshaped / Symmetrical Swapping On										
7	Visual / Contextual / Unshaped / Symmetrical Swapping Off										
8	Visual / Right-to-Left / Shaped / Symmetrical Swapping Off										
9	Visual / Right-to-Left / Shaped / Symmetrical Swapping On										
10	Implicit / Contextual-Left / Unshaped / Symmetrical Swapping On										
11	Implicit / Contextual-Right / Unshaped / Symmetrical Swapping On										
12	Implicit / Right-to-Left / Shaped / Symmetrical Swapping On										

Table 90. SBCS CCSIDs for EBCDIC Group 2 (DBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
290	Japan Katakana (extended)	X	X	C	C	C	C	C	C	C	C
833	Korea (extended)	X	X	C	C	C	C	C	C	C	C
836	Simplified Chinese (extended)	X	X	C	C	C	C	C	C	C	C
838	Thailand (extended)	X	X	C	C	C	C	C	C	C	C
1027	Japan English (extended)	X	X	C	C	C	C	C	C	C	C
1130	Vietnam	X	X	C	C	C	C	C	C	C	C
1132	Lao	X	X								
1159	Traditional Chinese (extended with Euro)			C	C	C	C	C	C	C	C
1160	Thai (with Euro)	X	X	C	C	C	C	C	C	C	C
1164	Vietnam (with Euro)	X	X	C	C	C	C	C	C	C	C
5123	Japan (with Euro)	X	X								
8482	Japan Katakana (extended with Euro)	X		C	C	C	C	C	C	C	C
9030	Thailand (extended)	X	X								
13121	Korea Windows	X	X								
13124	Traditional Chinese	X	X								
28709	Traditional Chinese (extended)	X	X	C	C	C	C	C	C	C	C

CCSID Values

Table 91. SBCS CCSIDs for PC-Data Group 2 (DBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
367	Korea and Simplified Chinese EUC	X	C		X			C			
874	Thailand (extended)	X	C	X	X	X		X			
891	Korea (non-extended)	C	C								
895	Japan EUC - JISX201 Roman Set	C									
896	Japan EUC - JISX201 Katakana Set	C									
897	Japan (non-extended)	C	C								
903	Simplified Chinese (non-extended)	C	C								
904	Traditional Chinese (non-extended)	X	C								
1040	Korea (extended)	C	C								
1041	Japan (extended)	X	C								
1042	Simplified Chinese (extended)	C	C								
1043	Traditional Chinese (extended)	X	C								
1088	Korea (KS Code 5601-89)	X	C								
1114	Traditional Chinese (Big-5)	X	C								
1115	Simplified Chinese GB-Code	X	C								
1126	Korea Windows	X	C								
1129	Vietnam	X	C	X	X						
1133	Lao ISO	X	C								
1162	Thailand (extended) (180 char) (with Euro)	X									
1163	ISO Vietnam (with Euro)	X									
1258	Vietnam	X	C				X				
4970	Thailand (extended)	X	C								
5210	Traditional Chinese	X	C								
5222	Korea Windows (with Euro)										
9066	Thailand (extended)	X	C								

Table 92. DBCS CCSIDs for EBCDIC Group 2 (DBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
300	Japan - including 4370 user-defined characters (UDC)	X	X	C	C	C	C	C	C	C	C
834	Korea - including 1880 UDC	X	X	C	C	C	C	C	C	C	C
835	Traditional Chinese - including 6204 UDC	X	X	C	C	C	C	C	C	C	C
837	Simplified Chinese - including 1880 UDC	X	X	C	C	C	C	C	C	C	C
4396	Japan - including 1880 UDC	X	X	C	C	C	C	C	C	C	C
4930	Korea Windows	X	X	C	C	C	C	C	C	C	C
4933	Simplified Chinese	X	X	C	C	C	C	C	C	C	C
9027	Traditional Chinese (with Euro) - including 6204 UDC	C		C	C	C	C	C	C	C	C
16684	Japan (with Euro)	X	X	C	C	C	C	C	C	C	C

CCSID Values

Table 93. DBCS CCSIDs for PC-Data Group 2 (DBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
301	Japan - including 1880 UDC	X	C	X	X	C	C	C	C	C	C
926	Korea - including 1880 UDC	C	C								
927	Traditional Chinese - including 6204 UDC	X	C	X	C	C	C	C	C	C	C
928	Simplified Chinese - including 1880 UDC	C	C								
941	Japan Windows	X	C	C	C	C	C	X	C	C	C
947	Traditional Chinese (Big-5)	X	C	X	X	C	C	X	C	C	C
951	Korea (KS Code 5601-89) - including 1880 UDC	X	C	X	C	C	C	X	C	C	C
952	Japan (EUC) X208-1990 set	C									
953	Japan (EUC) X212-1990 set	C									
971	Korea (EUC) - including 188 UDC	X	C	C	X	X	X	C	C	C	C
1351	Japan HP-UX (J15)	X	C	C	C	X	C	C	C	C	C
1362	Korea Windows	X	C	C	C	C	C	X	C	C	C
1380	Simplified Chinese (GB-Code) - including 1880 UDC	X	C	X	C	C	C	X	X	C	C
1382	Simplified Chinese (EUC) - including 1360 UDC	X	C	C	X	X	X	C	X	C	C
1385	Traditional Chinese	X	C	C	C	C	C	X	C	C	C
21427	T-Chinese (Big-5) (6204 UDC)(with Euro)										

Table 94. Mixed CCSIDs for EBCDIC Group 2 (DBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
930	Japan Katakana/Kanji (extended) - including 4370 UDC	X	X	C	C	C	C	C	C	C	C
933	Korea (extended) - including 1880 UDC	X	X	C	C	C	C	C	C	C	C
935	Simplified Chinese (extended) - including 1880 UDC	X	X	C	C	C	C	C	C	C	C
937	Traditional Chinese (extended) - including 4370 UDC	X	X	C	C	C	C	C	C	C	C
939	Japan English/Kanji (extended) - including 4370 UDC	X	X	C	C	C	C	C	C	C	C
1364	Korea (extended)	X	X	C	C	C	C	C	C	C	C
1371	Traditional Chinese (extended with Euro) - including 4370 UDC			C	C	C	C	C	C	C	C
1388	Simplified Chinese	X	X	C	C	C	C	C	C	C	C
1390	Japan Katakana/Kanji (extended with Euro) - including 4370 UDC	X		C	C	C	C	C	C	C	C
1399	Japan (with Euro)	X	X							C	C
5026	Japan Katakana/Kanji (extended) - including 1880 UDC	X	X	C	C	C	C	C	C	C	C
5035	Japan English/Kanji (extended) - including 1880 UDC	X	X	C	C	C	C	C	C	C	C

CCSID Values

Table 95. Mixed CCSIDs for PC-Data Group 2 (DBCS) Countries

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
932	Japan (non-extended) - including 1880 UDC	X	C	X	X	C	C	C	C	C	C
934	Korea (non-extended) including 1880 UDC		C								
936	Simplified Chinese (non-extended) - including 1880 UDC		C								
938	Traditional Chinese (non-extended) - including 6204 UDC)	X	C	X	C	C	C	C	C	C	C
942	Japan (extended) - including 1880 UDC	X	C	X	C	C	C	C	C	C	C
943	Japan NT	X	C	X	C	C	X	X	C	C	C
944	Korea (extended) - including 1880 UDC		C								
946	Simplified Chinese (extended) - including 1880 UDC		C								
948	Traditional Chinese (extended) - including 6204 UDC	X	C	X	C	C	C	C	C	C	C
949	Korea (KS Code 5601-89) - including 1880 UDC	X	C	X	C	C	C	C	C	C	C
950	Traditional Chinese (Big-5)	X	C	X	X	X	X	X	C	C	X
954	Japan (EUC)		C	C	X	X	X	C	X	C	X
956	Japan 2022 TCP		C								
957	Japan 2022 TCP		C								
958	Japan 2022 TCP		C								
959	Japan 2022 TCP		C								
964	Traditional Chinese (EUC)		C	C	X	X	X	C	C	C	C
965	Traditional Chinese 2022 TCP		C								
970	Korea EUC	X	C	C	X	X	X	C	C	X	X
1363	Korea Windows	X	C	C	C	C	C	X	C	C	C
1381	Simplified Chinese GB-Code	X	C	X	C	C	C	X	C	C	C
1383	Simplified Chinese EUC	X	C	C	X	X	X	C	X	C	X
1386	Simplified Chinese	X	C	X	X	C	C	X	C	C	C
1392	Simplified Chinese GB18030 only supported in LOAD		C								
5039	Japan HP-UX (J15)	X		C	C	X	C	C	C	C	C
5050	Japan (EUC)		C								
5052	Japan 2022 TCP		C								

Table 95. Mixed CCSIDs for PC-Data Group 2 (DBCS) Countries (continued)

CCSID	Description	z/OS OS/390	iSeries	OS/2	AIX	HP	Sun	NT	SCO	SGI	Linux
5053	Japan 2022 TCP		C								
5054	Japan 2022 TCP		C								
5055	Japan 2022 TCP		C								
5307	Japan HP-UX (J15) HISTORICAL	X									
5488	Simplified Chinese PC Data (GB18030) — UWO only supporting for LOAD										
17354	Korea 2022 TCP		C								
25546	Korea 2022 TCP		C								
33722	Japan EUC		C								

CCSID Values

Appendix F. DB2 UDB for iSeries Catalog Views

The views contained in a DB2 UDB for iSeries catalog are described in this section. The database manager maintains a set of tables containing information about the data in each relational database. These tables are collectively known as the *catalog*. The *catalog tables* contain information about tables, user-defined functions, distinct types, parameters, procedures, packages, views, indexes, aliases, constraints, triggers, and languages supported by DB2 UDB for iSeries. The catalog also contains information about all relational databases that are accessible from this system.

There are three classes of catalog views:

- iSeries catalog tables and views

The iSeries catalog tables and views are modeled after the ANS and ISO catalog views, but are not identical to the ANS and ISO catalog views. These tables and views are compatible with prior releases of DB2 UDB for iSeries.

These tables and views exist in schemas QSYS and QSYS2.

The catalog tables and views contain information about all tables, parameters, procedures, functions, distinct types, packages, views, indexes, triggers, and constraints in the entire relational database. When an SQL schema is created, an additional set of these views (except SYSPARMS, SYSPROCS, SYSFUNCS, SYSROUTINES, SYSROUTINEDEP, and SYSTYPES) are created into the schema that only contain information about tables, packages, views, indexes, and constraints in that schema.

- ODBC and JDBC catalog views

The ODBC and JDBC catalog views are designed to satisfy ODBC and JDBC metadata API requests. For example, SQLColumns. These views are compatible with views on DB2 UDB for OS/390 and z/OS and DB2 UDB UWO Version 8. These views will be modified as ODBC or JDBC enhances or modifies their metadata APIs.

These views exist in schema SYSIBM.

- ANS and ISO catalog views

The ANS and ISO catalog views are designed to comply with the ANS and ISO SQL standard (the Information Schema catalog views). These views are compatible with views on DB2 UDB UWO Version 8. These views will be modified as the ANS and ISO standard is enhanced or modified.

These views exist in schema QSYS2 and SYSIBM.

There are several columns in these views that are reserved for future standard enhancements.

Note: Some of these views use special catalog functions as part of the view definition. These functions exist in SYSIBM, but should not be used directly in applications. The functions are created for specific independent auxiliary storage pools (IASP) and will likely change in future releases.

Notes

Names in the Catalog: In general, all names stored in columns of a catalog table are un delimited and case sensitive. For example, assume the following table was created:

```
CREATE TABLE "colname"/"long_table_name"
           ("long_column_name" CHAR(10),
           INTCOL INTEGER)
```

If the following select statement is used to return information about the mapping between SQL names and system names, the following select statement could be used:

```
SELECT TABLE_NAME, SYSTEM_TABLE_NAME, COLUMN_NAME, SYSTEM_COLUMN_NAME
FROM QSYS2/SYSCOLUMNS
WHERE TABLE_NAME = 'long_table_name' AND
      TABLE_SCHEMA = 'colname'
```

The following rows would be returned:

TABLE_NAME	SYSTEM_TABLE_NAME	COLUMN_NAME	SYSTEM_COLUMN_NAME
long_table_name	"long0001"	long_column_name	LONG_00001
long_table_name	"long0001"	INTCOL	INTCOL

System Names in the Catalog: In general, the longer SQL column names should be used rather than the short system column names. The short system column names for iSeries catalog tables and views are explicitly maintained for compatibility with prior releases and other DB2 UDB products. The short system column names for the ODBC and JDBC catalog views and the ANS and ISO catalog views are not explicitly maintained and may change between releases.

Null Values in the Catalog: If the information in a column is not applicable, the null value is returned. Using the table created above, the following select statement, which queries the NUMERIC_SCALE and the CHARACTER_MAXIMUM_LENGTH, would return the null value when the data was not applicable to the data type of the column.

```
SELECT COLUMN_NAME, NUMERIC_SCALE, CHARACTER_MAXIMUM_LENGTH
FROM QSYS2/SYSCOLUMNS
WHERE TABLE_NAME = 'long_table_name' AND
      TABLE_SCHEMA = 'colname'
```

The following rows would be returned:

COLUMN_NAME	NUMERIC_SCALE	CHARACTER_MAXIMUM_LENGTH
long_column_name	?	10
INTCOL	0	?

Because numeric scale is not valid for a character column, the null value is returned for NUMERIC_SCALE for the "long_column_name" column. Because character length is not valid for a numeric column, the null value is returned for CHARACTER_MAXIMUM_LENGTH for the INTCOL column.

Install and Backup Considerations: Certain catalog tables and any views created over the catalog tables and views should be regularly saved:

- The catalog table QSYS.QADBXRDBD contains relational database information. This table should be regularly saved.
- When an ILE external function or procedure or an SQL function or procedure is restored, information is automatically inserted into these catalog tables. This does not occur for non-ILE external functions and procedures. In order to back up the definitions of non-ILE external functions or procedures, ensure that the catalog tables SYSROUTINES and SYSPARMS are saved or ensure you have a back up of the SQL source statements that were used to create these functions and procedures.
- All catalog views in the QSYS2 or SYSIBM schemas are system objects. This means that any user views created over these catalog views must be deleted when the operating system is installed. All dependent objects must be deleted as well. To avoid this requirement, you can save views before installation and then restore them afterwards.
- Catalog tables in the QSYS library are also system objects. However, the catalog tables in the QSYS library are not deleted during installation. Hence, any views created over these tables are preserved throughout the installation process.

Granting Privileges to Catalog Views: Tables and views in the catalog are like any other database tables and views. If you have authorization, you can use SQL statements to look at data in the catalog views in the same way that you retrieve data from any other table. The tables and views in the catalogs are shipped with the SELECT privilege to PUBLIC. This privilege may be revoked and the SELECT privilege granted to individual users.

QSYS Catalog Tables: Most of the catalog views are based on the following tables in the QSYS library (sometimes called the database cross reference files). These tables are not shipped with the SELECT privilege to PUBLIC and should not be used directly:

QADBCCST	QADBKFLD	QADBXSFLD
QADBFDEP	QADBPKG	QADBXRIGB
QADBFCST	QADBXRDBD	QADBXRIGC
QADBIFLD	QADBXREF	QADBXRIGD

iSeries Catalog Tables and Views

The iSeries catalog includes the following views and tables in the QSYS2 schema:

DB2 UDB for iSeries name	Corresponding ANSI/ISO name	Description
"SYSCATALOGS" on page 885	CATALOGS	Information about relational databases
"SYSCHKCST" on page 887	CHECK_CONSTRAINTS	Information about check constraints
"SYSCOLUMNS" on page 888	COLUMNS	Information about column attributes
"SYSCST" on page 897	TABLE_CONSTRAINTS	Information about all constraints
"SYSCSTCOL" on page 898	CONSTRAINT_COLUMN_USAGE	Information about the columns referenced in a constraint
"SYSCSTDEP" on page 899	CONSTRAINT_TABLE_USAGE	Information about constraint dependencies on tables
"SYSFUNCS" on page 900	ROUTINES	Information about user-defined functions
"SYSINDEXES" on page 906		Information about indexes
"SYSJARCONTENTS" on page 907		Information about jars for Java routines.
"SYSJAROBJECTS" on page 908		Information about jars for Java routines.
"SYSKEYCST" on page 909	KEY_COLUMN_USAGE	Information about unique, primary, and foreign keys
"SYSKEYS" on page 910		Information about index keys
"SYSPACKAGE" on page 911		Information about packages
"SYSPARMS" on page 913	PARAMETERS	Information about routine parameters
"SYSPROCS" on page 917	ROUTINES	Information about procedures
"SYSREFCST" on page 922	REFERENTIAL_CONSTRAINTS	Information about referential constraints
"SYSROUTINES" on page 924	ROUTINES	Information about functions and procedures
"SYSROUTINEDEP" on page 923	ROUTINE_TABLE_USAGE	Information about function and procedure dependencies
"SYSTABLES" on page 932	TABLES	Information about tables and views
"SYSTRIGCOL" on page 934	TRIGGER_COLUMN_USAGE	Information about columns used in a trigger
"SYSTRIGDEP" on page 935	TRIGGER_TABLE_USAGE	Information about objects used in a trigger
"SYSTRIGGERS" on page 936	TRIGGERS	Information about triggers
"SYSTRIGUPD" on page 940	TRIGGERED_UPDATE_COLUMNS	Information about columns in the WHEN clause of a trigger
"SYSTYPES" on page 941	USER_DEFINED_TYPES	Information about built-in data types and distinct types
"SYSVIEWDEP" on page 946	VIEW_TABLE_USAGE	Information about view dependencies on tables
"SYSVIEWS" on page 948	VIEWS	Information about definition of a view

SYSCATALOGS

The SYSCATALOGS view contains one row for each relational database that a user can connect to. The following table describes the columns in the SYSCATALOGS view.

Table 96. SYSCATALOGS view

Column Name	System Column Name	Data Type	Description
CATALOG_NAME	LOCATION	VARCHAR(18)	Relational database name.
CATALOG_STATUS	RDBASPSTAT	CHAR(10)	Status of a relational database. ACTIVE The relational database is associated with an independent auxiliary storage pool (IASP) that is active, but not yet available. AVAILABLE The relational database is available. VARYOFF The relational database is associated with an independent auxiliary storage pool (IASP) that is varied off. VARYON The relational database is associated with an independent auxiliary storage pool (IASP) that is varied on, but not yet available. UNKNOWN The status of the relational database is unknown. The status of remote relational databases is always unknown.
CATALOG_TYPE	RDBTYPE	CHAR(7)	Relational database type. LOCAL The relational database is local to this system. REMOTE The relational database is on a remote system.
CATALOG_ASPGRP	RDBASPGRP	VARCHAR(10) Nullable	Independent auxiliary storage pool (IASP) name. Contains the null value if the relational database status is UNKNOWN.
CATALOG_ASPNUM	RDBASPNUM	VARCHAR(10) Nullable	Independent auxiliary storage pool (IASP) number. Contains the null value if the relational database status is UNKNOWN.

SYSCATALOGS

| *Table 96. SYSCATALOGS view (continued)*

		System Column Name	Data Type	Description
	Column Name			
	CATALOG_TEXT	RDBTEXT	CHAR(50)	Relational database text description.

SYSCHKCST

The SYSCHKCST view contains one row for each check constraint in the SQL schema. The following table describes the columns in the SYSCHKCST view.

Table 97. SYSCHKCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	DBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint
CHECK_CLAUSE	CHECK	VARCHAR(2000) Nullable	Text of the check constraint clause
			Contains the null value if the check clause cannot be expressed without truncation.

SYSCOLUMNS

SYSCOLUMNS

The SYSCOLUMNS view contains one row for every column of each table and view in the SQL schema (including the columns of the SQL catalog). The following table describes the columns in the SYSCOLUMNS view:

Table 98. SYSCOLUMNS view

Column name	System Column Name	Data Type	Description
COLUMN_NAME	NAME	VARCHAR(128)	Name of the column. This will be the SQL column name if one exists; otherwise, it will be the system column name.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table or view that contains the column. This will be the SQL table or view name if one exists; otherwise, it will be the system table or view name.
TABLE_OWNER	TBCREATOR	VARCHAR(128)	The owner of the table or view.
ORDINAL_POSITION	COLNO	INTEGER	Numeric place of the column in the table or view, ordered from left to right.

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description	
DATA_TYPE	COLTYPE	VARCHAR(8)	Type of column:	
			BIGINT	Big number
			INTEGER	Large number
			SMALLINT	Small number
			DECIMAL	Packed decimal
			NUMERIC	Zoned decimal
			FLOAT	Floating point; FLOAT, REAL, or DOUBLE PRECISION
			CHAR	Fixed-length character string
			VARCHAR	Varying-length character string
			CLOB	Character large object string
			GRAPHIC	Fixed-length graphic string
			VARG	Varying-length graphic string
			DBCLOB	Double-byte character large object string
			BLOB	Binary large object string
			DATE	Date
			TIME	Time
			TIMESTAMP	Timestamp
DATALINK	Datalink			
ROWID	Row ID			
DISTINCT	Distinct type			

SYSCOLUMNS

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
LENGTH	LENGTH	INTEGER	<p>The length attribute of the column; or, in the case of a decimal, numeric, or nonzero precision binary column, its precision:</p> <p>8 bytes BIGINT</p> <p>4 bytes INTEGER</p> <p>2 bytes SMALLINT</p> <p>Precision of number DECIMAL</p> <p>Precision of number NUMERIC</p> <p>8 bytes FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION</p> <p>4 bytes FLOAT(n) where n = 1 to 24, or REAL</p> <p>Length of string CHAR</p> <p>Maximum length of string VARCHAR or CLOB</p> <p>Length of graphic string GRAPHIC</p> <p>Maximum length of graphic string VARGRAPHIC or DBCLOB</p> <p>Maximum length of binary string BLOB</p> <p>4 bytes DATE</p> <p>3 bytes TIME</p> <p>10 bytes TIMESTAMP</p> <p>Maximum length of datalink URL and comment DATALINK</p> <p>40 bytes ROWID</p> <p>Same value as the source type DISTINCT</p>
NUMERIC_SCALE	SCALE	INTEGER Nullable	<p>Scale of numeric data.</p> <p>Contains the null value if the column is not decimal, numeric, or binary.</p>
IS_NULLABLE	NULLS	CHAR(1)	<p>If the column can contain null values:</p> <p>N No</p> <p>Y Yes</p>

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
IS_UPDATABLE	UPDATES	CHAR(1)	If the column can be updated: N No Y Yes
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
HAS_DEFAULT	DEFAULT	CHAR(1)	If the column has a default value (DEFAULT clause or null capable): N No Y Yes A The column has a ROWID data type and the GENERATED ALWAYS attribute. D The column has a ROWID data type and the GENERATED BY DEFAULT attribute. I The column is defined with the AS IDENTITY and GENERATED ALWAYS attributes. J The column is defined with the AS IDENTITY and GENERATED BY DEFAULT attributes.
COLUMN_HEADING	LABEL	VARCHAR(60) Nullable	A character string supplied with the LABEL statement (column headings) Contains the null value if there is no column heading.

SYSCOLUMNS

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
STORAGE	STORAGE	INTEGER	<p>The storage requirements for the column:</p> <p>8 bytes BIGINT</p> <p>4 bytes INTEGER</p> <p>2 bytes SMALLINT</p> <p>(Precision/2) + 1 DECIMAL</p> <p>Precision of number NUMERIC</p> <p>8 bytes FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION</p> <p>4 bytes FLOAT(n) where n = 1 to 24, or REAL</p> <p>Length of string CHAR</p> <p>Maximum length of string + 2 VARCHAR</p> <p>Maximum length of string + 29 CLOB</p> <p>Length of string * 2 GRAPHIC</p> <p>Maximum length of string * 2 + 2 VARGRAPHIC</p> <p>Maximum length of string * 2 + 29 DBCLOB</p> <p>4 bytes DATE</p> <p>3 bytes TIME</p> <p>10 bytes TIMESTAMP</p> <p>Maximum length of datalink URL and comment + 24 DATALINK</p> <p>42 bytes ROWID</p> <p>Same value as the source type DISTINCT</p> <p>Note: This column supplies the storage requirements for all data types.</p>

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	<p>The precision of all numeric columns.</p> <p>Note: This column supplies the precision of all numeric data types, including single- and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits.</p> <p>Contains the null value if the column is not numeric.</p>
CCSID	CCSID	INTEGER Nullable	<p>The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB, and DATALINK columns.</p> <p>Contains 65535 if the column is a BLOB or ROWID.</p> <p>Contains the null value if the column is a numeric data type.</p>
TABLE_SCHEMA	DBNAME	VARCHAR(128)	The name of the SQL schema containing the table or view.
COLUMN_DEFAULT	DFTVALUE	VARCHAR(2000) Nullable	<p>The default value of a column, if one exists. If the default value of the column cannot be represented without truncation, then the value of the column is the string 'TRUNCATED'. The default value is stored in character form. The following special values also exist:</p> <p>CURRENT_DATE The default value is the current date.</p> <p>CURRENT_TIME The default value is the current time.</p> <p>CURRENT_TIMESTAMP The default value is the current timestamp.</p> <p>NULL The default value is the null value.</p> <p>USER The default value is the current job user.</p> <p>Contains the null value if the column has no default value. For example, if the column has an IDENTITY attribute or is a row ID.</p>

SYSCOLUMNS

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	Maximum length of the string for binary, character and graphic string data types. Contains the null value if the column is not a string.
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	Number of bytes for binary, character and graphic string data types. Contains the null value if the column is not a string.
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits 2 Binary; floating-point precision is specified in binary digits. 10 Decimal; all other numeric types are specified in decimal digits. Contains the null value if the column is not numeric.
DATETIME_PRECISION	DATPRC	INTEGER Nullable	The fractional part of a date, time, or timestamp. 0 For DATE and TIME data types 6 For TIMESTAMP data types (number of microseconds). Contains the null value if the column is not a date, time, or timestamp.
COLUMN_TEXT	LABELTEXT	VARCHAR(50) Nullable	A character string supplied with the LABEL statement (column text) Contains the null value if the column has no column text.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	The system name of the column
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	The system name of the table or view
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	The system name of the schema
USER_DEFINED_TYPE_SCHEMA	TYPESHEMA	VARCHAR(128) Nullable	The name of the schema if this is a the distinct type. Contains the null value if the column is not a distinct type.
USER_DEFINED_TYPE_NAME	TYPENAME	VARCHAR(128) Nullable	The name of the distinct type. Contains the null value if the column is not a distinct type.

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
IS_IDENTITY	IDENTITY	VARCHAR(3)	This column identifies whether the column is an identity column. NO The column is not an identity column. YES The column is an identity column.
IDENTITY_GENERATION	GENERATED	VARCHAR(10) Nullable	This column identifies whether the column is GENERATED ALWAYS or GENERATED BY DEFAULT. ALWAYS The column value is always generated. BY DEFAULT The column value is generated by default. Contains the null value if the column is not a ROWID or IDENTITY column.
IDENTITY_START	START	DECIMAL(31,0) Nullable	Starting value of the identity column. Contains the null value if the column is not an IDENTITY column.
IDENTITY_INCREMENT	INCREMENT	DECIMAL(31,0) Nullable	Increment value of the identity column. Contains the null value if the column is not an IDENTITY column.
IDENTITY_MINIMUM	MINVALUE	DECIMAL(31,0) Nullable	Minimum value of the identity column. Contains the null value if the column is not an IDENTITY column.
IDENTITY_MAXIMUM	MAXVALUE	DECIMAL(31,0) Nullable	Maximum value of the identity column. Contains the null value if the column is not an IDENTITY column.
IDENTITY_CYCLE	CYCLE	VARCHAR(3) Nullable	This column identifies whether the identity column values will continue to be generated after the minimum or maximum value has been reached. NO Values will not continue to be generated. YES Values will continue to be generated. Contains the null value if the column is not an IDENTITY column.

SYSCOLUMNS

Table 98. SYSCOLUMNS view (continued)

Column name	System Column Name	Data Type	Description
IDENTITY_CACHE	CACHE	INTEGER Nullable	Specifies the number of identity values that may be preallocate for faster access. Zero indicates that the values will not be preallocated.
			Contains the null value if the column is not an IDENTITY column.
IDENTITY_ORDER	ORDER	VARCHAR(3) Nullable	Specifies whether the identity values must be generated in order of the request.
			NO Values do not need to be generated in order of the request.
			YES Values must be generated in order of the request.
			Contains the null value if the column is not an IDENTITY column.

SYSCST

The SYSCST view contains one row for each constraint in the SQL schema. The following table describes the columns in the SYSCST view:

Table 99. SYSCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
CONSTRAINT_TYPE	TYPE	VARCHAR(11)	Constraint Type CHECK UNIQUE PRIMARY KEY FOREIGN KEY
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table which the constraint is created over. This will be the SQL table name if it exists; otherwise, it will be the system table name.
IS_DEFERRABLE	ISDEFER	VARCHAR(3)	Indicates whether the constraint checking can be deferred. Will always be 'NO'.
INITIALLY_DEFERRED	INITDEFER	VARCHAR(3)	Indicates whether the constraint was defined as initially deferred. Will always be 'NO'.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the table.
CONSTRAINT_KEYS	COLCOUNT	SMALLINT Nullable	Specifies the number of key columns if this is a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. Contains the null value if the constraint is a CHECK constraint.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSCSTCOL

SYSCSTCOL

The SYSCSTCOL view records the columns on which constraints are defined. There is one row for every column in a unique or primary key constraint and the referencing columns of a referential constraint. The following table describes the columns in the SYSCSTCOL view:

Table 100. SYSCSTCOL view

Column Name	System Column Name	Data Type	Description
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the SQL schema that contains the table the constraint is dependent on.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table the constraint is dependent on. This is the SQL table name if it exists; otherwise, it is the system table name.
COLUMN_NAME	COLUMN	VARCHAR(128)	Column that the constraint was created over. This is the SQL column name if it exists; otherwise, it is the system column name.
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema of the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	System name of the column.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the table.

SYSCSTDEP

The SYSCSTDEP view records the tables on which constraints are defined. The following table describes the columns in the SYSCSTDEP view:

Table 101. SYSCSTDEP view

Column Name	System Column Name	Data Type	Description
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the SQL schema that contains the table on which the constraint is dependent
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table on which the constraint is dependent. This is the SQL table name if it exists otherwise it is the system table name.
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema of the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the table.
I IASP_NUMBER	I IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSFUNCS

SYSFUNCS

The SYSFUNCS view contains one row for each function created by the CREATE FUNCTION statement. The following table describes the columns in the SYSFUNCS view:

Table 102. SYSFUNCS view

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSHEMA	VARCHAR(128)	Schema name of the routine (function) instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ROUTINE_SCHEMA	FUNCSHEMA	VARCHAR(128)	Name of the SQL schema (schema) that contains the routine.
ROUTINE_NAME	FUNCNAME	VARCHAR(128)	Name of the routine.
ROUTINE_CREATED	FUNCCREATE	TIMESTAMP	Identifies the timestamp when the routine was created.
ROUTINE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the routine.
ROUTINE_BODY	BODY	VARCHAR(8)	The type of the routine body: EXTERNAL This is an external routine. SQL This is an SQL routine.
EXTERNAL_NAME	EXTNAME	VARCHAR(279) Nullable	If this is an external routine, this column identifies the external program name. <ul style="list-style-type: none">• For ILE service programs, the external program name is <i>schema-name/service-program-name(entry-point-name)</i>.• For Java programs, the external program name is an optional jar-id followed by a <i>fully-qualified-class-name!method-name</i> or <i>fully-qualified-class-name.method-name</i>.• For all other languages, the external program name is <i>schema-name/program-name</i>. Contains the null value if this is not an external routine.

Table 102. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
EXTERNAL_LANGUAGE	LANGUAGE	VARCHAR(8) Nullable	<p>If this is an external routine, this column identifies the external program name.</p> <p>C The external program is written in C.</p> <p>C++ The external program is written in C++.</p> <p>CL The external program is written in CL.</p> <p>COBOL The external program is written in COBOL.</p> <p>COBOLLE The external program is written in ILE COBOL.</p> <p>JAVA The external program is written in JAVA.</p> <p>PLI The external program is written in PL/I.</p> <p>RPG The external program is written in RPG.</p> <p>RPGLE The external program is written in ILE RPG.</p> <p>Contains the null value if this is not an external routine.</p>
PARAMETER_STYLE	PARAM_STYLE	VARCHAR(7) Nullable	<p>If this is an external routine, this column identifies the parameter style (calling convention).</p> <p>DB2SQL This is the DB2SQL calling convention.</p> <p>DB2GNRL This is the DB2GENERAL calling convention.</p> <p>GENERAL This is the GENERAL calling convention.</p> <p>JAVA This is the JAVA calling convention.</p> <p>NULLS This is the GENERAL WITH NULLS calling convention.</p> <p>SQL This is the SQL standard calling convention.</p> <p>Contains the null value if this is not an external routine.</p>

SYSFUNCS

Table 102. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
IS_DETERMINISTIC	DETERMINE	VARCHAR(3)	<p>This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result.</p> <p>NO The routine is not deterministic.</p> <p>YES The routine is deterministic.</p>
SQL_DATA_ACCESS	DATAACCESS	VARCHAR(8)	<p>This column identifies whether a routine contains SQL and whether it reads or modifies data.</p> <p>NONE The routine does not contain any SQL statements.</p> <p>CONTAINS The routine contains SQL statements.</p> <p>READS The routine possibly reads data from a table or view.</p> <p>MODIFIES The routine possibly modifies data in a table or view or issues SQL DDL statements.</p>
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	<p>If this is an SQL routine, this column identifies the path.</p> <p>Contains the null value if this is an external routine.</p>
PARAM_SIGNATURE	SIGNATURE	VARCHAR(510)	This column identifies the routine signature.
NUMBER_OF_RESULTS	NUMRESULTS	SMALLINT	Identifies the number of results.
IN_PARM	IN_PARM	SMALLINT	Identifies the number of input parameters. 0 indicates that there are no input parameters.
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	<p>A character string supplied with the COMMENT statement.</p> <p>Contains the null value if there is no long comment.</p>
ROUTINE_DEFINITION	ROUTINEDEF	VARCHAR(24000) Nullable	<p>If this is an SQL routine, this column contains the SQL routine body.</p> <p>Contains the null value if this is not an SQL routine or if the routine body cannot be contained in this column without truncation.</p>

Table 102. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
FUNCTION_ORIGIN	ORIGIN	CHAR(1)	Identifies the type of function. If this is a procedure, this column contains a blank. B This is a built-in function (defined by DB2 UDB for iSeries). E This is a user-defined function. U This is a user-defined function that is based on another function. S This is a system-generated function.
FUNCTION_TYPE	TYPE	CHAR(1)	Identifies the form of the function. If this is a procedure, this column contains a blank. S This is a scalar function. C This is a column function. T This is a table function.
EXTERNAL_ACTION	EXTACTION	CHAR(1) Nullable	Identifies the whether the invocation of the function has external effects. E This function has external side effects. N This function does not have any external side effects.
IS_NULL_CALL	NULL_CALL	VARCHAR(3) Nullable	Identifies whether the function needs to be called if an input parameter is the null value. NO This function need not be called if an input parameter is the null value. If this is a scalar function, the result of the function is implicitly null if any of the operands are null. If this is a table function, the result of the function is an empty table if any of the operands are the null value. YES This function must be called even if an input operand is null.

SYSFUNCS

Table 102. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
SCRATCH_PAD	SCRATCHPAD	INTEGER Nullable	<p>Identifies whether the address of a static memory area (scratch pad) is passed to the function.</p> <p>0 The function does not have a scratch pad.</p> <p>integer Indicates the size of the scratch pad passed to the function.</p>
FINAL_CALL	FINAL_CALL	VARCHAR(3) Nullable	<p>Indicates whether a final call to the function should be made to allow the function to clean up its work areas (scratch pads).</p> <p>NO No final call is made.</p> <p>YES A final call to the function is made when the statement is complete.</p>
PARALLELIZABLE	PARALLEL	VARCHAR(3) Nullable	<p>Identifies whether the function can be run in parallel.</p> <p>NO The function must be synchronous.</p> <p>YES The function can be run in parallel.</p>
DBINFO	DBINFO	VARCHAR(3) Nullable	<p>Identifies whether information about the database is passed to the function.</p> <p>NO No database information is passed to the function.</p> <p>YES Information about the database is passed to the function.</p>
SOURCE_SPECIFIC_SCHEMA	SRCSCHEMA	VARCHAR(128) Nullable	<p>If this is sourced function and the source is user-defined, this column contains the name of the source schema. If this is a sourced function and the source is built-in, this column contains 'QSYS2'.</p> <p>Contains the null value if this is not a sourced function.</p>
SOURCE_SPECIFIC_NAME	SRCNAME	VARCHAR(128) Nullable	<p>If this is sourced function and the source is user-defined, this column contains the specific name of the source function name.</p> <p>Contains the null value if this is not a sourced function.</p>

Table 102. SYSFUNCS view (continued)

Column Name	System Column Name	Data Type	Description
IS_USER_DEFINED_CAST	CAST_FUNC	VARCHAR(3) Nullable	Identifies whether this function is a cast function created when a distinct type was created. NO This function is not a cast function. YES This function is a cast function.
CARDINALITY	CARD	BIGINT Nullable	Specifies the cardinality for a table function. Contains the null value if the function is not a table function or if cardinality was not specified.
FENCED	FENCED	VARCHAR(3) Nullable	Identifies whether the function is fenced. NO The function is not fenced. YES The function is fenced.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSINDEXES

SYSINDEXES

The SYSINDEXES view contains one row for every index in the SQL schema created using the SQL CREATE INDEX statement, including indexes on the SQL catalog. The following table describes the columns in the SYSINDEXES view:

Table 103. SYSINDEXES view

Column Name	System Column Name	Data Type	Description
INDEX_NAME	NAME	VARCHAR(128)	Name of the index. This will be the SQL index name if one exists; otherwise, it will be the system index name.
INDEX_OWNER	CREATOR	VARCHAR(128)	Owner of the index
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table on which the index is defined. This will be the SQL table name if one exists; otherwise, it will be the system table name.
TABLE_OWNER	TBCREATOR	VARCHAR(128)	Owner of the table
TABLE_SCHEMA	TBDBNAME	VARCHAR(128)	Name of the SQL schema that contains the table on which the index is defined
IS_UNIQUE	UNIQUERULE	CHAR(1)	If the index is unique: D No (duplicates are allowed) V Yes (duplicate NULL values are allowed) U Yes E Encoded vector index
COLUMN_COUNT	COLCOUNT	INTEGER	Number of columns in the key
INDEX_SCHEMA	DBNAME	VARCHAR(128)	Name of the SQL schema that contains the index
SYSTEM_INDEX_NAME	SYS_IXNAME	CHAR(10)	System index name
SYSTEM_INDEX_SCHEMA	SYS_IDNAME	CHAR(10)	System index schema name
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System table schema name
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSJARCONTENTS

The SYSJARCONTENTS table contains one row for each class defined by a jarid in the SQL schema. The following table describes the columns in the SYSJARCONTENTS view.

Table 104. SYSJARCONTENTS view

Column Name	System Column Name	Data Type	Description
JARSHEMA	JARSHEMA	VARCHAR(128)	Name of the schema containing the jar_id.
JAR_ID	JAR_ID	VARCHAR(128)	Name of the jar_id.
CLASS	CLASS	VARCHAR(128)	Name of the class.
CLASS_SOURCE	CLASSSRC	DBCLOB(10485760) Nullable	Reserved. Contains the null value.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSJAROBJECTS

SYSJAROBJECTS

The SYSJAROBJECTS table contains one row for each jarid in the SQL schema. The following table describes the columns in the SYSJAROBJECTS view.

Table 105. SYSJAROBJECTS view

Column Name	System Column Name	Data Type	Description
JARSCHEMA	JARSCHEMA	VARCHAR(128)	Name of the schema containing the jar_id.
JAR_ID	JAR_ID	VARCHAR(128)	Name of the jar_id.
DEFINER	DEFINER	VARCHAR(128)	Name of the owner of the jarid.
JAR_DATA	JAR_DATA	BLOB(104857600) Nullable	Byte-codes for the jar.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
JAR_CREATED	CREATEDTS	TIMESTAMP	Jar created timestamp
LAST_ALTERED	ALTEREDTS	TIMESTAMP Nullable	Reserved. Contains the null value.
DEBUG_MODE	DEBUG_MODE	CHAR(1)	Identifies whether the function is debuggable. 0 The function is not debuggable. 2 The function is debuggable.
DEBUG_DATA	DEBUG_DATA	CLOB(1048576) Nullable	Reserved. Contains the null value.

SYSKEYCST

The SYSKEYCST view contains one or more rows for each UNIQUE KEY, PRIMARY KEY, or FOREIGN KEY in the SQL schema. There is one row for each column in every unique or primary key constraint and the referencing columns of a referential constraint. The following table describes the columns in the SYSKEYCST view:

Table 106. SYSKEYCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
TABLE_SCHEMA	TDBNAME	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	TBNAME	VARCHAR(128)	Name of the table.
COLUMN_NAME	COLNAME	VARCHAR(128)	Name of the column.
ORDINAL_POSITION	COLSEQ	INTEGER	The position of the column within the key
COLUMN_POSITION	COLNO	INTEGER	The position of the column within the row
TABLE_OWNER	CREATOR	VARCHAR(128)	Owner of the table.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	System name of the column.
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System name of the table.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the schema table.

SYSKEYS

SYSKEYS

The SYSKEYS view contains one row for every column of an index in the SQL schema, including the keys for the indexes on the SQL catalog. The following table describes the columns in the SYSKEYS view:

Table 107. SYSKEYS view

Column Name	System Column Name	Data Type	Description
INDEX_NAME	IXNAME	VARCHAR(128)	Name of the index. This will be the SQL index name if one exists; otherwise, it will be the system index name.
INDEX_OWNER	IXCREATOR	VARCHAR(128)	Owner of the index
COLUMN_NAME	COLNAME	VARCHAR(128)	Name of the column of the key. This will be the SQL column name if one exists; otherwise, it will be the system column name.
COLUMN_POSITION	COLNO	INTEGER	Numeric position of the column in the row
ORDINAL_POSITION	COLSEQ	INTEGER	Numeric position of the column in the key
ORDERING	ORDERING	CHAR(1)	Order of the column in the key: A Ascending D Descending
INDEX_SCHEMA	IXDBNAME	VARCHAR(128)	Name of the schema containing the index.
SYSTEM_COLUMN_NAME	SYS_CNAME	CHAR(10)	System name of the column
SYSTEM_INDEX_NAME	SYS_IXNAME	CHAR(10)	System name of the index
SYSTEM_INDEX_SCHEMA	SYS_IDNAME	CHAR(10)	System name of the schema containing the index

SYSPACKAGE

The SYSPACKAGE view contains one row for each SQL package in the SQL schema. The following table describes the columns in the SYSPACKAGE view:

Table 108. SYSPACKAGE view

Column Name	System Column Name	Data Type	Description
PACKAGE_CATALOG	LOCATION	VARCHAR(128)	Relational database name (RDBNAME) of the SQL package
PACKAGE_SCHEMA	COLLID	VARCHAR(128)	Name of the schema
PACKAGE_NAME	NAME	VARCHAR(128)	Name of the SQL package
PACKAGE_OWNER	OWNER	VARCHAR(128)	Owner of the SQL package
PACKAGE_CREATOR	CREATOR	VARCHAR(128)	Creator of the SQL package
CREATION_TIMESTAMP	TIMESTAMP	CHAR(26)	Timestamp of when the SQL package was created
DEFAULT_SCHEMA	QUALIFIER	VARCHAR(128)	Implicit name for unqualified tables, views, and indexes
PROGRAM_NAME	PROGNAME	VARCHAR(128)	Name of program the package was created from
PROGRAM_SCHEMA	LIBRARY	VARCHAR(128)	Name of schema containing the program
PROGRAM_CATALOG	RDB	VARCHAR(128)	Name of the relational database where the program resides
ISOLATION	ISOLATION	CHAR(2)	Isolation option specification: RR Repeatable Read RS Read Stability (*ALL) CS Cursor Stability (*CS) UR Uncommitted Read (*CHG) NO None (*NONE)
QUOTE	QUOTE	CHAR(1)	Escape character specification (Y/N): Y = Quotation mark N = Apostrophe
COMMA	COMMA	CHAR(1)	Comma option specification (Y/N): Y = Comma N = Period
PACKAGE_TEXT	LABEL	VARCHAR(50)	A character string you supply with the LABEL statement.
LONG_COMMENT	REMARKS	VARCHAR(2000)	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
CONSISTENCY_TOKEN	CONTOKEN	CHAR(8) FOR BIT DATA	Consistency token of package
SYSTEM_PACKAGE_NAME	SYS_NAME	CHAR(10)	System name of the package.
SYSTEM_PACKAGE_SCHEMA	SYS_DNAME	CHAR(10)	System name of the schema containing the package.
SYSTEM_DEFAULT_SCHEMA	SYS_DDNAME	CHAR(10)	System name of the implicit qualifier for unqualified table, views, indexes, and packages.

SYSPACKAGE

Table 108. SYSPACKAGE view (continued)

Column Name	System Column Name	Data Type	Description
SYSTEM_PROGRAM_NAME	SYS_PNAME	CHAR(10)	System name of the program.
SYSTEM_PROGRAM_SCHEMA	SYS_PDNAME	CHAR(10)	System name of the schema containing the program
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

SYSPARMS

The SYSPARMS table contains one row for each parameter of a procedure created by the CREATE PROCEDURE statement or function created by the CREATE FUNCTION statement. The following table describes the columns in the SYSPARMS table:

Table 109. SYSPARMS table

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ORDINAL_POSITION	PARMNO	INTEGER	Numeric place of the parameter in the parameter list, ordered from left to right.
PARAMETER_MODE	PARMMODE	VARCHAR(5)	Type of the parameter: IN This is an input parameter. OUT This is an output parameter. INOUT This is an input/output parameter.
PARAMETER_NAME	PARMNAME	VARCHAR(128) Nullable	Name of the parameter. Contains the null value if the parameter does not have a name.

SYSPARMS

Table 109. SYSPARMS table (continued)

Column Name	System Column Name	Data Type	Description
DATA_TYPE	DATA_TYPE	VARCHAR(128)	Type of column: BIGINT Big number INTEGER Large number SMALLINT Small number DECIMAL Packed decimal NUMERIC Zoned decimal DOUBLE PRECISION Floating point; DOUBLE PRECISION REAL Floating point; REAL CHARACTER Fixed-length character string CHARACTER VARYING Varying-length character string CHARACTER LARGE OBJECT Character large object string GRAPHIC Fixed-length graphic string GRAPHIC VARYING Varying-length graphic string DOUBLE-BYTE CHARACTER LARGE OBJECT Double-byte character large object string BINARY LARGE OBJECT Binary large object string DATE Date TIME Time TIMESTAMP Timestamp DATALINK Datalink ROWID Row ID DISTINCT Distinct type
NUMERIC_SCALE	SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if the parameter is not decimal, numeric, or binary.

Table 109. SYSPARMS table (continued)

Column Name	System Column Name	Data Type	Description
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	<p>The precision of all numeric parameters.</p> <p>Note: This column supplies the precision of all numeric data types, including single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits.</p> <p>Contains the null value if the parameter is not numeric.</p>
CCSID	CCSID	INTEGER Nullable	<p>The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB and DATALINK parameters.</p> <p>A CCSID of 0 indicates that the CCSID of the job at run time is used.</p> <p>Contains the null value if the parameter is numeric.</p>
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	<p>Maximum length of the string for binary, character, and graphic string data types.</p> <p>Contains the null value if the parameter is not a string.</p>
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	<p>Number of bytes for binary, character, and graphic string data types.</p> <p>Contains the null value if the parameter is not a string.</p>
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	<p>Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits:</p> <p>2 Binary; floating-point precision is specified in binary digits.</p> <p>10 Decimal; all other numeric types are specified in decimal digits.</p> <p>Contains the null value if the parameter is not numeric.</p>

SYSPARMS

Table 109. SYSPARMS table (continued)

Column Name	System Column Name	Data Type	Description
DATETIME_PRECISION	DATPRC	INTEGER Nullable	<p>The fractional part of a date, time, or timestamp.</p> <p>0 For DATE and TIME data types</p> <p>6 For TIMESTAMP data types (number of microseconds).</p> <p>Contains the null value if the parameter is not date, time, or timestamp.</p>
IS_NULLABLE	NULLS	VARCHAR(3)	<p>Indicates whether the parameter is nullable.</p> <p>NO The parameter does not allow nulls.</p> <p>YES The parameter does allow nulls.</p>
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	<p>A character string supplied with the COMMENT statement.</p> <p>Contains the null value if there is no long comment.</p>
ROW_TYPE	ROWTYPE	CHAR(1)	<p>Indicates the type of row. If this is a parameter to a procedure, this column contains the null value.</p> <p>P Parameter.</p> <p>R Result before casting.</p> <p>C Result after casting.</p>
DATA_TYPE_SCHEMA	TYPESHEMA	VARCHAR(128) Nullable	<p>Schema of the data type if this is a distinct type.</p> <p>Contains the null value if the parameter is not a distinct type.</p>
DATA_TYPE_NAME	TYPENAME	VARCHAR(128) Nullable	<p>Name of the data type if this is a distinct type.</p> <p>Contains the null value if the parameter is not a distinct type.</p>
AS_LOCATOR	ASLOCATOR	VARCHAR(3)	<p>Indicates whether the parameter was specified as a locator.</p> <p>NO The parameter was not specified as a locator.</p> <p>YES The parameter was specified as a locator.</p>
IASP_NUMBER	IASPNUMBER	SMALLINT	<p>Specifies the independent auxiliary storage pool (IASP) number.</p>

SYSPROCS

The SYSPROCS view contains one row for each procedure created by the CREATE PROCEDURE statement. The following table describes the columns in the SYSPROCS view:

Table 110. SYSPROCS view

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSHEMA	VARCHAR(128)	Schema name of the routine (procedure) instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ROUTINE_SCHEMA	PROCSHEMA	VARCHAR(128)	Name of the SQL schema (schema) that contains the routine.
ROUTINE_NAME	PROCNAME	VARCHAR(128)	Name of the routine.
ROUTINE_CREATED	RTNCREATE	TIMESTAMP	Identifies the timestamp when the routine was created.
ROUTINE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the routine.
ROUTINE_BODY	BODY	VARCHAR(8)	The type of the routine body: EXTERNAL This is an external routine. SQL This is an SQL routine.
EXTERNAL_NAME	EXTNAME	VARCHAR(279) Nullable	If this is an external routine, this column identifies the external program name. <ul style="list-style-type: none"> • For REXX, the external program name is <i>schema-name/source-file-name(member-name)</i>. • For Java programs, the external program name is an optional jar-id followed by a <i>fully-qualified-class-name!method-name</i> or <i>fully-qualified-class-name.method-name</i>. • For all other languages, the external program name is <i>schema-name/program-name</i>. Contains the null value if this is not an external routine.

SYSPROCS

Table 110. SYSPROCS view (continued)

Column Name	System Column Name	Data Type	Description
EXTERNAL_LANGUAGE	LANGUAGE	VARCHAR(8) Nullable	If this is an external routine, this column identifies the external program name. C The external program is written in C. C++ The external program is written in C++. CL The external program is written in CL. COBOL The external program is written in COBOL. COBOLLE The external program is written in ILE COBOL. FORTRAN The external program is written in FORTRAN. JAVA The external program is written in JAVA. PLI The external program is written in PL/I. REXX The external program is a REXX procedure. RPG The external program is written in RPG. RPGLE The external program is written in ILE RPG. Contains the null value if this is not an external routine.

Table 110. SYSPROCS view (continued)

Column Name	System Column Name	Data Type	Description
PARAMETER_STYLE	PARM_STYLE	VARCHAR(7) Nullable	If this is an external routine, this column identifies the parameter style (calling convention).
			DB2GNRL This is the DB2GENERAL calling convention.
			DB2SQL This is the DB2SQL calling convention.
			GENERAL This is the GENERAL calling convention.
			JAVA This is the JAVA calling convention.
			NULLS This is the GENERAL WITH NULLS calling convention.
			SQL This is the SQL standard calling convention.
			Contains the null value if this is not an external routine.
IS_DETERMINISTIC	DETERMINE	VARCHAR(3)	This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result.
			NO The routine is not deterministic.
			YES The routine is deterministic.
SQL_DATA_ACCESS	DATAACCESS	VARCHAR(8)	This column identifies whether a routine contains SQL and whether it reads or modifies data.
			NONE The routine does not contain any SQL statements.
			CONTAINS The routine contains SQL statements.
			READS The routine possibly reads data from a table or view.
			MODIFIES The routine possibly modifies data in a table or view or issues SQL DDL statements.

SYSPROCS

Table 110. SYSPROCS view (continued)

Column Name	System Column Name	Data Type	Description
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	If this is an SQL routine, this column identifies the path. Contains the null value if this is not an SQL routine.
PARAM_SIGNATURE	SIGNATURE	VARCHAR(510)	This column identifies the routine signature.
RESULT_SETS	RESULTS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
IN_PARMs	IN_PARMs	SMALLINT	Identifies the number of input parameters. 0 indicates that there are no input parameters.
OUT_PARMs	OUT_PARMs	SMALLINT	Identifies the number of output parameters. 0 indicates that there are no output parameters.
INOUT_PARMs	INOUT_PARM	SMALLINT	Identifies the number of input/output parameters. 0 indicates that there are no input/output parameters.
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
ROUTINE_DEFINITION	ROUTINEDEF	VARCHAR(24000) Nullable	If this is an SQL routine, this column contains the SQL routine body. Contains the null value if this is not an SQL routine or if the routine body cannot be contained in this column without truncation.
DBINFO	DBINFO	VARCHAR(3) Nullable	Identifies whether information about the database is passed to the procedure. NO No database information is passed to the procedure. YES Information about the database is passed to the procedure.
COMMIT_ON_RETURN	CMTONRET	VARCHAR(3) Nullable	This column identifies whether the procedure commits on a successful return from the procedure. NO A commit is not performed on successful return from the procedure. YES A commit is performed on successful return from the procedure.

Table 110. SYSPROCS view (continued)

Column Name	System Column Name	Data Type	Description
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
NEW_SAVEPOINT_LEVEL	NEWSAVEPTL	VARCHAR(3) Nullable	This column identifies whether the routine starts a new savepoint level. NO A new savepoint level is not started. YES A new savepoint level is started.

SYSREFCST

SYSREFCST

The SYSREFCST view contains one row for each foreign key in the SQL schema. The following table describes the columns in the SYSREFCST view:

Table 111. SYSREFCST view

Column Name	System Column Name	Data Type	Description
CONSTRAINT_SCHEMA	CDBNAME	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	RELNAME	VARCHAR(128)	Name of the constraint.
UNIQUE_CONSTRAINT_SCHEMA	UNQDBNAME	VARCHAR(128)	Name of the SQL schema containing the unique constraint referenced by the referential constraint.
UNIQUE_CONSTRAINT_NAME	UNQNAME	VARCHAR(128)	Name of the unique constraint referenced by the referential constraint.
MATCH_OPTION	MATCH	VARCHAR(7)	Match option. Will always be NONE.
UPDATE_RULE	UPDATE	VARCHAR(11)	Update Rule. <ul style="list-style-type: none">• NO ACTION• RESTRICT
DELETE_RULE	DELETE	VARCHAR(11)	Delete Rule <ul style="list-style-type: none">• NO ACTION• CASCADE• SET NULL• SET DEFAULT• RESTRICT
COLUMN_COUNT	COLCOUNT	INTEGER	Number of columns in the foreign key.

SYSROUTINEDEP

The SYSROUTINEDEP view records the dependencies of routines. The following table describes the columns in the SYSROUTINEDEP view:

Table 112. SYSROUTINEDEP view

Column name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
OBJECT_SCHEMA	BSCHEMA	VARCHAR(128)	Name of the SQL schema that contains the object.
OBJECT_NAME	BNAME	VARCHAR(128)	Name of the object the routine is dependent on.
OBJECT_TYPE	BTYPE	CHAR(10)	Indicates the object type of the object referenced in the routine: ALIAS The object is an alias. FUNCTION The object is a function. INDEX The object is an index. PROCEDURE The object is a procedure. SCHEMA The object is a schema. TABLE The object is a table. TYPE The object is a distinct type. VIEW The object is a view.
PARAM_SIGNATURE	SIGNATURE	VARCHAR(10000) Nullable	This column identifies the routine signature. Contains the null value if the object is not a routine.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number of the object.
NUMBER_OF_PARAMS	NUMPARMS	SMALLINT Nullable	Identifies the number of parameters. Contains the null value if the object is not a routine.

SYSROUTINES

SYSROUTINES

The SYSROUTINES table contains one row for each procedure created by the CREATE PROCEDURE statement and each function created by the CREATE FUNCTION statement. The following table describes the columns in the SYSROUTINES view:

Table 113. SYSROUTINES view

Column Name	System Column Name	Data Type	Description
SPECIFIC_SCHEMA	SPECSHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	SPECNAME	VARCHAR(128)	Specific name of the routine instance.
ROUTINE_SCHEMA	RTNSHEMA	VARCHAR(128)	Name of the SQL schema (schema) that contains the routine.
ROUTINE_NAME	RTNNAME	VARCHAR(128)	Name of the routine.
ROUTINE_TYPE	RTNTYPE	VARCHAR(9)	Type of the routine. PROCEDURE This is a procedure. FUNCTION This is a function.
ROUTINE_CREATED	RTNCREATE	TIMESTAMP	Identifies the timestamp when the routine was created.
ROUTINE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that defined the routine.
ROUTINE_BODY	BODY	VARCHAR(8)	The type of the routine body: EXTERNAL This is an external routine. SQL This is an SQL routine.
EXTERNAL_NAME	EXTNAME	VARCHAR(279) Nullable	If this is an external routine, this column identifies the external program name. <ul style="list-style-type: none"> • For REXX, the external program name is <i>schema-name/source-file-name(member-name)</i>. • For ILE service programs, the external program name is <i>schema-name/service-program-name(entry-point-name)</i>. • For Java programs, the external program name is an optional jar-id followed by a <i>fully-qualified-class-name!method-name</i> or <i>fully-qualified-class-name.method-name</i>. • For all other languages, the external program name is <i>schema-name/program-name</i>. Contains the null value if this is not an external routine.

Table 113. SYSROUTINES view (continued)

Column Name	System Column Name	Data Type	Description
EXTERNAL_LANGUAGE	LANGUAGE	VARCHAR(8) Nullable	If this is an external routine, this column identifies the external program name.
			C The external program is written in C.
			C++ The external program is written in C++.
			CL The external program is written in CL.
			COBOL The external program is written in COBOL.
			COBOLLE The external program is written in ILE COBOL.
			FORTRAN The external program is written in FORTRAN.
			JAVA The external program is written in JAVA.
			PLI The external program is written in PL/I.
			REXX The external program is a REXX procedure.
			RPG The external program is written in RPG.
			RPGLE The external program is written in ILE RPG.
			Contains the null value if this is not an external routine.

SYSROUTINES

Table 113. SYSROUTINES view (continued)

Column Name	System Column Name	Data Type	Description
PARAMETER_STYLE	PARAM_STYLE	VARCHAR(7) Nullable	If this is an external routine, this column identifies the parameter style (calling convention).
			DB2GNRL This is the DB2GENERAL calling convention.
			DB2SQL This is the DB2SQL calling convention.
			GENERAL This is the GENERAL calling convention.
			JAVA This is the JAVA calling convention.
			NULLS This is the GENERAL WITH NULLS calling convention.
			SQL This is the SQL standard calling convention.
			Contains the null value if this is not an external routine.
IS_DETERMINISTIC	DETERMINE	VARCHAR(3)	This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result.
			NO The routine is not deterministic.
			YES The routine is deterministic.
SQL_DATA_ACCESS	DATAACCESS	VARCHAR(8)	This column identifies whether a routine contains SQL and whether it reads or modifies data.
			NONE The routine does not contain any SQL statements.
			CONTAINS The routine contains SQL statements.
			READS The routine possibly reads data from a table or view.
			MODIFIES The routine possibly modifies data in a table or view or issues SQL DDL statements.

Table 113. SYSROUTINES view (continued)

Column Name	System Column Name	Data Type	Description
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	If this is an SQL routine, this column identifies the path. Contains the null value if this is not an SQL routine.
PARAM_SIGNATURE	SIGNATURE	VARCHAR(510)	This column identifies the routine signature.
NUMBER_OF_RESULTS	NUMRESULTS	SMALLINT	Identifies the number of results.
MAX_DYNAMIC_RESULT_SETS	RESULTS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
IN_PARAMS	IN_PARAMS	SMALLINT	Identifies the number of input parameters. 0 indicates that there are no input parameters.
OUT_PARAMS	OUT_PARAMS	SMALLINT	Identifies the number of output parameters. 0 indicates that there are no output parameters.
INOUT_PARAMS	INOUT_PARAM	SMALLINT	Identifies the number of input/output parameters. 0 indicates that there are no input/output parameters.
PARSE_TREE	PARSE_TREE	VARCHAR(666) FOR BIT DATA	If this is a routine, this column identifies the parse tree of the CREATE FUNCTION or CREATE PROCEDURE statement. It is only used internally.
PARAM_ARRAY	PARAM_ARRAY	VARCHAR(10008) FOR BIT DATA	If this is an external routine, this column identifies the parameter array built from the CREATE FUNCTION or CREATE PROCEDURE statement. It is only used internally.
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
ROUTINE_DEFINITION	ROUTINEDEF	DBCLOB(1048576) Nullable	If this is an SQL routine, this column contains the SQL routine body. Contains the null value if this is not an SQL routine or if the routine body cannot be contained in this column without truncation.

SYSROUTINES

Table 113. SYSROUTINES view (continued)

Column Name	System Column Name	Data Type	Description
FUNCTION_ORIGIN	ORIGIN	CHAR(1)	<p>Identifies the type of function. If this is a procedure, this column contains a blank.</p> <p>B This is a built-in function (defined by DB2 UDB for iSeries).</p> <p>E This is a user-defined function.</p> <p>U This is a user-defined function that is sourced on another function.</p> <p>S This is a system-generated function.</p>
FUNCTION_TYPE	TYPE	CHAR(1)	<p>Identifies the form of the function. If this is a procedure, this column contains a blank.</p> <p>S This is a scalar function.</p> <p>C This is a column function.</p> <p>T This is a table function.</p>
EXTERNAL_ACTION	EXTACTION	CHAR(1) Nullable	<p>Identifies whether the invocation of the function has external effects.</p> <p>E This function has external side effects.</p> <p>N This function does not have any external side effects.</p> <p>Contains the null value if the routine is a procedure.</p>
IS_NULL_CALL	NULL_CALL	VARCHAR(3) Nullable	<p>Identifies whether the function needs to be called if an input parameter is the null value.</p> <p>NO This function need not be called if an input parameter is the null value. If this is a scalar function, the result of the function is implicitly null if any of the operands are null. If this is a table function, the result of the function is an empty table if any of the operands are the null value.</p> <p>YES This function must be called even if an input operand is null.</p> <p>Contains the null value if the routine is a procedure.</p>

Table 113. SYSROUTINES view (continued)

Column Name	System Column Name	Data Type	Description
SCRATCH_PAD	SCRATCHPAD	INTEGER Nullable	<p>Identifies whether the address of a static memory area (scratch pad) is passed to the function.</p> <p>0 The function does not have a scratch pad.</p> <p>integer Indicates the size of the scratch pad passed to the function.</p> <p>Contains the null value if the routine is a procedure.</p>
FINAL_CALL	FINAL_CALL	VARCHAR(3) Nullable	<p>Indicates whether a final call to the function should be made to allow the function to clean up its work areas (scratch pads).</p> <p>NO No final call is made.</p> <p>YES A final call to the function is made when the statement is complete.</p> <p>Contains the null value if the routine is a procedure.</p>
PARALLELIZABLE	PARALLEL	VARCHAR(3) Nullable	<p>Identifies whether the function can be run in parallel.</p> <p>NO The function must be synchronous.</p> <p>YES The function can be run in parallel.</p> <p>Contains the null value if the routine is a procedure.</p>
DBINFO	DBINFO	VARCHAR(3) Nullable	<p>Identifies whether information about the database is passed to the routine.</p> <p>NO No database information is passed to the routine.</p> <p>YES Information about the database is passed to the routine.</p> <p>Contains the null value if the routine is a procedure.</p>
SOURCE_SPECIFIC_SCHEMA	SRCSCHEMA	VARCHAR(128) Nullable	<p>If this is sourced function and the source is user-defined, this column contains the name of the source schema. If this is a sourced function and the source is built-in, this column contains 'QSYS2'.</p> <p>Contains the null value if the routine is not a sourced function.</p>

SYSROUTINES

Table 113. SYSROUTINES view (continued)

Column Name	System Column Name	Data Type	Description
SOURCE_SPECIFIC_NAME	SRCNAME	VARCHAR(128) Nullable	If this is sourced function and the source is user-defined, this column contains the specific name of the source function name. Contains the null value if the routine is not a sourced function.
IS_USER_DEFINED_CAST	CAST_FUNC	VARCHAR(3) Nullable	Identifies whether the this function is a cast function created when a distinct type was created. NO This function is not a cast function. YES This function is a cast function. Contains the null value if the routine is a procedure.
CARDINALITY	CARD	BIGINT Nullable	Specifies the cardinality for a table function. Contains the null value if the function is not a table function or if cardinality was not specified.
FENCED	FENCED	VARCHAR(3) Nullable	Identifies whether a function is fenced. NO The function is not fenced. YES The function is fenced. Contains the null value if the routine is a procedure.
COMMIT_ON_RETURN	CMTONRET	VARCHAR(3) Nullable	This column identifies whether the procedure commits on a successful return from the procedure. NO A commit is not performed on successful return from the procedure. YES A commit is performed on successful return from the procedure. Contains the null value if the routine is a function.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

Table 113. SYSROUTINES view (continued)

Column Name	System Column Name	Data Type	Description
NEW_SAVEPOINT_LEVEL	NEWSAVEPTL	VARCHAR(3) Nullable	This column identifies whether the routine starts a new savepoint level. NO A new savepoint level is not started. YES A new savepoint level is started. Contains the null value if the routine is a function.
LAST_ALTERED	ALTEREDTS	TIMESTAMP Nullable	Routine last changed timestamp. Contains the null value.
DEBUG_MODE	DEBUG_MODE	CHAR(1)	Identifies whether the routine is debuggable. 0 The routine is not debuggable. 2 The routine is debuggable.
DEBUG_DATA	DEBUG_DATA	CLOB(1048576) Nullable	Reserved. Contains the null value.

SYSTABLES

SYSTABLES

The SYSTABLES view contains one row for every table, view or alias in the SQL schema, including the tables and views of the SQL catalog. The following table describes the columns in the SYSTABLES view:

Table 114. SYSTABLES view

Column name	System Column Name	Data Type	Description
TABLE_NAME	NAME	VARCHAR(128)	Name of the table, view or alias. This is the SQL table, view or alias name if it exists; otherwise, it is the system table, view or alias name.
TABLE_OWNER	CREATOR	VARCHAR(128)	Owner of the table, view or alias
TABLE_TYPE	TYPE	CHAR(1)	If the row describes a table, view, or alias: A Alias L Logical file P Physical file T Table V View
COLUMN_COUNT	COLCOUNT	INTEGER	Number of columns in the table or view. Zero for an alias.
ROW_LENGTH	RECLENGTH ⁸⁶	INTEGER	Maximum length of any record in the table. Zero for an alias.
TABLE_TEXT	LABEL	VARCHAR(50)	A character string provided with the LABEL statement.
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
TABLE_SCHEMA	DBNAME	VARCHAR(128)	Name of the SQL schema that contains the table, view or alias
LAST_ALTERED_TIMESTAMP	ALTEREDTS	TIMESTAMP	Table last changed timestamp
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10)	System table name.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10)	System schema name
FILE_TYPE	FILETYPE	CHAR(1)	File type D Data file or alias S Source file
BASE_TABLE_SCHEMA	TBDBNAME	VARCHAR(128) Nullable	For an alias, this is the name of the SQL schema that contains the table or view the alias is based on. Contains the null value if the table is not an alias.

Table 114. SYSTABLES view (continued)

Column name	System Column Name	Data Type	Description
BASE_TABLE_NAME	TBNAME	VARCHAR(128) Nullable	For an alias, this is the name of the table or view the alias is based on. Contains the null value if the table is not an alias.
BASE_TABLE_MEMBER	TBMEMBER	VARCHAR(10) Nullable	For an alias, this is the name of the file member the alias is based on. Contains *FIRST if this is an alias, but a member name was not specified. Contains the null value if the table is not an alias.
SYSTEM_TABLE	SYSTABLE	CHAR(1)	System table N The table is not a system table. Y The table is a system table.
SELECT_OMIT	SELECTOMIT	CHAR(1)	Select/omit logical file N The table is not a select/omit logical file. Y The table is a select/omit logical file.
IS_INSERTABLE_INTO	INSERTABLE	VARCHAR(3)	Identifies whether an INSERT is allowed on the table. NO An INSERT is not allowed on this table. YES An INSERT is allowed on this table.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

86. The length is the number of bytes passed in database buffers, not the internal storage length.

SYSTRIGCOL

SYSTRIGCOL

The SYSTRIGCOL view contains one row for each column either implicitly or explicitly referenced in the WHEN clause or the triggered SQL statements of a trigger. The following table describes the columns in the SYSTRIGCOL view:

Table 115. SYSTRIGCOL view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the schema containing the table or view that contains the column that is referenced in the trigger.
TABLE_NAME	TABNAME	VARCHAR(128)	Name of the table or view that contains the column that is referenced in the trigger.
COLUMN_NAME	TABCOLUMN	VARCHAR(128)	Name of the column that is referenced in the trigger.
OBJECT_TYPE	BTYPE	VARCHAR(10)	Indicates the object type of the object that contains the column referenced in the trigger: FUNCTION The object is a function. TABLE The object is a table. VIEW The object is a view.

SYSTRIGDEP

The SYSTRIGDEP view contains one row for each object referenced in the WHEN clause or the triggered SQL statements of a trigger. The following table describes the columns in the SYSTRIGDEP view:

Table 116. SYSTRIGDEP view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
OBJECT_SCHEMA	BSCHEMA	VARCHAR(128)	Name of the schema containing the object referenced in the trigger.
OBJECT_NAME	BNAME	VARCHAR(128)	Name of the object referenced in the trigger.
OBJECT_TYPE	BTYPE	CHAR(10)	Indicates the object type of the object referenced in the trigger: ALIAS The object is an alias. FUNCTION The object is a function. INDEX The object is an index. PACKAGE The object is a package. PROCEDURE The object is a procedure. SCHEMA The object is a schema. TABLE The object is a table. TYPE The object is a distinct type. VIEW The object is a view.
PARM_SIGNATURE	SIGNATURE	VARCHAR(10000) Nullable	This column identifies the routine signature. Contains the null value if the object is not a routine.

SYSTRIGGERS

SYSTRIGGERS

The SYSTRIGGERS view contains one row for each trigger in an SQL schema. The following table describes the columns in the SYSTRIGGERS view:

Table 117. SYSTRIGGERS view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
EVENT_MANIPULATION	TRIGEVENT	VARCHAR(6)	Indicates the event that causes the trigger to fire: DELETE Trigger fires on a DELETE. INSERT Trigger fires on a INSERT. UPDATE Trigger fires on a DELETE. READ Trigger fires when a row is read. This is only valid for triggers created via the ADDPFTRG command.
EVENT_OBJECT_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the schema containing the subject table of the trigger.
EVENT_OBJECT_TABLE	TABNAME	VARCHAR(128)	Name of the subject table of the trigger.
ACTION_ORDER	ORDERSEQNO	INTEGER	The ordinal position this trigger in the list of triggers for the table. This indicates the order in which the trigger will be fired.
ACTION_CONDITION	CONDITION	DBCLOB(1048576) Nullable	Text of the WHEN clause for the trigger. Contains the null value if there is no WHEN clause.
ACTION_STATEMENT	TEXT	DBCLOB(1048576) Nullable	Text of the SQL statements in the trigger action. Contains the null value if this is a trigger created via the ADDPFTRG command.
ACTION_ORIENTATION	GRANULAR	VARCHAR(9)	Indicates whether this is a ROW or STATEMENT trigger: ROW Trigger fires for each ROW. STATEMENT Trigger fires for each statement.

Table 117. SYSTRIGGERS view (continued)

Column Name	System Column Name	Data Type	Description
ACTION_TIMING	TRIGTIME	VARCHAR(6)	Indicates whether this is a BEFORE or AFTER trigger: BEFORE Trigger fires before the triggering event. AFTER Trigger fires after the triggering event.
TRIGGER_MODE	TRIGMODE	VARCHAR(6)	Indicates the firing mode for the trigger: DB2SQL The trigger mode is DB2SQL. DB2ROW The trigger mode is DB2ROW.
ACTION_REFERENCE_OLD_ROW	OLD_ROW	VARCHAR(128) Nullable	Name of the OLD ROW correlation name. Contains the null value if an OLD ROW correlation name was not specified.
ACTION_REFERENCE_NEW_ROW	NEW_ROW	VARCHAR(128) Nullable	Name of the NEW ROW correlation name. Contains the null value if a NEW ROW correlation name was not specified.
ACTION_REFERENCE_OLD_TABLE	OLD_TABLE	VARCHAR(128) Nullable	Name of the OLD TABLE correlation name. Contains the null value if an OLD TABLE correlation name was not specified.
ACTION_REFERENCE_NEW_TABLE	NEW_TABLE	VARCHAR(128) Nullable	Name of the NEW TABLE correlation name. Contains the null value if a NEW TABLE correlation name was not specified.
SQL_PATH	SQL_PATH	VARCHAR(3483) Nullable	SQL path used when the trigger was created. Contains the null value if the trigger was created via the ADDPFTRG command.
CREATED	CREATE_DTS	TIMESTAMP	Timestamp when the trigger was created.
TRIGGER_PROGRAM_NAME	TRIGPGM	VARCHAR(128)	Name of the trigger program.
TRIGGER_PROGRAM_LIBRARY	TRIGPMLIB	VARCHAR(128)	System name of the schema containing the trigger program.

SYSTRIGGERS

Table 117. SYSTRIGGERS view (continued)

Column Name	System Column Name	Data Type	Description
OPERATIVE	OPERATIVE	VARCHAR(1)	<p>Indicates whether the trigger is operative (is associated with a file that has a member).</p> <p>Y The trigger is operative.</p> <p>N The trigger is inoperative.</p>
ENABLED	ENABLED	VARCHAR(1)	<p>Indicates whether the trigger is enabled (see the CL command CHGPFTRG)</p> <p>Y The trigger is enabled.</p> <p>N The trigger is disabled.</p>
THREADSAFE	THDSAFE	VARCHAR(8)	<p>Indicates whether the trigger is thread safe.</p> <p>YES The trigger is thread safe.</p> <p>NO The trigger is not thread safe.</p> <p>UNKNOWN The thread safety of the trigger is unknown.</p>
MULTITHREADED_JOB_ACTION	MLTTHDACN	VARCHAR(8)	<p>Indicates the action to take when the trigger program is called in a multithreaded job.</p> <p>SYSVAL Use the QMLTTHDACN system value to determine the action to take.</p> <p>MSG Run the trigger program in a multithreaded job, but send a diagnostic message.</p> <p>NORUN Do not run the trigger program in a multithreaded job.</p> <p>RUN Run the trigger program in a multithreaded job.</p>
ALLOW_REPEATED_CHANGE	ALWREPCHG	VARCHAR(8)	<p>Indicates the condition under which an update event fires the trigger.</p> <p>YES The trigger allows repeated changes to the same row.</p> <p>NO The trigger does not allow repeated changes to the same row.</p>

Table 117. SYSTRIGGERS view (continued)

Column Name	System Column Name	Data Type	Description
TRIGGER_UPDATE_CONDITION	TRGUPDCND	CHAR(8) Nullable	<p>Indicates whether an UPDATE trigger is always fired on an update event or only when a column value is actually changed.</p> <p>ALWAYS The trigger is always fired on an update event.</p> <p>CHANGE The trigger is only fired on an update event if a column value is actually changed.</p> <p>Contains the null value if the trigger is not an UPDATE trigger.</p>
LONG_COMMENT	REMARKS	VARGRAPHIC(2000) Nullable	<p>A character string supplied with the COMMENT statement.</p> <p>Contains the null value if there is no long comment.</p>

SYSTRIGUPD

SYSTRIGUPD

The SYSTRIGUPD view contains one row for each column identified in the UPDATE column list, if any. The following table describes the columns in the SYSTRIGUPD view:

Table 118. SYSTRIGUPD view

Column Name	System Column Name	Data Type	Description
TRIGGER_SCHEMA	TRIGSCHEMA	VARCHAR(128)	Name of the schema containing the trigger.
TRIGGER_NAME	TRIGNAME	VARCHAR(128)	Name of the trigger.
EVENT_OBJECT_SCHEMA	TABSCHEMA	VARCHAR(128)	Name of the schema containing the subject table of the trigger.
EVENT_OBJECT_TABLE	TABNAME	VARCHAR(128)	Name of the subject table of the trigger.
TRIGGERED_UPDATE_COLUMNS	TABCOLUMN	VARCHAR(128)	Name of a column specified in the UPDATE column list of the trigger.

SYSTYPES

The SYSTYPES table contains one row for each built-in data type and each distinct type created by the CREATE DISTINCT TYPE statement. The following table describes the columns in the SYSTYPES table:

Table 119. SYSTYPES table

Column Name	System Column Name	Data Type	Description
USER_DEFINED_TYPE_SCHEMA	TYPESHEMA	VARCHAR(128)	Schema name of the data type.
USER_DEFINED_TYPE_NAME	TYPENAME	VARCHAR(128)	Name of the data type.
USER_DEFINED_TYPE_DEFINER	DEFINER	VARCHAR(128)	Name of the user that created the data type.
SOURCE_SCHEMA	SRCSHEMA	VARCHAR(128) Nullable	The schema for the source data type of this data type. Contains the null value if this is a built-in data type.
SOURCE_TYPE	SRCTYPE	VARCHAR(128) Nullable	Name of the source data type of this data type. Contains the null value if this is a built-in data type.
SYSTEM_TYPE_SCHEMA	SYSTSHEMA	CHAR(10)	System schema name of the data type.
SYSTEM_TYPE_NAME	SYSTEMNAME	CHAR(10)	System name of the data type.
METATYPE	METATYPE	CHAR(1)	Indicates the type of data type. S System predefined data type. T User-defined distinct type.

SYSTYPES

Table 119. SYSTYPES table (continued)

Column Name	System Column Name	Data Type	Description
LENGTH	LENGTH	INTEGER	<p>The length attribute of the data type; or, in the case of a decimal, numeric, or nonzero precision binary column, its precision:</p> <p>8 bytes BIGINT</p> <p>4 bytes INTEGER</p> <p>2 bytes SMALLINT</p> <p>Precision of number DECIMAL</p> <p>Precision of number NUMERIC</p> <p>8 bytes FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION</p> <p>4 bytes FLOAT(n) where n = 1 to 24, or REAL</p> <p>Length of string CHARACTER</p> <p>Maximum length of string VARCHAR or CLOB</p> <p>Length of graphic string GRAPHIC</p> <p>Maximum length of graphic string VARGRAPHIC or DBCLOB</p> <p>Maximum length of binary string BLOB</p> <p>4 bytes DATE</p> <p>3 bytes TIME</p> <p>10 bytes TIMESTAMP</p> <p>Maximum length of datalink URL and comment DATALINK</p> <p>40 bytes ROWID</p> <p>Same value as the source type DISTINCT</p>
NUMERIC_SCALE	SCALE	INTEGER Nullable	<p>Scale of numeric data.</p> <p>Contains the null value if the data type is not decimal, numeric, or binary.</p>

Table 119. SYSTYPES table (continued)

Column Name	System Column Name	Data Type	Description
CCSID	CCSID	INTEGER Nullable	The CCSID value for CHAR, VARCHAR, CLOB, DATE, TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, DBCLOB and DATALINK data types. Contains the null value if the data type is numeric.
STORAGE	STORAGE	INTEGER	The storage requirements for the column: 8 bytes BIGINT 4 bytes INTEGER 2 bytes SMALLINT (Precision/2) + 1 DECIMAL Precision of number NUMERIC 8 bytes FLOAT, FLOAT(n) where n = 25 to 53, or DOUBLE PRECISION 4 bytes FLOAT(n) where n = 1 to 24, or REAL Length of string CHAR Maximum length of string + 2 VARCHAR Maximum length of string + 29 CLOB Length of string * 2 GRAPHIC Maximum length of string * 2 + 2 VARGRAPHIC Maximum length of string * 2 + 29 DBCLOB 4 bytes DATE 3 bytes TIME 10 bytes TIMESTAMP Maximum length of datalink URL and comment + 24 DATALINK 42 bytes ROWID Same value as the source type DISTINCT Note: This column supplies the storage requirements for all data types.

SYSTYPES

Table 119. SYSTYPES table (continued)

Column Name	System Column Name	Data Type	Description
NUMERIC_PRECISION	PRECISION	INTEGER Nullable	<p>The precision of all numeric data types.</p> <p>Note: This column supplies the precision of all numeric data types, including single- and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits.</p> <p>Contains the null value if the data type is not numeric.</p>
CHARACTER_MAXIMUM_LENGTH	CHARLEN	INTEGER Nullable	<p>Maximum length of the string for binary, character, and graphic string data types.</p> <p>Contains the null value if the data type is not a string.</p>
CHARACTER_OCTET_LENGTH	CHARBYTE	INTEGER Nullable	<p>Number of bytes for binary, character, and graphic string data types.</p> <p>Contains the null value if the data type is not a string.</p>
ALLOCATE	ALLOCATE	INTEGER Nullable	<p>Allocated length of the string for binary, varying-length character, and varying-length graphic string data types.</p> <p>Contains the null value if the data type is numeric or fixed-length.</p>
NUMERIC_PRECISION_RADIX	RADIX	INTEGER Nullable	<p>Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits:</p> <p>2 Binary; floating-point precision is specified in binary digits.</p> <p>10 Decimal; all other numeric types are specified in decimal digits.</p> <p>Contains the null value if the data type is not numeric.</p>
DATETIME_PRECISION	DATPRC	INTEGER Nullable	<p>The fractional part of a date, time, or timestamp.</p> <p>0 For DATE and TIME data types</p> <p>6 For TIMESTAMP data types (number of microseconds).</p> <p>Contains the null value if the data type is not date, time, or timestamp.</p>

Table 119. SYSTYPES table (continued)

Column Name	System Column Name	Data Type	Description
CREATE_TIME	CRTTIME	TIMESTAMP	Identifies the timestamp when the data type was created.
LONG_COMMENT	REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
IASP_NUMBER 	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number of the data type.
LAST_ ALTERED	ALTEREDTS	TIMESTAMP Nullable	Reserved. Contains the null value.

SYSVIEWDEP

SYSVIEWDEP

The SYSVIEWDEP view records the dependencies of views on tables, including the views of the SQL catalog. The following table describes the columns in the SYSVIEWDEP view:

Table 120. SYSVIEWDEP view

Column name	System Column Name	Data Type	Description
VIEW_NAME	DNAME	VARCHAR(128)	Name of the view. This is the SQL view name if it exists; otherwise, it is the system view name.
VIEW_OWNER	DCREATOR	VARCHAR(128)	Owner of the view
OBJECT_NAME	ONAME	VARCHAR(128)	Name of the object the view is dependent on.
OBJECT_SCHEMA	OSHEMA	VARCHAR(128)	Name of the SQL schema that contains the object the view is dependent on.
OBJECT_TYPE	OTYPE	CHAR(10)	Type of object the view was based on: FUNCTION Function TABLE Table TYPE Distinct Type VIEW View
VIEW_SCHEMA	DDBNAME	VARCHAR(128)	Name of the schema of the view.
SYSTEM_VIEW_NAME	SYS_VNAME	CHAR(10)	System View name
SYSTEM_VIEW_SCHEMA	SYS_VDNAME	CHAR(10)	System View schema
SYSTEM_TABLE_NAME	SYS_TNAME	CHAR(10) Nullable	System Table name. Contains the null value if the object is a function or distinct type.
SYSTEM_TABLE_SCHEMA	SYS_DNAME	CHAR(10) Nullable	System Table schema. Contains the null value if the object is a function or distinct type.
TABLE_NAME	BNAME	VARCHAR(128) Nullable	Name of the table or view the view is dependent on. This is the SQL view name if it exists; otherwise, it is the system view name. Contains the null value if the object is a function or distinct type.
TABLE_OWNER	BCREATOR	VARCHAR(128) Nullable	Owner of the table or view the view is dependent on. Contains the null value if the object is a function or distinct type.
TABLE_SCHEMA	BDBNAME	VARCHAR(128) Nullable	Name of the SQL schema that contains the table or view the view is dependent on. Contains the null value if the object is a function or distinct type.

Table 120. SYSVIEWDEP view (continued)

Column name	System Column Name	Data Type	Description
TABLE_TYPE	BTYPE	CHAR(1) Nullable	Type of object the view was based on: T Table P Physical file V View L Logical file Contains the null value if the object is a function or distinct type.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.
PARM_SIGNATURE	SIGNATURE	VARCHAR(10000) Nullable	This column identifies the routine signature. Contains the null value if the object is not a routine.

SYSVIEWS

SYSVIEWS

The SYSVIEWS view contains one row for each view in the SQL schema, including the views of the SQL catalog. The following table describes the columns in the SYSVIEWS view:

Table 121. SYSVIEWS view

Column Name	System Column Name	Data Type	Description
TABLE_NAME	NAME	VARCHAR(128)	Name of the view. This is the SQL view name if it exists; otherwise, it is the system view name.
VIEW_OWNER	CREATOR	VARCHAR(128)	Owner of the view
SEQNO	SEQNO	INTEGER	Sequence number of this row; will always be 1.
CHECK_OPTION	CHECK	CHAR(1)	The check option used on the view N No check option was specified Y The local option was specified C The cascaded option was specified
VIEW_DEFINITION	TEXT	VARCHAR(10000) Nullable	The query expression portion of the CREATE VIEW statement. Contains the null value if the view definition cannot be contained in the column without truncation.
IS_UPDATABLE	UPDATES	CHAR(1)	Specifies if the view is updatable: Y The view is updatable N The view is read-only
TABLE_SCHEMA	DBNAME	VARCHAR(128)	Name of the SQL schema that contains the view.
SYSTEM_VIEW_NAME	SYS_VNAME	CHAR(10)	System View name
SYSTEM_VIEW_SCHEMA	SYS_VDNAME	CHAR(10)	System View schema name
IS_INSERTABLE_INTO	INSERTABLE	VARCHAR(3)	Identifies whether an INSERT is allowed on the view. NO An INSERT is not allowed on this view. YES An INSERT is allowed on this view.
IASP_NUMBER	IASPNUMBER	SMALLINT	Specifies the independent auxiliary storage pool (IASP) number.

ODBC and JDBC Catalog Views

The catalog includes the following views and tables in the SYSIBM library:

View Name	Description
"SQLCOLPRIVILEGES" on page 950	Information about privileges granted on columns
"SQLCOLUMNS" on page 951	Information about column attributes
"SQLFOREIGNKEYS" on page 956	Information about foreign keys
"SQLPRIMARYKEYS" on page 957	Information about primary keys
"SQLPROCEDURECOLS" on page 958	Information about procedure parameters
"SQLPROCEDURES" on page 962	Information about procedures
"SQLSCHEMAS" on page 963	Information about schemas
"SQLSPECIALCOLUMNS" on page 964	Information about columns of a table that can be used to uniquely identify a row
"SQLSTATISTICS" on page 966	Statistical information about tables
"SQLTABLEPRIVILEGES" on page 967	Information about privileges granted on tables
"SQLTABLES" on page 968	Information about tables
"SQLTYPEINFO" on page 969	Information about the types of tables
"SQLUDTS" on page 974	Information about built-in data types and distinct types

SQLCOLPRIVILEGES

SQLCOLPRIVILEGES

The SQLCOLPRIVILEGES view contains one row for every privileges granted on a column. Note that this catalog view cannot be used to determine whether a user is authorized to a column because the privilege to use a column could be acquired through a group user profile or special authority (such as *ALLOBJ). The following table describes the columns in the view:

Table 122. SQLCOLPRIVILEGES view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name.
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	VARCHAR(128)	Table name.
COLUMN_NAME	VARCHAR(128)	Column name.
GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
PRIVILEGE	VARCHAR(10)	The privilege granted: UPDATE The privilege to update the column. REFERENCES The privilege to reference the column in a referential constraint.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. NO The privilege is not grantable. YES The privilege is grantable.
DBNAME	VARCHAR(8) Nullable	Reserved. The column contains the null value.

SQLCOLUMNS

The SQLCOLUMNS view contains one row for every column in a table, view, or alias. The following table describes the columns in the view:

Table 123. SQLCOLUMNS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name.
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	VARCHAR(128)	Table name.
COLUMN_NAME	VARCHAR(128)	Column name.
DATA_TYPE	SMALLINT	The data type of the column:
	-5	BIGINT
	4	INTEGER
	5	SMALLINT
	3	DECIMAL
	2	NUMERIC
	8	DOUBLE PRECISION
	7	REAL
	1	CHARACTER
	-2	CHARACTER FOR BIT DATA
	12	VARCHAR
	-3	VARCHAR FOR BIT DATA
	40	CLOB
	-95	GRAPHIC
	-96	VARGRAPHIC
	-350	DBCLOB
	30	BLOB
	91	DATE
	92	TIME
	93	TIMESTAMP
	70	DATALINK
	-100	ROWID
	17	DISTINCT

SQLCOLUMNS

Table 123. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(128)	The name of the data type of the column:
		BIGINT BIGINT
		INTEger INTEGER
		SMALLINT SMALLINT
		DECIMAL DECIMAL
		NUMERIC NUMERIC
		FLOAT DOUBLE PRECISION
		REAL REAL
		CHARacter CHARACTER
		CHARacter FOR BIT DATA CHARACTER FOR BIT DATA
		VARCHAR VARCHAR
		VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA
		CLOB CLOB
		GRAPHIC GRAPHIC
		VARGRAPHIC VARGRAPHIC
		DBCLOB DBCLOB
		BLOB BLOB
		DATE DATE
		TIME TIME
		TIMESTAMP TIMESTAMP
		DATALINK DATALINK
		ROWID ROWID
		Qualified Type Name DISTINCT
COLUMN_SIZE	INTEGER	The length of the column.
BUFFER_LENGTH	INTEGER	Indicates the length of the column in a buffer.
DECIMAL_DIGITS	SMALLINT Nullable	Indicates the number of digits for a numeric column. Contains the null value if the object is not numeric.
NUM_PREC_RADIX	SMALLINT Nullable	Indicates the radix of a numeric column. Contains the null value if the object is not numeric.
NULLABLE	SMALLINT	Indicates whether the column can contain the null value. 0 The column does not allow nulls. 1 The column does allow nulls.
REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.

| Table 123. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
COLUMN_DEF	VARCHAR(2000) Nullable	The default value of the column. Contains the null value if there is no default value.
SQL_DATA_TYPE	SMALLINT	Indicates the SQL data type of the column.
SQL_DATETIME_SUB	SMALLINT Nullable	The datetime subtype of the data type: 1 DATE 2 TIME 3 TIMESTAMP Contains the null value if the column is not a datetime data type.
CHAR_OCTET_LENGTH	INTEGER Nullable	Indicates the length in characters of the column. Contains the null value if the column is not a string.
ORDINAL_POSITION	INTEGER	Indicates the ordinal position of the column in the table.
IS_NULLABLE	VARCHAR(3)	Indicates whether the column can contain the null value. NO The column is not nullable. YES The column is nullable.

SQLCOLUMNS

| Table 123. SQLCOLUMNS view (continued)

Column Name	Data Type	Description
JDBC_DATA_TYPE	SMALLINT	Indicates the JDBC data type of the column.
	-5	BIGINT
	4	INTEGER
	5	SMALLINT
	3	DECIMAL
	2	NUMERIC
	8	DOUBLE PRECISION
	7	REAL
	1	CHARACTER
	-2	CHARACTER FOR BIT DATA
	12	VARCHAR
	-3	VARCHAR FOR BIT DATA
	2005	CLOB
	1	GRAPHIC
	12	VARGRAPHIC
	1111	DBCLOB
	2004	BLOB
	91	DATE
	92	TIME
	93	TIMESTAMP
	70	DATALINK
	1111	ROWID
	2001	DISTINCT
SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_TABLE	VARCHAR(128) Nullable	Reserved. Contains the null value.
SOURCE_DATA_TYPE	VARCHAR(128) Nullable	The source data type if the data type of the column is a distinct type. Contains the null value if the data type is not a distinct type.
DBNAME	VARCHAR(8) Nullable	Reserved. Contains the null value.
COLUMN_TEXT	VARCHAR(50) Nullable	The text of the column. Contains the null value if the column has no column text.

| *Table 123. SQLCOLUMNS view (continued)*

Column Name	Data Type	Description
PSEUDO_COLUMN	SMALLINT	Indicates whether this is a ROWID or identity column.
		1 The column is not a ROWID or identity column.
		2 The column is a ROWID or identity column.

SQLFOREIGNKEYS

SQLFOREIGNKEYS

The SQLFOREIGNKEYS view contains one row for every referential constraint key on a table. The following table describes the columns in the view:

Table 124. SQLFOREIGNKEYS view

Column Name	Data Type	Description
PKTABLE_CAT	VARCHAR(128)	Relational database name
PKTABLE_SCHEM	VARCHAR(128)	Name of the SQL schema containing the parent table.
PKTABLE_NAME	VARCHAR(128)	Parent table name.
PKCOLUMN_NAME	VARCHAR(128)	Parent key column name.
FKTABLE_CAT	VARCHAR(128)	Relational database name
FKTABLE_SCHEM	VARCHAR(128)	Name of the SQL schema containing the dependent table of the referential constraint.
FKTABLE_NAME	VARCHAR(128)	Dependent table name of the referential constraint.
FKCOLUMN_NAME	VARCHAR(128)	Dependent key name.
KEY_SEQ	SMALLINT	The position of the column within the key.
UPDATE_RULE	SMALLINT	Update Rule: 1 RESTRICT 3 NO ACTION
DELETE_RULE	SMALLINT	Delete Rule: 0 CASCADE 1 RESTRICT 2 SET NULL 3 NO ACTION 4 SET DEFAULT
FK_NAME	VARCHAR(128)	Name of the referential constraint
PK_NAME	VARCHAR(128)	Name of the unique constraint
DEFERRABILITY	SMALLINT	Indicates whether the constraint checking can be deferred. Will always be 7.

SQLPRIMARYKEYS

The SQLPRIMARYKEYS view contains one row for every primary constraint key on a table. The following table describes the columns in the view:

Table 125. SQLPRIMARYKEYS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the schema containing the table with the primary key.
TABLE_NAME	VARCHAR(128)	Name of the table with the primary key.
COLUMN_NAME	VARCHAR(128)	Name of a primary key column.
KEY_SEQ	SMALLINT	The position of the column within the key.
PK_NAME	VARCHAR(128)	Name of the primary key constraint.

SQLPROCEDURECOLS

SQLPROCEDURECOLS

The SQLPROCEDURECOLS view contains one row for every parameter of a procedure. The following table describes the columns in the view:

Table 126. SQLPROCEDURECOLS view

Column Name	Data Type	Description
PROCEDURE_CAT	VARCHAR(128)	Relational database name
PROCEDURE_SCHEM	VARCHAR(128)	Schema name of the procedure instance.
PROCEDURE_NAME	VARCHAR(128)	Name of the procedure instance.
COLUMN_NAME	VARCHAR(128) Nullable	Name of a procedure parameter. Contains the null value if the parameter does not have a name.
COLUMN_TYPE	SMALLINT	Type of the parameter: 1 IN 2 INOUT 4 OUT
DATA_TYPE	SMALLINT	The data type of the parameter: -5 BIGINT 4 INTEGER 5 SMALLINT 3 DECIMAL 2 NUMERIC 8 DOUBLE PRECISION 7 REAL 1 CHARACTER -2 CHARACTER FOR BIT DATA 12 VARCHAR -3 VARCHAR FOR BIT DATA 40 CLOB -95 GRAPHIC -96 VARGRAPHIC -350 DBCLOB 30 BLOB 91 DATE 92 TIME 93 TIMESTAMP 70 DATALINK -100 ROWID 17 DISTINCT

Table 126. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(260)	The name of the data type of the parameter:
		BIGINT BIGINT
		INTEger INTEGER
		SMALLINT SMALLINT
		DECIMAL DECIMAL
		NUMERIC NUMERIC
		FLOAT DOUBLE PRECISION
		REAL REAL
		CHARacter CHARACTER
		CHARacter FOR BIT DATA CHARACTER FOR BIT DATA
		VARCHAR VARCHAR
		VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA
		CLOB CLOB
		GRAPHIC GRAPHIC
		VARGRAPHIC VARGRAPHIC
		DBCLOB DBCLOB
		BLOB BLOB
		DATE DATE
		TIME TIME
		TIMESTAMP TIMESTAMP
		DATALINK DATALINK
		ROWID ROWID
		Qualified Type Name DISTINCT
COLUMN_SIZE	INTEGER	Length of the parameter.
BUFFER_LENGTH	INTEGER	Indicates the length of the parameter in a buffer.
DECIMAL_DIGITS	SMALLINT Nullable	Scale of numeric or datetime data. Contains the null value if the parameter is not decimal, numeric, binary, time or timestamp.
NUM_PREC_RADIX	SMALLINT Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: 2 Binary; floating-point precision is specified in binary digits. 10 Decimal; all other numeric types are specified in decimal digits. Contains the null value if the parameter is not numeric.

SQLPROCEDURECOLS

| Table 126. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description
NULLABLE	SMALLINT	Indicates whether the parameter is nullable.
		0 The parameter does not allow nulls.
		1 The parameter does allow nulls.
REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement.
		Contains the null value if there is no long comment.
COLUMN_DEF	VARCHAR(1) Nullable	The default value for the column.
		Contains the null value if there is no default value.
SQL_DATA_TYPE	SMALLINT	The SQL data type of the parameter:
		-5 BIGINT
		4 INTEGER
		5 SMALLINT
		3 DECIMAL
		2 NUMERIC
		8 DOUBLE PRECISION
		7 REAL
		1 CHARACTER
		-2 CHARACTER FOR BIT DATA
		12 VARCHAR
		-3 VARCHAR FOR BIT DATA
		-99 CLOB
		-95 GRAPHIC
		-96 VARGRAPHIC
		-350 DBCLOB
		-98 BLOB
9 DATE		
10 TIME		
11 TIMESTAMP		
70 DATALINK		
-100 ROWID		
17 DISTINCT		
SQL_DATETIME_SUB	SMALLINT Nullable	The datetime subtype of the parameter:
		1 DATE
		2 TIME
		3 TIMESTAMP
		Contains the null value if the data type is not a datetime data type.

Table 126. SQLPROCEDURECOLS view (continued)

Column Name	Data Type	Description
CHAR_OCTET_LENGTH	INTEGER Nullable	Indicates the length in characters of the parameter. Contains the null value if the column is not a string.
ORDINAL_POSITION	INTEGER	Numeric place of the parameter in the parameter list, ordered from left to right.
IS_NULLABLE	VARCHAR(3)	Indicates whether the parameter is nullable. NO The parameter does not allow nulls. YES The parameter does allow nulls.
JDBC_DATA_TYPE	SMALLINT	The JDBC data type of the parameter: -5 BIGINT 4 INTEGER 5 SMALLINT 3 DECIMAL 2 NUMERIC 8 DOUBLE PRECISION 7 REAL 1 CHARACTER -2 CHARACTER FOR BIT DATA 12 VARCHAR -3 VARCHAR FOR BIT DATA 2005 CLOB 1 GRAPHIC 12 VARGRAPHIC 1111 DBCLOB 2004 BLOB 91 DATE 92 TIME 93 TIMESTAMP 70 DATALINK 1111 ROWID 2001 DISTINCT

SQLPROCEDURES

SQLPROCEDURES

The SQLPROCEDURES view contains one row for every procedure. The following table describes the columns in the view:

Table 127. SQLPROCEDURES view

Column Name	Data Type	Description
PROCEDURE_CAT	VARCHAR(128)	Relational database name
PROCEDURE_SCHEM	VARCHAR(128)	Name of the schema of the procedure instance.
PROCEDURE_NAME	VARCHAR(128)	Name of the procedure.
NUM_INPUT_PARAMS	SMALLINT	Identifies the number of input parameters. 0 indicates that there are no input parameters.
NUM_OUTPUT_PARAMS	SMALLINT	Identifies the number of output parameters. 0 indicates that there are no output parameters.
NUM_RESULT_SETS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
PROCEDURE_TYPE	SMALLINT	Reserved. Contains 0.
NUM_INOUT_PARAMS	SMALLINT	Identifies the number of input/output parameters. 0 indicates that there are no input/output parameters.

SQLSCHEMAS

The SQLSCHEMAS view contains one row for every schema. The following table describes the columns in the view:

Table 128. SQLSCHEMAS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the schema.
TABLE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
TABLE_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
REMARKS	VARCHAR(2000) Nullable	Reserved. Contains the null value.
DBNAME	VARCHAR(8) Nullable	Reserved. Contains the null value.
SCHEMA_TEXT	VARCHAR(50)	A character string that describes the schema. Contains the empty string if there is no text.

SQLSPECIALCOLUMNS

SQLSPECIALCOLUMNS

The SQLSPECIALCOLUMNS view contains one row for every column of a primary key, unique constraint, or unique index that can identify a row of the table. The following table describes the columns in the view:

Table 129. SQLSPECIALCOLUMNS view

Column Name	Data Type	Description
SCOPE	SMALLINT	Reserved. Contains 0.
COLUMN_NAME	VARCHAR(128)	Column name
DATA_TYPE	SMALLINT	The data type of the column:
	-5	BIGINT
	4	INTEGER
	5	SMALLINT
	3	DECIMAL
	2	NUMERIC
	8	DOUBLE PRECISION
	7	REAL
	1	CHARACTER
	-2	CHARACTER FOR BIT DATA
	12	VARCHAR
	-3	VARCHAR FOR BIT DATA
	40	CLOB
	-95	GRAPHIC
	-96	VARGRAPHIC
	-350	DBCLOB
	30	BLOB
	91	DATE
	92	TIME
	93	TIMESTAMP
	70	DATALINK
	-100	ROWID
	17	DISTINCT
TYPE_NAME	VARCHAR(260)	The name of the data type of the column.
COLUMN_SIZE	INTEGER	The length of the column.
BUFFER_LENGTH	INTEGER	Indicates the length of the column in a buffer.
DECIMAL_DIGITS	SMALLINT Nullable	Indicates the number of digits for a numeric column. Contains the null value if the column is not numeric.
PSEUDO_COLUMN	SMALLINT	Indicates whether this is a ROWID or identity column.
	1	The column is not a ROWID or identity column.
	2	The column is a ROWID or identity column.
TABLE_CAT	VARCHAR(128)	Relational database name

| Table 129. SQLSPECIALCOLUMNS view (continued)

Column Name	Data Type	Description
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema that contains the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
NULLABLE	SMALLINT	Indicates whether the column can contain the null value.
		0 The column is not nullable.
		1 The column is nullable.
JDBC_DATA_TYPE	SMALLINT	Indicates the JDBC data type of the column.
		-5 BIGINT
		4 INTEGER
		5 SMALLINT
		3 DECIMAL
		2 NUMERIC
		8 DOUBLE PRECISION
		7 REAL
		1 CHARACTER
		-2 CHARACTER FOR BIT DATA
		12 VARCHAR
		-3 VARCHAR FOR BIT DATA
		2005 CLOB
		1 GRAPHIC
		12 VARGRAPHIC
		1111 DBCLOB
		2004 BLOB
		91 DATE
		92 TIME
93 TIMESTAMP		
70 DATALINK		
1111 ROWID		
2001 DISTINCT		

SQLSTATISTICS

SQLSTATISTICS

The SQLSTATISTICS view contains statistic information on a table. The following table describes the columns in the view:

Table 130. SQLSTATISTICS view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema of the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
NON_UNIQUE	SMALLINT Nullable	Indicates whether an index prohibits duplicate keys on the table. Contains the null value if the TYPE is 0.
INDEX_QUALIFIER	VARCHAR(128) Nullable	Name of the schema of the index. Contains the null value if the TYPE is 0.
INDEX_NAME	VARCHAR(128) Nullable	Name of the index. Contains the null value if the TYPE is 0.
TYPE	SMALLINT	Indicates the type of information returned: 0 The number of rows in the table. 3 An index on the table.
ORDINAL_POSITION	SMALLINT Nullable	Indicates the ordinal position of the key in the index. Contains the null value if the TYPE is 0.
COLUMN_NAME	VARCHAR(128) Nullable	Name of the column for a key in the index. Contains the null value if the TYPE is 0.
ASC_OR_DESC	CHAR(1) Nullable	Order of the column in the key: A Ascending D Descending Contains the null value if the TYPE is 0.
CARDINALITY	INTEGER Nullable	Reserved. Contains the null value.
PAGES	INTEGER Nullable	Reserved. Contains the null value.
FILTER_CONDITION	VARCHAR(128) Nullable	Indicates whether the index is a select/omit index. empty-string This is a select/omit index. Contains the null value if the TYPE is 0 or this is not a select/omit index.

SQLTABLEPRIVILEGES

The SQLTABLEPRIVILEGES view contains one row for every privilege granted on a table. The following table describes the columns in the view:

Table 131. SQLTABLEPRIVILEGES view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the SQL schema of the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
GRANTOR	VARCHAR(128) Nullable	Reserved. Contains the null value.
GRANTEE	VARCHAR(128)	The user profile to which the privilege is granted.
PRIVILEGE	VARCHAR(10)	The privilege granted: ALTER The privilege to alter the table. DELETE The privilege to delete rows from the table. INDEX The privilege to create an index on the table. INSERT The privilege to insert rows into the table. REFERENCES The privilege to reference the table in a referential constraint. SELECT The privilege to select rows from the table. UPDATE The privilege to update the table.
IS_GRANTABLE	VARCHAR(3)	Indicates whether the privilege is grantable to other users. NO The privilege is not grantable. YES The privilege is grantable.
DBNAME	VARCHAR(8) Nullable	Reserved. Contains the null value.

SQLTABLES

SQLTABLES

The SQLTABLES view contains one row for every table, view, and alias. The following table describes the columns in the view:

Table 132. SQLTABLES view

Column Name	Data Type	Description
TABLE_CAT	VARCHAR(128)	Relational database name
TABLE_SCHEM	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	VARCHAR(128)	Name of the table.
TABLE_TYPE	VARCHAR(10)	Indicates the type of the table: ALIAS The table is an alias. TABLE The table is an SQL table or physical file. VIEW The table is an SQL view or logical file.
REMARKS	VARCHAR(128) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no long comment.
TYPE_CAT	VARCHAR(128) Nullable	Reserved. Contains the null value.
TYPE_SCHEM	VARCHAR(128) Nullable	Reserved. Contains the null value.
TYPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
SELF_REF_COL_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
REF_GENERATION	VARCHAR(128) Nullable	Reserved. Contains the null value.
DBNAME	VARCHAR(8) Nullable	Reserved. Contains the null value.
TABLE_TEXT	VARCHAR(50)	A character string provided with the LABEL statement.

SQLTYPEINFO

The SQLTYPEINFO view contains one row for every built-in data type. The following table describes the columns in the view:

Table 133. SQLTYPEINFO view

Column Name	Data Type	Description
TYPE_NAME	VARCHAR(128)	Name of the built-in data type:
		BIGINT BIGINT
		INTEger INTEGER
		SMALLINT SMALLINT
		DECIMAL DECIMAL
		NUMERIC NUMERIC
		FLOAT DOUBLE PRECISION
		REAL REAL
		CHARacter CHARACTER
		CHARacter FOR BIT DATA CHARACTER FOR BIT DATA
		VARCHAR VARCHAR
		VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA
		CLOB CLOB
		GRAPHIC GRAPHIC
		VARGRAPHIC VARGRAPHIC
		DBCLOB DBCLOB
		BLOB BLOB
		DATE DATE
		TIME TIME
		TIMESTAMP TIMESTAMP
		DATALINK DATALINK
		ROWID ROWID

SQLTYPEINFO

Table 133. SQLTYPEINFO view (continued)

Column Name	Data Type	Description
DATA_TYPE	SMALLINT	The data type of the column:
		-5 BIGINT
		4 INTEGER
		5 SMALLINT
		3 DECIMAL
		2 NUMERIC
		8 DOUBLE PRECISION
		7 REAL
		1 CHARACTER
		-2 CHARACTER FOR BIT DATA
		12 VARCHAR
		-3 VARCHAR FOR BIT DATA
		40 CLOB
		-95 GRAPHIC
		-96 VARGRAPHIC
		-350 DBCLOB
		30 BLOB
		9 DATE
		10 TIME
		11 TIMESTAMP
		70 DATALINK
		-100 ROWID
COLUMN_SIZE	INTEGER	The maximum length of the data type.
LITERAL_PREFIX	VARCHAR(128) Nullable	Indicates the prefix for a string literal. Contains the null value if the data type is not a string.
LITERAL_SUFFIX	VARCHAR(128) Nullable	Indicates the suffix for a string literal. Contains the null value if the data type is not a string.
CREATE_PARAMS	VARCHAR(128) Nullable	Indicates the parameters supported with the data type. LENGTH The parameter is a length. Returned for all string data types and DATALINK. PRECISION,SCALE The parameters include precision and scale. Returned for the DECIMAL and NUMERIC data types. Contains the null value for all other data types.
NULLABLE	SMALLINT	Indicates whether the data type is nullable.
		0 The data type does not allow nulls.
		1 The data type does allow nulls.

Table 133. SQLTYPEINFO view (continued)

Column Name	Data Type	Description
CASE_SENSITIVE	SMALLINT	Indicates whether the data type is case sensitive.
		0 The data type is not case sensitive.
		1 The data type is case sensitive.
SEARCHABLE	SMALLINT	Indicates whether the data type can be used in a predicate.
		0 The data type is cannot be used in predicates.
		2 The data type can be used in all predicates except the LIKE predicate.
		3 The data type can be used in all predicates including the LIKE predicate.
UNSIGNED_ATTRIBUTE	SMALLINT Nullable	Indicates whether the numeric data type is signed or unsigned.
		0 The data type is signed.
		1 The data type is unsigned.
		Contains the null value if the data type is not numeric.
FIXED_PREC_SCALE	SMALLINT	Indicates whether the data type has a fixed precision and scale.
		0 The data type does not have a fixed precision and scale.
		1 The data type does have a fixed precision and scale.
AUTO_UNIQUE_VALUE	SMALLINT Nullable	Indicates whether the numeric data type is auto-incrementing:
		0 The data type is not auto-incrementing.
		1 The data type is auto-incrementing.
		Contains the null value if the data type is not numeric.
LOCAL_TYPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
MINIMUM_SCALE	SMALLINT Nullable	Indicates the minimum scale of numeric data types.
		Contains the null value if the data type is not numeric.
MAXIMUM_SCALE	SMALLINT Nullable	Indicates the maximum scale of numeric data types.
		Contains the null value if the data type is not numeric.

SQLTYPEINFO

Table 133. SQLTYPEINFO view (continued)

Column Name	Data Type	Description	
SQL_DATA_TYPE	SMALLINT	Indicates the SQL data type value of the data type:	
		-5	BIGINT
		4	INTEGER
		5	SMALLINT
		3	DECIMAL
		2	NUMERIC
		8	DOUBLE PRECISION
		7	REAL
		1	CHARACTER
		-2	CHARACTER FOR BIT DATA
		12	VARCHAR
		-3	VARCHAR FOR BIT DATA
		-99	CLOB
		-95	GRAPHIC
		-96	VARGRAPHIC
		-350	DBCLOB
		-98	BLOB
		9	DATE
		10	TIME
11	TIMESTAMP		
70	DATALINK		
-100	ROWID		
SQL_DATETIME_SUB	SMALLINT Nullable	The datetime subtype of the data type:	
		1	DATE
		2	TIME
		3	TIMESTAMP
Contains the null value if the data type is not a datetime data type.			
NUM_PREC_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits:	
		2	Binary; floating-point precision is specified in binary digits.
		10	Decimal; all other numeric types are specified in decimal digits.
Contains the null value if the parameter is not numeric.			
INTERVAL_PRECISION	SMALLINT Nullable	Reserved. Contains the null value.	

| Table 133. SQLTYPEINFO view (continued)

Column Name	Data Type	Description
JDBC_DATA_TYPE	SMALLINT	The JDBC data type value of the data type:
	-5	BIGINT
	4	INTEGER
	5	SMALLINT
	3	DECIMAL
	2	NUMERIC
	8	DOUBLE PRECISION
	7	REAL
	1	CHARACTER
	-2	CHARACTER FOR BIT DATA
	12	VARCHAR
	-3	VARCHAR FOR BIT DATA
	2005	CLOB
	1	GRAPHIC
	12	VARGRAPHIC
	1111	DBCLOB
	2004	BLOB
	91	DATE
	92	TIME
	93	TIMESTAMP
	70	DATALINK
	1111	ROWID

SQLUDTS

SQLUDTS

The SQLUDTS view contains one row for every distinct type. The following table describes the columns in the view:

Table 134. SQLUDTS view

Column Name	Data Type	Description
TYPE_CAT	VARCHAR(128)	Relational database name
TYPE_SCHEM	VARCHAR(128)	Name of the schema containing the user-defined type.
TYPE_NAME	VARCHAR(128)	Name of the user-defined type.
CLASS_NAME	VARCHAR(20)	Java class name of the user-defined type.
		java.math.BigInteger BIGINT
		java.lang.Integer INTEGER
		java.lang.Short SMALLINT
		java.math.BigDecimal DECIMAL
		java.sql.BigDecimal NUMERIC
		java.lang.Double DOUBLE PRECISION
		java.lang.Float REAL
		java.lang.String CHARACTER
		byte[] CHARACTER FOR BIT DATA
		java.lang.String VARCHAR
		byte[] VARCHAR FOR BIT DATA
		java.sql.Clob CLOB
		java.lang.String GRAPHIC
		java.lang.String VARGRAPHIC
		java.sql.Clob DBCLOB
		java.sql.Blob BLOB
		java.sql.Date DATE
		java.sql.Time TIME
		java.sql.Timestamp TIMESTAMP
		java.net.URL DATALINK
		byte[] ROWID
DATA_TYPE	SMALLINT	Reserved. Contains 2001.

| Table 134. SQLUDTS view (continued)

Column Name	Data Type	Description
BASE_TYPE	SMALLINT	The source data type of the user-defined data type:
		-5 BIGINT
		4 INTEGER
		5 SMALLINT
		3 DECIMAL
		2 NUMERIC
		8 DOUBLE PRECISION
		7 REAL
		1 CHARACTER
		-2 CHARACTER FOR BIT DATA
		12 VARCHAR
		-3 VARCHAR FOR BIT DATA
		2005 CLOB
		1 GRAPHIC
		12 VARGRAPHIC
		1111 DBCLOB
		2004 BLOB
		91 DATE
		92 TIME
		93 TIMESTAMP
		70 DATALINK
		1111 ROWID
REMARKS	VARCHAR(2000) Nullable	A character string supplied with the COMMENT statement. Contains the null value if there is no comment.

ANS and ISO Catalog Views

There are two versions of some of the ANS and ISO catalog views. The version documented is the normal set of ANS and ISO views. A second set of views have names that are limited to no more than 18 characters and other than the view names are not documented in this book.

The ANS and ISO catalog includes the following tables in the QSYS2 library:

View Name	Shorter View Name	Description
"SQL_FEATURES" on page 998		Information about features supported by the database manager
"SQL_LANGUAGES" on page 999	SQL_LANGUAGES_S	Information about the supported languages
"SQL_SIZING" on page 1000		Information about the limits supported by the database manager

The ANS and ISO catalog includes the following views and tables in the SYSIBM library:

View Name	Shorter View Name	Description
"CHARACTER_SETS" on page 977	CHARACTER_SETS_S	Information about supported CCSIDs
"CHECK_CONSTRAINTS" on page 978		Information about check constraints
"COLUMNS" on page 979	COLUMNS_S	Information about columns
"INFORMATION_SCHEMA_CATALOG_NAME" on page 983	CATALOG_NAME	Information about the relational database
"PARAMETERS" on page 984	PARAMETERS_S	Information about procedure parameters
"REFERENTIAL_CONSTRAINTS" on page 988	REF_CONSTRAINTS	Information about referential constraints
"ROUTINES" on page 989	ROUTINES_S	Information about routines
"SCHEMATA" on page 997	SCHEMATA_S	Statistical information about schemas
"TABLE_CONSTRAINTS" on page 1001		Information about constraints
"TABLES" on page 1002	TABLES_S	Information about tables
"USER_DEFINED_TYPES" on page 1003	UDT_S	Information about distinct types
"VIEWS" on page 1007		Information about views

CHARACTER_SETS

The CHARACTER_SETS view contains one row for every CCSID supported. The following table describes the columns in the view:

Table 135. CHARACTER_SETS view

Column Name	Data Type	Description
CHARACTER_SET_CATALOG	VARCHAR(128)	Relational database name
CHARACTER_SET_SCHEMA	VARCHAR(128)	The schema name of the character set. Contains 'SYSIBM'.
CHARACTER_SET_NAME	VARCHAR(128)	The character set name.
FORM_OF_USE	VARCHAR(128) Nullable	Reserved. Contains the null value.
NUMBER_OF_CHARACTERS	INTEGER Nullable	Reserved. Contains the null value.
DEFAULT_COLLATE_CATALOG	VARCHAR(128)	Reserved. Contains the relational database name.
DEFAULT_COLLATE_SCHEMA	VARCHAR(128)	Reserved. Contains SYSIBM.
DEFAULT_COLLATE_NAME	VARCHAR(128)	Reserved. Contains IBMDEFAULT.

CHECK_CONSTRAINTS

CHECK_CONSTRAINTS

The CHECK_CONSTRAINTS view contains one row for every check constraint.
The following table describes the columns in the view:

Table 136. CHECK_CONSTRAINTS view

Column Name	Data Type	Description
CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name
CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the schema containing the constraint
CONSTRAINT_NAME	VARCHAR(128)	Name of the constraint
CHECK_CLAUSE	VARCHAR(2000) Nullable	Text of the check constraint clause Contains the null value if the check clause cannot be contained in the column without truncation.

COLUMNS

The COLUMNS view contains one row for every column. The following table describes the columns in the view:

Table 137. COLUMNS view

Column Name	Data Type	Description
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema containing the table or view
TABLE_NAME	VARCHAR(128)	Name of the table or view that contains the column
COLUMN_NAME	VARCHAR(128)	Name of the column
ORDINAL_POSITION	INTEGER	Numeric place of the column in the table or view, ordered from left to right
COLUMN_DEFAULT	VARCHAR(2000) Nullable	The default value of a column, if one exists. If the default value of the column cannot be represented without truncation, then the value of the column is the string 'TRUNCATED'. The default value is stored in character form. The following special values also exist: CURRENT_DATE The default value is the current date. CURRENT_TIME The default value is the current time. CURRENT_TIMESTAMP The default value is the current timestamp. NULL The default value is the null value. USER The default value is the current job user. Contains the null value if the column has no default value. For example, if the column has an IDENTITY attribute or is a row ID.
IS_NULLABLE	VARCHAR(3)	Indicates whether the column can contain null values: NO The column cannot contain null values. YES The column can contain null values.

COLUMNS

Table 137. COLUMNS view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128)	Type of column:
		BIGINT Big number
		INTEGER Large number
		SMALLINT Small number
		DECIMAL Packed decimal
		NUMERIC Zoned decimal
		DOUBLE PRECISION Double-precision floating point
		REAL Single-precision floating point
		CHARACTER Fixed-length character string
		CHARACTER VARYING Varying-length character string
		CHARACTER LARGE OBJECT Character large object string
		GRAPHIC Fixed-length graphic string
		GRAPHIC VARYING Varying-length graphic string
		DOUBLE-BYTE CHARACTER LARGE OBJECT Double-byte character large object string
		BINARY LARGE OBJECT Binary large object string
		DATE Date
		TIME Time
		TIMESTAMP Timestamp
		DATALINK Datalink
		ROWID Row ID
		USER-DEFINED Distinct type
CHARACTER_MAXIMUM_LENGTH	INTEGER Nullable	Maximum length of the string for binary, character and graphic string data types. Contains the null value if the column is not a string.
CHARACTER_OCTET_LENGTH	INTEGER Nullable	Number of bytes for binary, character and graphic string data types. Contains the null value if the column is not a string.

Table 137. COLUMNS view (continued)

Column Name	Data Type	Description
NUMERIC_PRECISION	INTEGER Nullable	The precision of all numeric columns. Note: This column supplies the precision of all numeric data types, including single- and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the column is not numeric.
NUMERIC_PRECISION_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits 2 Binary; floating-point precision is specified in binary digits. 10 Decimal; all other numeric types are specified in decimal digits. Contains the null value if the column is not numeric.
NUMERIC_SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if the column is not decimal, numeric, or binary.
DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp. 0 For DATE and TIME data types 6 For TIMESTAMP data types (number of microseconds). Contains the null value if the column is not a date, time, or timestamp.
INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
CHARACTER_SET_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.
CHARACTER_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set. Contains SYSIBM. Contains the null value if the column is not a string.
CHARACTER_SET_NAME	VARCHAR(128) Nullable	The character set name. Contains the null value if the column is not a string.
COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.

COLUMNS

Table 137. COLUMNS view (continued)

Column Name	Data Type	Description
COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation. Contains SYSIBM. Contains the null value if the column is not a string.
COLLATION_NAME	VARCHAR(128) Nullable	The collation name. Contains IBMBINARY. Contains the null value if the column is not a string.
DOMAIN_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
DOMAIN_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
DOMAIN_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
UDT_CATALOG	VARCHAR(128) Nullable	The relational database name if this is a the distinct type. Contains the null value if this is not a distinct type.
UDT_SCHEMA	VARCHAR(128) Nullable	The name of the schema if this is a the distinct type. Contains the null value if this is not a distinct type.
UDT_NAME	VARCHAR(128) Nullable	The name of the distinct type. Contains the null value if this is not a distinct type.
SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
MAXIMUM_CARDINALITY	INTEGER Nullable	Reserved. Contains the null value.
DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the column.
IS_SELF_REFERENCING	VARCHAR(3)	Reserved. Contains 'NO'.

INFORMATION_SCHEMA_CATALOG_NAME

The INFORMATION_SCHEMA_CATALOG_NAME view contains one row for the relational database. The following table describes the columns in the view:

Table 138. INFORMATION_SCHEMA_CATALOG_NAME view

Column Name	Data Type	Description
CATALOG_NAME	VARCHAR(128)	Relational database name

PARAMETERS

PARAMETERS

The PARAMETERS view contains one row for each parameter of a routine in the relational database. The following table describes the columns in the view:

Table 139. PARAMETERS view

Column Name	Data Type	Description
SPECIFIC_CATALOG	VARCHAR(128)	Relational database name
SPECIFIC_SCHEMA	VARCHAR(128)	Schema name of the routine instance
SPECIFIC_NAME	VARCHAR(128)	Specific name of the routine instance
ORDINAL_POSITION	INTEGER	Numeric place of the parameter in the parameter list, ordered from left to right.
PARAMETER_MODE	VARCHAR(5)	The type of the parameter: IN This is an input parameter. OUT This is an output parameter. INOUT This is an input/output parameter.
IS_RESULT	VARCHAR(3)	Reserved. Contains 'NO'.
AS_LOCATOR	VARCHAR(3)	Indicates whether the parameter was specified as a locator. NO The parameter was not specified as a locator. YES The parameter was specified as a locator.
PARAMETER_NAME	VARCHAR(128) Nullable	The name of the parameter Contains the null value if the parameter does not have a name.
FROM_SQL_SPECIFIC_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
FROM_SQL_SPECIFIC_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
FROM_SQL_SPECIFIC_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.

Table 139. PARAMETERS view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128) Nullable	Type of the parameter: BIGINT Big number INTEGER Large number SMALLINT Small number DECIMAL Packed decimal NUMERIC Zoned decimal DOUBLE PRECISION Floating point; DOUBLE PRECISION REAL Floating point; REAL CHARACTER Fixed-length character string CHARACTER VARYING Varying-length character string CHARACTER LARGE OBJECT Character large object string GRAPHIC Fixed-length graphic string GRAPHIC VARYING Varying-length graphic string DOUBLE-BYTE CHARACTER LARGE OBJECT Double-byte character large object string BINARY LARGE OBJECT Binary large object string DATE Date TIME Time TIMESTAMP Timestamp DATALINK Datalink ROWID Row ID USER-DEFINED Distinct Type
CHARACTER_MAXIMUM_LENGTH	INTEGER Nullable	Maximum length of the string for binary, character, and graphic string data types. Contains the null value if the parameter is not a string.
CHARACTER_OCTET_LENGTH	INTEGER Nullable	Number of bytes for binary, character, and graphic string data types. Contains the null value if the parameter is not a string.
CHARACTER_SET_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.

PARAMETERS

Table 139. PARAMETERS view (continued)

Column Name	Data Type	Description				
CHARACTER_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set. Contains 'SYSIBM'. Contains the null value if the column is not a string.				
CHARACTER_SET_NAME	VARCHAR(128) Nullable	The character set name. Contains the null value if the column is not a string.				
COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the column is not a string.				
COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation. SYSIBM is returned. Contains the null value if the column is not a string.				
COLLATION_NAME	VARCHAR(128) Nullable	The collation name. IBM_BINARY is returned. Contains the null value if the column is not a string.				
NUMERIC_PRECISION	INTEGER Nullable	The precision of all numeric parameters. Note: This column supplies the precision of all numeric data types, including single- and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the parameter is not numeric.				
NUMERIC_PRECISION_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: <table border="0"> <tr> <td style="padding-right: 20px;">2</td> <td>Binary; floating-point precision is specified in binary digits.</td> </tr> <tr> <td>10</td> <td>Decimal; all other numeric types are specified in decimal digits.</td> </tr> </table> Contains the null value if the parameter is not numeric.	2	Binary; floating-point precision is specified in binary digits.	10	Decimal; all other numeric types are specified in decimal digits.
2	Binary; floating-point precision is specified in binary digits.					
10	Decimal; all other numeric types are specified in decimal digits.					
NUMERIC_SCALE	INTEGER Nullable	Scale of numeric data. Contains the null value if not decimal, numeric, or binary parameter.				
DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp. <table border="0"> <tr> <td style="padding-right: 20px;">0</td> <td>For DATE and TIME data types</td> </tr> <tr> <td>6</td> <td>For TIMESTAMP data types (number of microseconds).</td> </tr> </table> Contains the null value if the parameter is not a date, time, or timestamp.	0	For DATE and TIME data types	6	For TIMESTAMP data types (number of microseconds).
0	For DATE and TIME data types					
6	For TIMESTAMP data types (number of microseconds).					

| Table 139. PARAMETERS view (continued)

Column Name	Data Type	Description
INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
UDT_CATALOG	VARCHAR(128) Nullable	The relational database name if this is a distinct type. Contains the null value if this is not a distinct type.
UDT_SCHEMA	VARCHAR(128) Nullable	The name of the schema if this is a distinct type. Contains the null value if this is not a distinct type.
UDT_NAME	VARCHAR(128) Nullable	The name of the distinct type. Contains the null value if this is not a distinct type.
SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
MAXIMUM_CARDINALITY	INTEGER Nullable	Reserved. Contains the null value.
DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the parameter.

REFERENTIAL_CONSTRAINTS

REFERENTIAL_CONSTRAINTS

The REFERENTIAL_CONSTRAINTS view contains one row for each referential constraint. The following table describes the columns in the view:

Table 140. REFERENTIAL_CONSTRAINTS view

Column Name	Data Type	Description
CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name
CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	VARCHAR(128)	Name of the constraint.
UNIQUE_CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name containing the unique constraint referenced by the referential constraint.
UNIQUE_CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the SQL schema containing the unique constraint referenced by the referential constraint.
UNIQUE_CONSTRAINT_NAME	VARCHAR(128)	Name of the unique constraint referenced by the referential constraint.
MATCH_OPTION	VARCHAR(7)	Reserved. Contains 'NONE'.
UPDATE_RULE	VARCHAR(11)	Update Rule. <ul style="list-style-type: none">• NO ACTION• RESTRICT
DELETE_RULE	VARCHAR(11)	Delete Rule <ul style="list-style-type: none">• NO ACTION• CASCADE• SET NULL• SET DEFAULT• RESTRICT

ROUTINES

The ROUTINES view contains one row for each routine. The following table describes the columns in the view:

Table 141. ROUTINES view

Column Name	Data Type	Description
SPECIFIC_CATALOG	VARCHAR(128)	Relational database name
SPECIFIC_SCHEMA	VARCHAR(128)	Schema name of the routine instance.
SPECIFIC_NAME	VARCHAR(128)	Specific name of the routine.
ROUTINE_CATALOG	VARCHAR(128)	Relational database name
ROUTINE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the routine.
ROUTINE_NAME	VARCHAR(128)	Name of the routine.
ROUTINE_TYPE	VARCHAR(15)	Type of the routine. PROCEDURE This is a procedure. FUNCTION This is a function. INSTANCE METHOD This is a built-in data type function created for a distinct type.
MODULE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
MODULE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
MODULE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
UDT_CATALOG	VARCHAR(128) Nullable	Relational database name. Contains the null value if this is not an INSTANCE METHOD.
UDT_SCHEMA	VARCHAR(128) Nullable	Name of the SQL schema that contains the distinct type related to this function. Contains the null value if this is not an INSTANCE METHOD.
UDT_NAME	VARCHAR(128) Nullable	Name of the distinct type name related to this function. Contains the null value if this is not an INSTANCE METHOD.

ROUTINES

Table 141. ROUTINES view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128) Nullable	Type of the result of the function: BIGINT Big number INTEGER Large number SMALLINT Small number DECIMAL Packed decimal NUMERIC Zoned decimal DOUBLE PRECISION Floating point; DOUBLE PRECISION REAL Floating point; REAL CHARACTER Fixed-length character string CHARACTER VARYING Varying-length character string CHARACTER LARGE OBJECT Character large object string GRAPHIC Fixed-length graphic string GRAPHIC VARYING Varying-length graphic string DOUBLE-BYTE CHARACTER LARGE OBJECT Double-byte character large object string BINARY LARGE OBJECT Binary large object string DATE Date TIME Time TIMESTAMP Timestamp DATALINK Datalink ROWID Row ID USER-DEFINED Distinct Type Contains the null value if this is not a scalar function.
CHARACTER_MAXIMUM_LENGTH	INTEGER Nullable	Maximum length of the result string of the function for binary, character, and graphic string data types. Contains the null value if this is not a scalar function or the parameter is not a string.
CHARACTER_OCTET_LENGTH	INTEGER Nullable	Number of bytes for the result string of the function for binary, character, and graphic string data types. Contains the null value if this is not a scalar function or the parameter is not a string.

Table 141. ROUTINES view (continued)

Column Name	Data Type	Description
CHARACTER_SET_CATALOG	VARCHAR(128) Nullable	Relational database name of the result of the function. Contains the null value if this is not a scalar function or the result is not a string.
CHARACTER_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set of the result of the function. Contains 'SYSIBM'. Contains the null value if this is not a scalar function or the result is not a string.
CHARACTER_SET_NAME	VARCHAR(128) Nullable	The character set name of the result of the function. Contains the null value if this is not a scalar function or the result is not a string.
COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name of the result of the function. Contains the null value if this is not a scalar function or the result is not a string.
COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation of the result of the function. SYSIBM is returned. Contains the null value if this is not a scalar function or the result is not a string.
COLLATION_NAME	VARCHAR(128) Nullable	The collation name of the result of the function. IBM_BINARY is returned. Contains the null value if this is not a scalar function or the result is not a string.
NUMERIC_PRECISION	INTEGER Nullable	The precision of the result of the function. Note: This column supplies the precision of all numeric data types, including single- and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if this is not a scalar function or the result is not numeric.
NUMERIC_PRECISION_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: 2 Binary; floating-point precision is specified in binary digits. 10 Decimal; all other numeric types are specified in decimal digits. Contains the null value if this is not a scalar function or the result is not numeric.
NUMERIC_SCALE	INTEGER Nullable	Scale of numeric result of the function. Contains the null value if this is not a scalar function or the result is not numeric.

ROUTINES

Table 141. ROUTINES view (continued)

Column Name	Data Type	Description
DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp result of the function. 0 For DATE and TIME data types 6 For TIMESTAMP data types (number of microseconds). Contains the null value if this is not a scalar function or the result is not a date, time, or timestamp.
INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
TYPE_UDT_CATALOG	VARCHAR(128) Nullable	The relational database name if the result of the function is a distinct type. Contains the null value if this is not a scalar function or the result is not a distinct type.
TYPE_UDT_SCHEMA	VARCHAR(128) Nullable	The name of the schema if the result of the function is a distinct type. Contains the null value if this is not a scalar function or the result is not a distinct type.
TYPE_UDT_NAME	VARCHAR(128) Nullable	The name of the distinct type if the result of the function is a distinct type. Contains the null value if this is not a scalar function or the result is not a distinct type.
SCOPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
SCOPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
MAXIMUM_CARDINALITY	INTEGER Nullable	Reserved. Contains the null value.
DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the result of the function.
ROUTINE_BODY	VARCHAR(8)	The type of the routine body: EXTERNAL This is an external routine. SQL This is an SQL routine.
ROUTINE_DEFINITION	DBCLOB Nullable	If this is an SQL routine, this column contains the SQL routine body. Contains the null value if this is not an SQL routine or if the routine body cannot be contained in this column without truncation.

Table 141. ROUTINES view (continued)

Column Name	Data Type	Description
EXTERNAL_NAME	VARCHAR(279) Nullable	<p>If this is an external routine, this column identifies the external program name.</p> <ul style="list-style-type: none"> For REXX, the external program name is <i>schema-name/source-file-name(member-name)</i>. For ILE service programs, the external program name is <i>schema-name/service-program-name(entry-point-name)</i>. For Java programs, the external program name is an optional jar-id followed by a <i>fully-qualified-class-name!method-name</i> or <i>fully-qualified-class-name.method-name</i>. For all other languages, the external program name is <i>schema-name/program-name</i>. <p>Contains the null value if this is not an external routine.</p>
EXTERNAL_LANGUAGE	VARCHAR(8) Nullable	<p>If this is an external routine, this column identifies the external program name.</p> <p>C The external program is written in C.</p> <p>C++ The external program is written in C++.</p> <p>CL The external program is written in CL.</p> <p>COBOL The external program is written in COBOL.</p> <p>COBOLLE The external program is written in ILE COBOL.</p> <p>FORTRAN The external program is written in FORTRAN.</p> <p>JAVA The external program is written in JAVA.</p> <p>PLI The external program is written in PL/I.</p> <p>REXX The external program is a REXX procedure.</p> <p>RPG The external program is written in RPG.</p> <p>RPGLE The external program is written in ILE RPG.</p> <p>Contains the null value if this is not an external routine.</p>

ROUTINES

Table 141. ROUTINES view (continued)

Column Name	Data Type	Description
PARAMETER_STYLE	VARCHAR(18) Nullable	If this is an external routine, this column identifies the parameter style (calling convention).
		DB2GENERAL This is the DB2GENERAL calling convention.
		DB2SQL This is the DB2SQL calling convention.
		GENERAL This is the GENERAL calling convention.
		JAVA This is the JAVA calling convention.
		GENERAL WITH NULLS This is the GENERAL WITH NULLS calling convention.
		SQL This is the SQL standard calling convention.
		Contains the null value if this is not an external routine.
IS_DETERMINISTIC	VARCHAR(3)	This column identifies whether the routine is deterministic. That is, whether a call to the routine with the same arguments will always return the same result.
		NO The routine is not deterministic.
		YES The routine is deterministic.
SQL_DATA_ACCESS	VARCHAR(17)	This column identifies whether a routine contains SQL and whether it reads or modifies data.
		NO SQL The routine does not contain any SQL statements.
		CONTAINS SQL The routine contains SQL statements.
		READS SQL DATA The routine possibly reads data from a table or view.
		MODIFIES SQL DATA The routine possibly modifies data in a table or view or issues SQL DDL statements.

Table 141. ROUTINES view (continued)

Column Name	Data Type	Description
IS_NULL_CALL	VARCHAR(3) Nullable	Identifies whether the function needs to be called if an input parameter is the null value. NO This function need not be called if an input parameter is the null value. If this is a scalar function, the result of the function is implicitly null if any of the operands are null. If this is a table function, the result of the function is an empty table if any of the operands are the null value. YES This function must be called even if an input operand is null. Contains the null value if this is not a function.
SQL_PATH	VARCHAR(3483) Nullable	If this is an SQL routine, this column identifies the path. Contains the null value if this is not an SQL routine.
SCHEMA_LEVEL_ROUTINE	VARCHAR(3)	Reserved. Contains 'YES'.
MAX_DYNAMIC_RESULT_SETS	SMALLINT	Identifies the maximum number of result sets returned. 0 indicates that there are no result sets.
IS_USER_DEFINED_CAST	VARCHAR(3) Nullable	Identifies whether the this function is a cast function created when a distinct type was created. NO This function is not a cast function. YES This function is a cast function. Contains the null value if the routine is not a function.
IS_IMPLICITLY_INVOCABLE	VARCHAR(3) Nullable	Identifies whether the this function is a cast function created when a distinct type was created and can be implicitly invoked. NO This function is not a cast function. YES This function is a cast function and can be implicitly invoked. Contains the null value if the routine is not a function.
SECURITY_TYPE	VARCHAR(22) Nullable	Reserved. Contains 'IMPLEMENTATION DEFINED' if this is an external routine. Contains the null value if the routine is not an external routine.
TO_SQL_SPECIFIC_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
TO_SQL_SPECIFIC_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.

ROUTINES

| *Table 141. ROUTINES view (continued)*

Column Name	Data Type	Description
AS_LOCATOR	VARCHAR(3) Nullable	Indicates whether the result was specified as a locator. NO The parameter was not specified as a locator. YES The parameter was specified as a locator. Contains the null value if this is not a scalar function.
CREATED	TIMESTAMP	Identifies the timestamp when the routine was created.
LAST_ALTERED	TIMESTAMP	Reserved. Contains 'CREATED'.

SCHEMATA

The SCHEMATA view contains one row for each schema. The following table describes the columns in the view:

Table 142. SCHEMATA view

Column Name	Data Type	Description
CATALOG_NAME	VARCHAR(128)	Relational database name
SCHEMA_NAME	VARCHAR(128)	Name of the schema
SCHEMA_OWNER	VARCHAR(128)	Owner of the schema
DEFAULT_CHARACTER_SET_CATALOG	VARCHAR(128)	Relational database name
DEFAULT_CHARACTER_SET_SCHEMA	VARCHAR(128)	The schema name of the default character set. Contains 'SYSIBM'.
DEFAULT_CHARACTER_SET_NAME	VARCHAR(128)	The default character set name.
SQL_PATH	VARCHAR(3483) Nullable	Reserved. Contains the null value.

SQL_FEATURES

SQL_FEATURES

The SQL_FEATURES view contains one row for each feature supported by the database manager. The following table describes the columns in the view:

Table 143. SQL_FEATURES view

Column Name	Data Type	Description
FEATURE_ID	VARCHAR(7)	ANS and ISO feature ID
FEATURE_NAME	VARCHAR(128)	The name of the ANS and ISO feature.
SUB_FEATURE_ID	VARCHAR(7)	ANS and ISO subfeature ID
SUB_FEATURE_NAME	VARCHAR(256)	The name of the ANS and ISO subfeature.
IS_SUPPORTED	VARCHAR(3)	Indicates whether the feature is supported: YES This feature is supported. NO This feature is not supported.
IS_VERIFIED_BY	VARCHAR(128) Nullable	Reserved. Contains the null value.
COMMENTS	VARCHAR(2000) Nullable	Reserved. Contains the null value.

SQL_LANGUAGES

The SQL_LANGUAGES (system name SYSLANGS) table contains one row for every SQL language binding and programming language for which conformance is claimed. The following table describes the columns in the SQL_LANGUAGES view:

Table 144. SQL_LANGUAGES view

Column Name	Data Type	Description
SQL_LANGUAGE_SOURCE	VARCHAR(254)	Name of the standard.
SQL_LANGUAGE_YEAR	VARCHAR(254)	Year in which the standard was approved.
SQL_LANGUAGE_CONFORMANCE	VARCHAR(254) Nullable	Level of conformance. 2 For the 1987 and 1989 standards, indicates that Level 2 conformance is claimed. ENTRY For the 1992 standard, indicates that Entry Level conformance is claimed. CORE For the 1999 standard, indicates that Core Level is conformance is claimed. Contains the null value if conformance is not yet claimed.
SQL_LANGUAGE_INTEGRITY	VARCHAR(254) Nullable	Support of the integrity feature. YES conformance is claimed to the integrity feature NO conformance is not claimed to the integrity feature Contains the null value if the standard does not have a separate integrity feature.
SQL_LANGUAGE_IMPLEMENTATION	VARCHAR(254) Nullable	Reserved. Contains the null value.
SQL_LANGUAGE_BINDING_STYLE	VARCHAR(254)	The style of binding of the SQL language EMBEDDED support for embedded SQL for the language in SQL_LANGUAGE_PROGRAMMING_LANG DIRECT DIRECT SQL is supported (for example Interactive SQL) CLI Support for CLI for the language in SQL_LANGUAGE_PROGRAMMING_LANG
SQL_LANGUAGE_PROGRAMMING_LANG	VARCHAR(254) Nullable	The language supported by EMBEDDED or CLI. C The C language is supported. COBOL The COBOL language is supported. PLI The PL/I language is supported. Contains the null value if the SQL_LANGUAGE_BINDING_STYLE is DIRECT.

SQL_SIZING

SQL_SIZING

The SQL_SIZING view contains one row for each limit supported by the database manager. The following table describes the columns in the view:

Table 145. SQL_SIZING view

Column Name	Data Type	Description
SIZING_ID	INTEGER	ANS and ISO sizing ID
SIZING_NAME	VARCHAR(128)	Name of the ANS and ISO sizing.
SUPPORTED_VALUE	INTEGER Nullable	Indicates the sizing limit. Contains the null value if the sizing limit is not applicable.
COMMENTS	VARCHAR(2000) Nullable	Reserved. Contains the null value.

TABLE_CONSTRAINTS

The TABLE_CONSTRAINTS view contains one row for each constraint. The following table describes the columns in the view:

Table 146. TABLE_CONSTRAINTS view

Column Name	Data Type	Description
CONSTRAINT_CATALOG	VARCHAR(128)	Relational database name
CONSTRAINT_SCHEMA	VARCHAR(128)	Name of the schema containing the constraint.
CONSTRAINT_NAME	VARCHAR(128)	Name of the constraint.
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the schema containing the table.
TABLE_NAME	VARCHAR(128)	Name of the table which the constraint is created over.
CONSTRAINT_TYPE	VARCHAR(11)	Constraint Type CHECK UNIQUE PRIMARY KEY FOREIGN KEY
IS_DEFERRABLE	VARCHAR(3)	Indicates whether the constraint checking can be deferred. Contains 'NO'.
INITIALLY_DEFERRED	VARCHAR(3)	Indicates whether the constraint was defined as initially deferred. Contains 'NO'.

TABLES

TABLES

The TABLES view contains one row for each table, view, and alias. The following table describes the columns in the view:

Table 147. TABLES view

Column Name	Data Type	Description
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the table, view or alias.
TABLE_NAME	VARCHAR(128)	Name of the table, view or alias.
TABLE_TYPE	VARCHAR(10)	Indicates the type of the table: ALIAS The table is an alias. BASE_TABLE The table is an SQL table or physical file. VIEW The table is an SQL view or logical file.
SELF_REFERENCING_COLUMN_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
REFERENCE_GENERATION	VARCHAR(128) Nullable	Reserved. Contains the null value.
USER_DEFINED_TYPE_CATALOG	VARCHAR(128) Nullable	Reserved. Contains the null value.
USER_DEFINED_TYPE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains the null value.
USER_DEFINED_TYPE_NAME	VARCHAR(128) Nullable	Reserved. Contains the null value.
IS_INSERTABLE_INTO	VARCHAR(3)	Identifies whether an INSERT is allowed on the table. NO An INSERT is not allowed on this table. YES An INSERT is allowed on this table.

USER_DEFINED_TYPES

The USER_DEFINED_TYPES view contains one row for each distinct type.⁸⁷ The following table describes the columns in the view:

Table 148. USER_DEFINED_TYPES view

Column Name	Data Type	Description
USER_DEFINED_TYPE_CATALOG	VARCHAR(128)	Relational database name
USER_DEFINED_TYPE_SCHEMA	VARCHAR(128)	Schema name of the distinct type.
USER_DEFINED_TYPE_NAME	VARCHAR(128)	Name of the user that created the distinct type.
USER_DEFINED_TYPE_CATEGORY	VARCHAR(128)	Indicates the type of user-defined type. Contains 'DISTINCT'.
IS_INSTANTIABLE	VARCHAR(3)	Reserved. Contains 'YES'.
IS_FINAL	VARCHAR(3)	Reserved. Contains 'YES'.
ORDERING_FORM	VARCHAR(4)	Indicates what kind of predicates are allowed when this distinct type is a comparand: FULL All predicates are allowed. NONE No predicates are allowed
ORDERING_CATEGORY	VARCHAR(8)	Reserved. Contains 'MAP'.
ORDERING_ROUTINE_CATALOG	VARCHAR(128) Nullable	Relational database name Contains the null value if the ORDERING_FORM is 'NONE'.
ORDERING_ROUTINE_SCHEMA	VARCHAR(128) Nullable	Reserved. Contains 'SYSIBM'. Contains the null value if the ORDERING_FORM is 'NONE'.
ORDERING_ROUTINE_NAME	VARCHAR(128) Nullable	Reserved. Contains a data type name. Contains the null value if the ORDERING_FORM is 'NONE'.
REFERENCE_TYPE	VARCHAR(16) Nullable	Reserved. Contains the null value.

⁸⁷. This view does not contain information about built-in data types.

USER_DEFINED_TYPES

Table 148. USER_DEFINED_TYPES view (continued)

Column Name	Data Type	Description
DATA_TYPE	VARCHAR(128) Nullable	Source data type of the distinct type: BIGINT Big number INTEGER Large number SMALLINT Small number DECIMAL Packed decimal NUMERIC Zoned decimal DOUBLE PRECISION Floating point; DOUBLE PRECISION REAL Floating point; REAL CHARACTER Fixed-length character string CHARACTER VARYING Varying-length character string CHARACTER LARGE OBJECT Character large object string GRAPHIC Fixed-length graphic string GRAPHIC VARYING Varying-length graphic string DOUBLE-BYTE CHARACTER LARGE OBJECT Double-byte character large object string BINARY LARGE OBJECT Binary large object string DATE Date TIME Time TIMESTAMP Timestamp DATALINK Datalink ROWID Row ID USER-DEFINED Distinct Type
CHARACTER_MAXIMUM_LENGTH	INTEGER Nullable	Maximum length of the distinct type for binary, character, and graphic string data types. Contains the null value if the distinct type is not a string.
CHARACTER_OCTET_LENGTH	INTEGER Nullable	Number of bytes of the distinct type for binary, character, and graphic string data types. Contains the null value if the distinct type is not a string.
CHARACTER_SET_CATALOG	VARCHAR(128) Nullable	Relational database name of the distinct type. Contains the null value if the distinct type is not a string.

Table 148. USER_DEFINED_TYPES view (continued)

Column Name	Data Type	Description
CHARACTER_SET_SCHEMA	VARCHAR(128) Nullable	The schema name of the character set of the distinct type. Contains 'SYSIBM'. Contains the null value if the distinct type is not a string.
CHARACTER_SET_NAME	VARCHAR(128) Nullable	The character set name of the distinct type. Contains the null value if the distinct type is not a string.
COLLATION_CATALOG	VARCHAR(128) Nullable	Relational database name of the distinct type. Contains the null value if the distinct type is not a string.
COLLATION_SCHEMA	VARCHAR(128) Nullable	The schema of the collation of the distinct type. SYSIBM is returned. Contains the null value if the distinct type is not a string.
COLLATION_NAME	VARCHAR(128) Nullable	The collation name of the distinct type. IBMBINARY is returned. Contains the null value if the distinct type is not a string.
NUMERIC_PRECISION	INTEGER Nullable	The precision of the distinct type. Note: This column supplies the precision of all numeric data types, including single-and double-precision floating point. The NUMERIC_PRECISION_RADIX column indicates if the value in this column is in binary or decimal digits. Contains the null value if the distinct type is not numeric.
NUMERIC_PRECISION_RADIX	INTEGER Nullable	Indicates if the precision specified in column NUMERIC_PRECISION is specified as a number of binary or decimal digits: 2 Binary; floating-point precision is specified in binary digits. 10 Decimal; all other numeric types are specified in decimal digits. Contains the null value if the distinct type is not numeric.
NUMERIC_SCALE	INTEGER Nullable	Scale of numeric distinct type. Contains the null value if the distinct type is not decimal, numeric, or binary.

USER_DEFINED_TYPES

| Table 148. USER_DEFINED_TYPES view (continued)

Column Name	Data Type	Description
DATETIME_PRECISION	INTEGER Nullable	The fractional part of a date, time, or timestamp distinct type. 0 For DATE and TIME data types 6 For TIMESTAMP data types (number of microseconds). Contains the null value if the distinct type is not date, time, or timestamp.
INTERVAL_TYPE	VARCHAR(128) Nullable	Reserved. Contains the null value.
INTERVAL_PRECISION	INTEGER Nullable	Reserved. Contains the null value.
SOURCE_DTD_IDENTIFIER	VARCHAR(128) Nullable	A unique internal identifier for the source data type. Contains the null value if the distinct type is not sourced on another distinct type.
REF_DTD_IDENTIFIER	VARCHAR(256) Nullable	Reserved. Contains the null value.

VIEWS

The VIEWS view contains one row for each view. The following table describes the columns in the view:

Table 149. VIEWS view

Column Name	Data Type	Description
TABLE_CATALOG	VARCHAR(128)	Relational database name
TABLE_SCHEMA	VARCHAR(128)	Name of the SQL schema that contains the view.
TABLE_NAME	VARCHAR(128)	Name of the view.
VIEW_DEFINITION	VARCHAR(10000) Nullable	The query expression portion of the CREATE VIEW statement. Contains the null value if the view definition cannot be contained in the column without truncation.
CHECK_OPTION	VARCHAR(8)	The check option used on the view NONE No check option was specified LOCAL The local option was specified CASCADED The cascaded option was specified
IS_UPDATABLE	VARCHAR(3)	Specifies if the view is updatable: YES The view is updatable NO The view is read-only

Appendix G. Reserved Words

This is the list of currently reserved DB2 UDB for iSeries words. Words may be added at any time. For a list of additional words that may become reserved in the future, see the IBM SQL and ANSI reserved words in the *IBM SQL Reference Version 1 SC26-3255*.

Table 150. SQL Reserved Words

ADD	CURRENT_TIMEZONE	GENERAL	MINUTE
ALIAS	CURRENT_USER	GENERATED	MINUTES
ALL	CURSOR	GET	MINVALUE
ALLOCATE	CYCLE	GLOBAL	MODE
ALLOW	DATABASE	GO	MODIFIES
ALTER	DAY	GOTO	MONTH
AND	DAYS	GRANT	MONTHS
ANY	DBINFO	GRAPHIC	NEW
AS	DB2GENERAL	GROUP	NEW_TABLE
AUTHORIZATION	DB2GENRL	HANDLER	NO
BEGIN	DB2SQL	HAVING	NOCACHE
BETWEEN	DECLARE	HOLD	NOCYCLE
BINARY	DEFAULT	HOURL	NODENAME
BY	DEFAULTS	HOURS	NODENUMBER
CACHE	DEFINITION	IDENTITY	NOMAXVALUE
CALL	DELETE	IF	NOMINVALUE
CALLED	DESCRIPTOR	IMMEDIATE	NOORDER
CARDINALITY	DETERMINISTIC	IN	NOT
CASE	DISALLOW	INCLUDING	NULL
CAST	DISCONNECT	INCREMENT	OF
CCSID	DISTINCT	INDEX	OLD
CHAR	DO	INDICATOR	OLD_TABLE
CHARACTER	DOUBLE	INNER	ON
CHECK	DROP	INOUT	OPEN
CLOSE	DYNAMIC	INSENSITIVE	OPTIMIZE
COLLECTION	EACH	INSERT	OPTION
COLUMN	ELSE	INTEGRITY	OR
COMMENT	ELSEIF	INTO	ORDER
COMMIT	END	IS	OUT
CONCAT	END-EXEC (COBOL only)	ISOLATION	OUTER
CONDITION	ESCAPE	ITERATE	OVERRIDING
CONNECT	EXCEPTION	JAVA	PACKAGE
CONNECTION	EXCLUDING	JOIN	PARAMETER
CONSTRAINT	EXECUTE	KEY	PARTITION
CONTAINS	EXISTS	LABEL	PATH
CONTINUE	EXIT	LANGUAGE	POSITION
COUNT	EXTERNAL	LEAVE	PREPARE
COUNT_BIG	FENCED	LEFT	PRIMARY
CREATE	FETCH	LIKE	PRIVILEGES
CROSS	FILE	LINKTYPE	PROCEDURE
CURRENT	FINAL	LOCK	PROGRAM
CURRENT_DATE	FOR	LONG	READ
CURRENT_PATH	FOREIGN	LOOP	READS
CURRENT_SERVER	FREE	MAXVALUE	RECOVERY
CURRENT_TIME	FROM	MICROSECOND	REFERENCES
CURRENT_TIMESTAMP	FUNCTION	MICROSECONDS	REFERENCING


Reserved Words


| *Table 151. SQL Reserved Words (continued)*


	RELEASE	RRN	SQLID	UNTIL
	RENAME	RUN	START	UPDATE
	REPEAT	SAVEPOINT	STATIC	USAGE
	RESET	SCHEMA	SUBSTRING	USER
	RESIGNAL	SCRATCHPAD	SYNONYM	USING
	RESTART	SECOND	TABLE	VALUES
	RESULT	SECONDS	THEN	VARIABLE
	RETURN	SELECT	TO	VARIANT
	RETURNS	SET	TRANSACTION	VIEW
	REVOKE	SIGNAL	TRIGGER	WHEN
	RIGHT	SIMPLE	TRIM	WHERE
	ROLLBACK	SOME	TYPE	WHILE
	ROUTINE	SOURCE	UNDO	WITH
	ROW	SPECIFIC	UNION	WRITE
	ROWS	SQL	UNIQUE	YEAR
				YEARS


Bibliography


The publications listed here provide additional information about topics described or referred to in this guide. These manuals are listed with their full titles and order numbers. When these manuals are referred to in this guide, a shortened version of the title is used.

- Backup and Recovery 

The manual contains information about planning a backup and recovery strategy, the different types of media available to save and restore procedures, and disk recovery procedures. It also describes how to install the system again from backup.
- ILE COBOL Programmer's Guide 

This guide provides information needed to design, write, test, and maintain COBOL/400 programs on the iSeries 400 system.
- ILE RPG Programmer's Guide 


This guide provides information you need to design, write, test, and maintain ILE RPG/400 programs on the iSeries 400 system.
- REXX/400 Programmer's Guide 

This guide provides information you need to design, write, test, and maintain REXX/400 programs on the iSeries 400 system.
- CL Programming 

This guide provides a wide-ranging discussion of the iSeries 400 programming topics, including a general discussion of objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- File Management

This book provides information about using files in application programs.
- Database Programming


This book provides a detailed description of the iSeries database organization, including information on how to create, describe, and update database files on the system.
- Distributed Database Programming

Provides information on preparing and managing an iSeries system in a distributed relational database using the Distributed Relational Database Architecture (DRDA). Describes planning, setting up, programming, administering, and operating a distributed relational database on more than one iSeries system in a like-system environment.
- iSeries Security Reference 

This guide provides information about system security concepts, planning for security, and setting up security on the system. It also gives information about protecting the system and data from being used by people who do not have the proper authorization, protecting the data from intentional or unintentional damage or destruction, keeping security up-to-date, and setting up security on the system.
- SQL Programming Concepts

This book provides an overview of how to design, write, run, and test DB2 UDB for iSeries statements. It also describes interactive Structured Query Language (SQL).
- SQL Programming with Host Languages

This book provides examples of how to write SQL statements in COBOL, ILE COBOL/400, ILE RPG/400, ILE C/400, and PL/I programs.
- Database Performance and Query Optimization

This book provides information on optimizing the performance of your queries using available tools and techniques.
- IDDU Use 

This book describes how to use iSeries interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system.
- SQL Call Level Interfaces (ODBC)

This book describes how to use X/Open SQL Call Level Interface to access SQL functions directly through procedure calls to a service program provided by DB2 UDB for iSeries.

- Client Access Express category in the iSeries Information Center

This information describes how to set up and run ODBC applications on a client using Client Access ODBC. Included in this document are chapters on performance, examples, and configuring specific applications to run with Client Access ODBC.

- IBM Toolbox for Java

This book describes how to set up and run JDBC applications on a client using IBM Toolbox for Java. Included in this document are chapters on performance, examples, and configuring specific applications to run with IBM Toolbox for Java.

- IBM Developer Kit for Java

This information provides the details you need to design, write, test, and maintain JAVA programs on the iSeries system. The book also contains information on the IBM Developer Kit for Java JDBC driver.

- DB2 Multisystem

This book describes the fundamental concepts of distributed relational database files, nodegroups, and partitioning. The book provides the information you need to create and use database files that are partitioned across multiple systems. Information is provided on how to configure the systems, how to create the files, and how the files can be used in applications.

Index

Special characters

- (subtraction) 129
- ? (question mark)
 - See parameter marker
- / (divide) 129
- ' (apostrophe) 43, 99, 102
- " (quotation mark) 43
- * (asterisk) 170, 172
 - in subselect 331
- * (multiply) 129
- ** (exponentiation) 129
- *ALL (read stability) precompiler option 22
- *APOST precompiler option 103
- *APOSTSQL precompiler option 103
- *CHG (uncommitted read) precompiler option 23
- *CNULRQD precompiler option 84, 650, 761, 774
- *CS (cursor stability) precompiler option 23
- *DMY date and time format 69
- *EUR date and time format 69, 70
- *HMS date and time format 70
- *ISO date and time format 69, 70
- *JIS date and time format 69, 70
- *JUL date and time format 69
- *MDY date and time format 69
- *NC (no commit) precompiler option 24
- *NOCNULRQD precompiler option 84, 650, 761, 774
- *NONE (no commit) precompiler option 24
- *QUOTE precompiler option 103
- *QUOTESQL precompiler option 103
- *RR (repeatable read) precompiler option 22
- *RS (read stability) precompiler option 22
- *UR (uncommitted read) precompiler option 23
- *USA date and time format 69, 70
- *YMD date and time format 69
- || (concatenation operator) 131
- + (addition) 129

A

- ABS function 179
- ABSVAL function 179
- access plan and packages 13
- ACOS function 180
- activation group 15
 - threads 20
- ADD check-constraint clause
 - ALTER TABLE statement 387
- ADD COLUMN clause
 - in ALTER TABLE statement 376
- ADD unique-constraint clause
 - ALTER TABLE statement 384

- administrative authority
 - description 36
- AFTER clause
 - in FETCH statement 646
- alias
 - description 56
 - dropping 632
- ALIAS clause
 - COMMENT statement 408
 - CREATE ALIAS statement 425
 - DROP statement 632
 - LABEL statement 682
- alias-name
 - description 45
 - in CREATE ALIAS statement 425
 - in DROP statement 632
 - in LABEL statement 682
- ALL clause
 - clause of subselect 331
 - DISCONNECT statement 627
 - GRANT (Distinct Type Privileges) statement 652
 - GRANT (function or procedure privileges) statement 657
 - GRANT (package privileges) statement 662
 - in USING clause
 - DESCRIBE statement 619
 - DESCRIBE TABLE statement 625
 - PREPARE statement 693
- keyword
 - AVG function 169
 - COUNT function 170
 - COUNT_BIG function 172
 - MAX function 173
 - MIN function 174
 - STDDEV function 175
 - STDDEV_POP function 175
 - SUM function 176
 - VAR function 177
 - VAR_POP function 177
 - VARIANCE function 177
- quantified predicate 149
- RELEASE statement 700
- REVOKE (Distinct Type Privileges) statement 706
 - REVOKE (function or procedure privileges) statement 710
 - REVOKE (package privileges) statement 715
 - REVOKE (table privileges) statement 717
- ALL PRIVILEGES clause
 - GRANT (Distinct Type Privileges) statement 652
 - GRANT (function or procedure privileges) statement 657
 - GRANT (package privileges) statement 662
 - GRANT (table privileges) statement 665

- ALL PRIVILEGES clause (*continued*)
 - REVOKE (Distinct Type Privileges) statement 706
 - REVOKE (function or procedure privileges) statement 710
 - REVOKE (package privileges) statement 715
 - REVOKE (table privileges) statement 717
- ALL SQL clause
 - DISCONNECT statement 627
 - RELEASE statement 700
- ALLOCATE clause
 - CREATE TABLE statement 535
- ALLOW READ clause
 - in LOCK TABLE statement 684
- ALTER clause
 - GRANT (Distinct Type Privileges) statement 653
 - GRANT (function or procedure privileges) statement 658
 - GRANT (package privileges) statement 662
 - GRANT (table privileges) statement 665
 - REVOKE (Distinct Type Privileges) statement 706
 - REVOKE (function or procedure privileges) statement 711
 - REVOKE (package privileges) statement 715
 - REVOKE (table privileges) statement 718
- ALTER COLUMN clause
 - ALTER TABLE statement 382
- ALTER TABLE statement 370, 393
- ALWBK clause
 - in SET OPTION statement 737
- ALWCPYDTA clause
 - in SET OPTION statement 738
- ambiguous reference 111
- AND
 - truth table 161
- ANTILOG function 181
- ANY clause
 - in USING clause
 - DESCRIBE statement 619
 - DESCRIBE TABLE statement 625
 - PREPARE statement 692
 - quantified predicate 149
- application process 15
- application program
 - SQLCA 825
 - C 831
 - COBOL 831
 - FORTRAN 831
 - ILE RPG/400 833
 - PL/I 832
 - RPG/400 832
 - SQLDA 847
 - C 857

- application program (*continued*)
 - SQLDA (*continued*)
 - COBOL 860
 - description 847
 - ILE COBOL 860
 - ILE RPG/400 862
 - PL/I 861
- application requester 25, 840
- application server 25
- application-directed distributed unit of work 29
- arithmetic operators 129
- ARRAY clause
 - SET RESULT SETS statement 750
- AS clause 351
 - clause of subselect 331
 - CREATE VIEW statement 571
 - FROM clause of UPDATE 765
 - in FROM clause of DELETE 614
- AS LOCATOR clause
 - CREATE PROCEDURE (External) 504
 - in CREATE FUNCTION (External Scalar) 443
 - in CREATE FUNCTION (External Table) 459, 460
 - in CREATE FUNCTION (Sourced) 473
 - in DECLARE PROCEDURE statement 601
- AS subquery clause
 - in CREATE TABLE statement 544
 - in DECLARE GLOBAL TEMPORARY TABLE statement 593
- ASC clause
 - CREATE INDEX statement 498
 - of select-statement 352
- ASIN function 182
- assignment
 - binary strings 83
 - character strings 84
 - conversion rules 85
 - DataLink 86
 - date and time values 85
 - distinct type 88
 - graphic strings 84
 - numbers 82, 83
 - Row ID 87
 - strings 83
- assignment-clause
 - UPDATE statement 765
- asterisk (*)
 - in COUNT function 170
 - in COUNT_BIG function 172
 - in subselect 331
- ATAN2 function 185
- ATANH function 184
- authorization
 - description 36
 - privileges 36
 - to create in a schema 36
- authorization ID
 - description 57
- authorization-name
 - definition 45
 - description 58
 - in CONNECT (Type 1) statement 417

- authorization-name (*continued*)
 - in CONNECT (Type 2) statement 422
 - in CREATE SCHEMA statement 522
 - in GRANT (Distinct Type Privileges) statement 653
 - in GRANT (function or procedure privileges) statement 660
 - in GRANT (package privileges) statement 663
 - in GRANT (table privileges) statement 666
 - in REVOKE (Distinct Type Privileges) statement 707
 - in REVOKE (function or procedure privileges) statement 713
 - in REVOKE (package privileges) statement 716
 - in REVOKE (table privileges) statement 718
- AVG function 169

B

- base table 5
- basic operations in SQL 80
- basic predicate 148
- BEFORE clause
 - in FETCH statement 646
- BEGIN DECLARE SECTION
 - statement 394, 395
- BETWEEN predicate 151
- bibliography 1011
- big integers 61
- BIGINT
 - data type for CREATE FUNCTION (External Scalar) 442
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 532
 - data type for DECLARE PROCEDURE 601
- BIGINT data type 61
- BIGINT function 186
- binary data string
 - constants 102
- binary large object (BLOB)
 - data type 65
 - description 65
- binary string
 - assignment 83
 - description 65
- bind 3
- bit data 63
- BLOB
 - data type 65
 - data type for CREATE FUNCTION (External Scalar) 442

- BLOB (*continued*)
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 534
- DECLARE PROCEDURE
 - statement 601
 - description 65
- BLOB function 187
- BOTH clause
 - in USING clause
 - DESCRIBE statement 619
 - DESCRIBE TABLE statement 625
 - PREPARE statement 693
- built-in data type
 - CREATE DISTINCT TYPE statement 430
- built-in function 123
 - See* function
- built-in-type
 - description 532
 - in CREATE TABLE 532
- built-in-type
 - in DECLARE GLOBAL TEMPORARY TABLE statement 588

C

- C
 - application program
 - host variable 120
 - host structure arrays 122
 - host variable 114
 - SQLCA (SQL communication area) 831
 - SQLDA (SQL descriptor area) 857
- CACHE clause
 - in ALTER TABLE statement 380, 383
- call level interface (CLI) 4
- CALL statement 396, 401
- CALLED ON NULL INPUT clause
 - CREATE FUNCTION statement 446, 462, 483, 492
- calling
 - procedures, external 396
- CASCADE clause
 - DROP statement 633, 636, 637
 - in DROP COLUMN of ALTER TABLE statement 384
 - in DROP constraint of ALTER TABLE statement 388
- CASCADE delete rule
 - description 8
 - in ALTER TABLE statement 386
 - in CREATE TABLE statement 548
- CASCADED CHECK OPTION clause
 - CREATE VIEW statement 571
- CASE expression 140

CAST specification 143
 cast-function
 ALTER TABLE statement 378, 398
 CREATE TABLE statement 537
 DECLARE GLOBAL TEMPORARY TABLE statement 589
 catalog 15, 881
 catalog table
 SYSPARMS 913
 SYSROUTINES 924
 SYSTYPES 941
 catalog view
 CHARACTER_SETS 977
 CHECK_CONSTRAINTS 978
 COLUMNS 979
 description 881
 INFORMATION_SCHEMA_CATALOG_NAME 983
 PARAMETERS 984
 REFERENTIAL_CONSTRAINTS 988
 ROUTINES 989
 SCHEMATA 997
 SQL_FEATURES 998
 SQL_LANGUAGES 999
 SQL_SIZING 1000
 SQLCOLPRIVILEGES 950
 SQLCOLUMNS 951
 SQLFOREIGNKEYS 956
 SQLPRIMARYKEYS 957
 SQLPROCEDURECOLS 958
 SQLPROCEDURES 962
 SQLSCHEMAS 963
 SQLSPECIALCOLUMNS 964
 SQLSTATISTICS 966
 SQLTABLEPRIVILEGES 967
 SQLTABLES 968
 SQLTYPEINFO 969
 SQLUDTS 974
 SYSCATALOGS 885
 SYSCHKCST 887
 SYSCOLUMNS 888
 SYSCST 897
 SYSCSTCOL 898
 SYSCSTDEP 899
 SYSFUNCS 900
 SYSINDEXES 906
 SYSJARCONTENTS 907
 SYSJAROBJECTS 908
 SYSKEYCST 909
 SYSKEYS 910
 SYSPACKAGE 911
 SYSPROCS 917
 SYSREFCST 922
 SYSROUTINEDEP 923
 SYSTABLES 932
 SYSTRIGCOL 934
 SYSTRIGDEP 935
 SYSTRIGGERS 936
 SYSTRIGUPD 940
 SYSVIEWDEP 946
 SYSVIEWS 948
 TABLE_CONSTRAINTS 1001
 TABLES 1002
 USER_DEFINED_TYPES 1003
 VIEWS 1007
 CCSID (coded character set identifier)
 default 34
 CCSID (coded character set identifier) (continued)
 definition 34
 specifying
 in SQLDATA 856
 in SQLNAME 856
 values 863, 881
 CCSID clause
 CREATE FUNCTION (Sourced) 472
 CREATE FUNCTION (SQL Scalar) 481
 CREATE FUNCTION (SQL Table) 490
 CREATE PROCEDURE (External) 504
 CREATE PROCEDURE (SQL) 516
 CREATE TABLE statement 536
 data type for CREATE FUNCTION (External Scalar) 442
 data type for CREATE FUNCTION (External Table) 458
 DECLARE PROCEDURE statement 601
 DECLARE VARIABLE statement 609
 CDRA (Character Data Representation Architecture) 34
 CEILING function 189
 CHAR
 data type 62
 data type for CREATE FUNCTION (External Scalar) 442
 data type for CREATE FUNCTION (External Table) 458
 data type for CREATE FUNCTION (Sourced) 472
 data type for CREATE FUNCTION (SQL Scalar) 481
 data type for CREATE FUNCTION (SQL Table) 490
 data type for CREATE PROCEDURE (External) 504
 data type for CREATE PROCEDURE (SQL) 516
 data type for CREATE TABLE 533
 data type for DECLARE PROCEDURE 601
 function 190
 CHAR_LENGTH function 195
 character conversion 32
 character set 32
 code page 32
 code point 32
 coded character set 32
 encoding scheme 32
 substitution character 33
 Character Data Representation Architecture (CDRA) 34
 character data string
 bit data 63
 comparison 89
 constants 99
 description 62
 empty 62
 mixed data 63
 SBCS data 63
 character large object (CLOB)
 data type 65
 character large object (CLOB) (continued)
 description 65
 character set 32
 character string
 assignment 84
 CHARACTER_LENGTH function 195
 CHARACTER_SETS view 977
 check
 ALTER TABLE statement 387
 CHECK clause
 ALTER TABLE statement 382, 387
 CREATE TABLE statement 543, 549
 check constraint 9
 effect on insert 678
 effect on update 768
 check constraints
 delete rules 615
 CHECK OPTION clause
 CREATE VIEW statement 571
 effect on update 768
 CHECK_CONSTRAINTS view 978
 check-condition
 in CHECK clause of ALTER TABLE statement 387
 CLOB
 data type 65
 data type for CREATE FUNCTION (External Scalar) 442
 data type for CREATE FUNCTION (External Table) 458
 data type for CREATE FUNCTION (Sourced) 472
 data type for CREATE FUNCTION (SQL Scalar) 481
 data type for CREATE FUNCTION (SQL Table) 490
 data type for CREATE PROCEDURE (External) 504
 data type for CREATE PROCEDURE (SQL) 516
 data type for CREATE TABLE 533
 DECLARE PROCEDURE statement 601
 description 65
 CLOB function 196
 CLOSE statement 402, 403
 closed state of cursor 688
 CLOSQLCSR clause
 in SET OPTION statement 738
 CNULRQD clause
 in SET OPTION statement 739
 COALESCE function 200
 COBOL
 application program
 host structure arrays 122
 host variable 114, 120
 integers 61
 varying-length string variables 62
 SQLCA (SQL communication area) 831
 SQLDA (SQL descriptor area) 860
 code page 32
 code point 32
 collection
 in SQL path 53
 collection (see schema)
 description 5

- column
 - definition 5
 - length attribute 62
 - name
 - in a result 333
 - qualified 108
 - rules 346
 - system column name 6
- COLUMN clause
 - COMMENT statement 408
 - LABEL statement 682
- column function 123
 - See* function
- column-name
 - definition 45
 - in ADD PRIMARY clause of ALTER TABLE statement 384
 - in ADD UNIQUE clause of ALTER TABLE statement 384
 - in ALTER TABLE statement 376, 382
 - in CREATE TABLE statement 532, 546, 547
 - in CREATE VIEW statement 570
 - in DECLARE GLOBAL TEMPORARY TABLE statement 588
 - in DROP COLUMN of ALTER TABLE statement 384
 - in FOREIGN KEY clause of ALTER TABLE statement 385
 - in INSERT statement 675
 - in LABEL statement 682
 - in REFERENCES clause of ALTER TABLE statement 385
 - in UPDATE statement 765
- COLUMNS view 979
- comment
 - in catalog table 404
 - SQL 41, 368
- COMMENT statement 404, 412
 - name qualification 108
- COMMIT
 - effect on SET TRANSACTION 756
- COMMIT clause
 - in SET OPTION statement 739
- COMMIT ON RETURN clause
 - CREATE PROCEDURE (External) 509
 - CREATE PROCEDURE (SQL) 518
- commit point 413
- COMMIT statement 413, 415
- commitment definition 16
- common table expression clause
 - of select-statement 350
- comparison
 - compatibility rules 80
 - conversion rules 90
 - date and time values 91
 - distinct type values 91
 - numbers 89
 - strings 89
- compatibility
 - data types 80
 - rules 80
- composite key 6
- CONCAT (concatenation operator) 131
- CONCAT function 201
- concatenation operator (CONCAT) 131
- concurrency 15
 - with LOCK TABLE statement 684
- CONNECT
 - differences, type 1 and type 2 845
- CONNECT (Type 1) statement 416, 420
- CONNECT (Type 2) statement 421, 425
- connected state 30
- connection
 - changing with SET CONNECTION 730
 - ending 700
 - releasing 700
 - SQL 28
- connection states
 - activation group 30
 - CONNECT (Type 2) statement 28
 - distributed unit of work 29
 - remote unit of work 27
- constant
 - DECLARE GLOBAL TEMPORARY TABLE statement 590
 - in ALTER TABLE statement 377, 378
 - in CALL statement 398, 399
 - in CREATE TABLE statement 538
 - in LABEL statement 682
- constants
 - binary string 102
 - character string 99
 - decimal 99
 - floating point 99
 - graphic string 100
 - hexadecimal 99, 102
 - integer 99
 - UCS-2 101
- CONSTRAINT clause
 - in ALTER TABLE statement 381, 384, 385, 387
 - in CREATE TABLE statement 542, 546, 547, 548
- constraint-name
 - description 45
 - in ALTER TABLE statement 381, 384, 387
 - in CONSTRAINT clause of ALTER TABLE statement 385
 - in CREATE TABLE statement 542, 546, 547, 548
 - in DROP CHECK clause of ALTER TABLE statement 388
 - in DROP CONSTRAINT clause of ALTER TABLE statement 388
 - in DROP FOREIGN KEY clause of ALTER TABLE statement 388
 - in DROP UNIQUE clause of ALTER TABLE statement 388
- CONTAINS SQL clause
 - CREATE PROCEDURE (External) 508
 - in CREATE FUNCTION (External Scalar) 445
 - in CREATE FUNCTION (External Table) 461
 - in CREATE FUNCTION (SQL Scalar) 483
 - in CREATE FUNCTION (SQL Table) 492
 - in CREATE PROCEDURE (SQL) 517
- CONTAINS SQL clause (*continued*)
 - in DECLARE PROCEDURE 603
- CONTINUE clause
 - WHENEVER statement 776
- control characters 41
- conversion of numbers
 - conversion rule for comparisons 85
 - scale and precision 82
- correlated reference 111
- correlation name
 - defining 108
 - description 46
 - FROM clause
 - of subselect 336
 - qualifying a column name 108
- correlation-name
 - in DELETE statement 614
 - in UPDATE statement 765
- COS function 202
- COSH function 203
- COT function 204
- COUNT function 170
- COUNT_BIG function 172
- CREATE ALIAS statement 13, 425, 427
- CREATE DISTINCT TYPE statement 428
- CREATE FUNCTION (external scalar) statement 454
- CREATE FUNCTION (External Scalar) statement 439
- CREATE FUNCTION (External Table) statement 455
- CREATE FUNCTION (Sourced) statement 469
- CREATE FUNCTION (SQL Scalar) statement 478
- CREATE FUNCTION (SQL Table) statement 487
- CREATE INDEX statement 496
- CREATE PROCEDURE (External) statement 501
- CREATE PROCEDURE (SQL) statement 512, 520
- CREATE SCHEMA statement 521, 525
- CREATE TABLE statement 525
- CREATE TRIGGER statement 556
- CREATE VIEW statement 12, 569, 575
- CROSS JOIN clause
 - in FROM clause 339
- CS (cursor stability) 23
- CURDATE function 205
- CURRENT clause
 - in DISCONNECT statement 627
 - in FETCH statement 646
 - in RELEASE statement 700
- current connection state 30
- CURRENT DATE special register 105
- current path special register 747
 - SET PATH 747
 - SET SCHEMA 753
- CURRENT PATH special register 105
- CURRENT SCHEMA special register 106
- CURRENT SERVER special register 106
- CURRENT TIME special register 106
- CURRENT TIMESTAMP special register 107

CURRENT TIMEZONE special register 107

CURRENT_DATE
 ALTER TABLE statement 378, 379
 CREATE TABLE statement 537, 538
 DECLARE GLOBAL TEMPORARY TABLE statement 589, 590

CURRENT_DATE special register 105

CURRENT_PATH special register 105

CURRENT_SERVER special register 106

CURRENT_TIME
 ALTER TABLE statement 378, 379
 CREATE TABLE statement 537, 538
 DECLARE GLOBAL TEMPORARY TABLE statement 589, 590

CURRENT_TIME special register 106

CURRENT_TIMESTAMP
 ALTER TABLE statement 378, 379
 CREATE TABLE statement 537, 538
 DECLARE GLOBAL TEMPORARY TABLE statement 589, 590

CURRENT_TIMESTAMP special register 107

CURRENT_TIMEZONE special register 107

cursor
 active set 686
 closed by error
 FETCH statement 649
 UPDATE 768
 closed state 688
 closing 402
 current row 649
 defining 576
 deletable 579
 moving position 645
 positions for open 649
 preparing 686
 read-only 579
 updatable 579

cursor stability 23

cursor-name
 description 46
 in CLOSE statement 402
 in DECLARE CURSOR statement 577
 in DELETE statement 614
 in FETCH statement 647
 in OPEN statement 686
 in SET RESULT SETS statement 750
 in UPDATE statement 767

CURTIME function 206

CYCLE clause
 in ALTER TABLE statement 381, 383

D

data access indication 838

DATA DICTIONARY clause
 CREATE SCHEMA statement 522

data representation
 in DRDA 31

data type
 binary large object (BLOB) 65
 binary string 65
 character large object (CLOB) 65
 character string 62

data type (*continued*)
 DataLink 72
 datetime 67
 description 59, 532
 distinct types 73
 double-byte character large object (DBCLOB) 65
 in SQLDA 849
 large object (LOB) 65
 numeric 61
 result columns 333
 Row ID 73
 user-defined types (UDTs) 73

data-type 443, 459, 473, 482, 491

CREATE PROCEDURE
 (External) 504
 in ALTER TABLE 376
 in ALTER TABLE statement 376, 382
 in CAST specification 145
 in CREATE FUNCTION (External Scalar) 443
 in CREATE FUNCTION (External Table) 459
 in CREATE FUNCTION (Sourced) 472, 473
 in CREATE FUNCTION (SQL Scalar) 482
 in CREATE FUNCTION (SQL Table) 491
 in CREATE PROCEDURE (SQL) 516
 in CREATE TABLE 532
 in DECLARE GLOBAL TEMPORARY TABLE statement 588
 in DECLARE PROCEDURE statement 601

database manager limits 823

DataLink
 assignment 86
 data type
 description 72
 limits 823

DATALINK
 data type for CREATE FUNCTION (Sourced) 472
 data type for CREATE FUNCTION (SQL Scalar) 481
 data type for CREATE FUNCTION (SQL Table) 490
 data type for CREATE PROCEDURE (SQL) 516
 data type for CREATE TABLE 535
 DECLARE PROCEDURE statement 601

datalink-options
 in ALTER TABLE statement 381
 in CREATE TABLE statement 541
 in DECLARE GLOBAL TEMPORARY TABLE statement 592

date
 duration 134
 strings 69

DATE
 arithmetic operations 135
 assignment 86
 data type 67
 data type for CREATE TABLE 535
 function 207

date and time 68
 arithmetic operations 134, 138
 assignments 85
 comparisons 91
 data types
 string representation 69
 default date format 69, 102
 default time format 71
 format 191
 day/month/year 69
 EUR 69, 70
 hours/minutes/seconds 70
 ISO 69, 70
 JIS 69, 70
 Julian 69
 month/day/year 69
 unformatted Julian 69
 USA 69, 70
 year/month/day 69

datetime
 data types
 description 67
 limits 823

DATFMT clause
 in SET OPTION statement 739

DATSEP clause
 in SET OPTION statement 740

DAY function 209

DAYOFMONTH function 210

DAYOFWEEK function 211

DAYOFWEEK_ISO function 212

DAYOFYEAR function 213

DAYS function 214

DB2GENERAL clause
 CREATE PROCEDURE (External) 506
 DECLARE PROCEDURE (External) 605
 in CREATE FUNCTION (External Scalar) 450
 in CREATE FUNCTION (External Table) 465

DB2SQL clause
 CREATE PROCEDURE (External) 506
 DECLARE PROCEDURE (External) 605
 in CREATE FUNCTION (External Scalar) 451
 in CREATE FUNCTION (External Table) 466

DBCLOB
 data type 65
 data type for CREATE FUNCTION (External Scalar) 442
 data type for CREATE FUNCTION (External Table) 458
 data type for CREATE FUNCTION (Sourced) 472
 data type for CREATE FUNCTION (SQL Scalar) 481
 data type for CREATE FUNCTION (SQL Table) 490
 data type for CREATE PROCEDURE (External) 504
 data type for CREATE PROCEDURE (SQL) 516

DBCLOB (*continued*)
 data type for CREATE TABLE 534
 DECLARE PROCEDURE
 statement 601
 description 65
 function 215
 DBCS (double-byte character set)
 description 65
 truncated during assignment 85
 DBGVIEW clause
 in SET OPTION statement 740
 decimal
 constants 99
 data type 61
 numbers 61
 DECIMAL
 data type for CREATE FUNCTION
 (External Scalar) 442
 data type for CREATE FUNCTION
 (External Table) 458
 data type for CREATE FUNCTION
 (Sourced) 472
 data type for CREATE FUNCTION
 (SQL Scalar) 481
 data type for CREATE FUNCTION
 (SQL Table) 490
 data type for CREATE PROCEDURE
 (External) 504
 data type for CREATE PROCEDURE
 (SQL) 516
 data type for CREATE TABLE 532
 data type for DECLARE
 PROCEDURE 601
 decimal data
 arithmetic 131
 DECIMAL function 217
 decimal point 102
 declaration
 inserting into a program 671
 DECLARE CURSOR statement 576, 578,
 582
 DECLARE GLOBAL TEMPORARY
 TABLE statement 583, 597
 DECLARE PROCEDURE statement 598,
 606
 DECLARE STATEMENT statement 607,
 608
 DECLARE statements
 BEGIN DECLARE SECTION
 statement 394
 END DECLARE SECTION
 statement 639
 DECLARE VARIABLE statement 609,
 611
 DECMPT clause
 in SET OPTION statement 741
 DEFAULT
 in SET transition-variable
 statement 759
 in UPDATE statement 766
 DEFAULT clause
 ALTER TABLE statement 377
 CREATE TABLE statement 536
 in DECLARE GLOBAL TEMPORARY
 TABLE statement 588
 in INSERT statement 676
 default date format 68, 69, 102
 default decimal separator character
 description 62
 default schema
 name qualification 54
 default time format 68, 71
 DEGREES function 220
 DELETE clause
 GRANT (table privileges)
 statement 665
 in ON DELETE clause of ALTER
 TABLE statement 386
 in ON DELETE clause of CREATE
 TABLE statement 548
 REVOKE (table privileges)
 statement 718
 delete rules
 check constraints 615
 referential constraint 9
 referential integrity 615
 triggers 615
 DELETE statement 612, 617
 delete-connected table 9
 deleting SQL objects 629
 delimited identifier 43
 in system names 43
 dependent row 7
 dependent table 7
 derived table 335
 DESC clause
 CREATE INDEX statement 498
 of select-statement 352
 descendent row 7
 descendent table 7
 DESCRIBE statement 618, 622
 variables
 SQLD 619
 SQLDABC 619
 SQLDAID 618
 SQLN 618
 SQLVAR 619
 DESCRIBE TABLE statement 623, 626
 description 626
 variables
 SQLD 624
 SQLDABC 624
 SQLDAID 624
 SQLN 623
 SQLVAR 624
 descriptor-name
 description 46
 in CALL statement 399
 in DESCRIBE statement 618
 in EXECUTE statement 640
 in FETCH statement 647
 in OPEN statement 687
 in PREPARE statement 692
 designator
 table 110, 287
 DETERMINISTIC clause
 CREATE PROCEDURE
 (External) 508
 in CREATE FUNCTION (External
 Scalar) 445
 in CREATE FUNCTION (External
 Table) 461
 in CREATE FUNCTION (SQL
 Scalar) 482
 DETERMINISTIC clause (*continued*)
 in CREATE FUNCTION (SQL
 Table) 491
 in CREATE PROCEDURE (SQL) 517
 in DECLARE PROCEDURE 602
 DFTRDBCOL clause
 in SET OPTION statement 741
 DIFFERENCE function 221
 DIGITS function 222
 dirty read 25
 DISCONNECT statement 627, 628
 DISCONNECT 628
 disconnecting SQL objects 627
 DISTINCT
 AVG function 169
 COUNT function 170
 COUNT_BIG function 172
 MAX function 173
 MIN function 174
 STDDEV function 175
 STDDEV_POP function 175
 SUM function 176
 VAR function 177
 VARIANCE function 177
 VARPOP function 177
 DISTINCT clause
 subselect 331
 distinct type
 assignment 88
 comparisons 91
 DISTINCT TYPE clause 404
 COMMENT statement 404, 409
 distinct types
 data types
 description 73
 distinct-type
 data type for CREATE FUNCTION
 (External Scalar) 442
 data type for CREATE FUNCTION
 (External Table) 458
 data type for CREATE FUNCTION
 (Sourced) 472
 data type for CREATE FUNCTION
 (SQL Scalar) 481
 data type for CREATE FUNCTION
 (SQL Table) 490
 data type for CREATE PROCEDURE
 (External) 504
 data type for CREATE PROCEDURE
 (SQL) 516
 data type for CREATE TABLE 535
 DECLARE PROCEDURE
 statement 601
 distinct-type-name
 description 46
 in CREATE DISTINCT TYPE
 statement 429
 in DROP statement 633
 in REVOKE (Distinct Type Privileges)
 statement 707
 distributed data
 CONNECT statement 845
 distributed relational database
 application requester 25
 application server 25
 application-directed distributed unit
 of work 29

- distributed relational database (*continued*)
 - considerations for using 840, 842, 843, 844
 - data representation considerations 31
 - distributed unit of work 29
 - remote unit of work 27
 - server 25
 - use of extensions to IBM SQL on unlike servers 840, 842, 843, 844
 - distributed relational database architecture (DRDA) 25
 - distributed tables
 - definition 6
 - syntax 549
 - distributed unit of work
 - mixed environment 836
 - division by zero 141
 - division operator 129
 - DLCOMMENT function 223
 - DLLINKTYPE function 224
 - DLURLCOMPLETE function 225
 - DLURLPATH function 226
 - DLURLPATHONLY function 227
 - DLURLSCHEME function 228
 - DLURLSERVER function 229
 - DLVALUE function 230
 - in INSERT statement 398
 - DLYPRP clause
 - in SET OPTION statement 741
 - dormant connection state 30
 - DOUBLE
 - function 232
 - DOUBLE PRECISION
 - data type for CREATE FUNCTION (External Scalar) 442
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 533
 - data type for DECLARE PROCEDURE 601
 - DOUBLE_PRECISION function 232
 - double-byte character
 - in COMMENT statement 412
 - in LIKE predicates 156
 - truncated during assignment 84
 - double-byte character large object (DBCLOB)
 - data type 65
 - description 65
 - double-byte character set (DBCS)
 - truncated during assignment 85
 - double-precision floating point 61
 - DRDA (Distributed Relational Database Architecture) 25
 - DROP CHECK clause
 - ALTER TABLE statement 388
 - DROP COLUMN clause
 - ALTER TABLE statement 384
 - DROP CONSTRAINT clause
 - ALTER TABLE statement 388
 - DROP DEFAULT clause
 - ALTER TABLE statement 383
 - DROP FOREIGN KEY clause
 - ALTER TABLE statement 388
 - DROP IDENTITY clause
 - ALTER TABLE statement 383
 - DROP NOT NULL clause
 - ALTER TABLE statement 383
 - DROP PRIMARY KEY clause
 - ALTER TABLE statement 388
 - DROP statement 629, 638
 - DROP UNIQUE clause
 - ALTER TABLE statement 388
 - duplicate rows with UNION 346
 - duration
 - date 134
 - labeled 134
 - time 134
 - timestamp 134
 - DYNAMIC SCROLL clause
 - in DECLARE CURSOR statement 577
 - dynamic select 367
 - dynamic SQL
 - defined 365
 - description 3
 - execution
 - EXECUTE IMMEDIATE statement 643
 - EXECUTE statement 640
 - in USING clause of DESCRIBE statement 618
 - obtaining statement information with
 - DESCRIBE 618
 - DESCRIBE TABLE 623
 - preparation and execution 366
 - PREPARE statement 691
 - SQLDA (SQL descriptor area) 847
 - statements allowed 836
 - use of SQL path 53
 - DYNDFTCOL clause
 - in SET OPTION statement 742
 - DYNUSRPRF clause
 - in SET OPTION statement 742
- ## E
- Embedded SQL for Java (SQLJ) 4
 - empty character string 62
 - ENCODED VECTOR clause
 - CREATE INDEX statement 497
 - encoding scheme 32
 - END DECLARE SECTION
 - statement 639
 - ending
 - unit of work 413, 720
 - error
 - closes cursor 688
 - during UPDATE 768
 - FETCH statement 649
 - escape character in SQL
 - delimited identifier 43
 - ESCAPE clause of LIKE predicate 157
 - evaluation order 138
 - EVENTF clause
 - in SET OPTION statement 742
 - EXCLUDING clause
 - in CREATE TABLE statement 545
 - in DECLARE GLOBAL TEMPORARY TABLE statement 594
 - EXCLUSIVE
 - ALLOW READ clause
 - LOCK TABLE statement 684
 - IN EXCLUSIVE MODE clause
 - LOCK TABLE statement 684
 - exclusive locks 22
 - EXCLUSIVE MODE clause
 - in LOCK TABLE statement 684
 - executable statement 365
 - EXECUTE clause
 - GRANT (function or procedure privileges) statement 658
 - GRANT (package privileges) statement 663
 - REVOKE (function or procedure privileges) statement 711
 - REVOKE (package privileges) statement 715
 - EXECUTE IMMEDIATE statement 643, 644
 - EXECUTE statement 640, 642
 - EXISTS predicate 152
 - EXP function 234
 - exponentiation operator 129
 - exposed name 336
 - expression
 - CASE expression 140
 - CAST specification 143
 - date and time operands 134
 - decimal operands 130
 - distinct type operands 131
 - floating-point operands 131
 - grouping 341
 - in INSERT statement 676
 - in statement 758, 761
 - in subselect 331
 - in UPDATE statement 766
 - in VALUES INTO statement 773
 - in VALUES statement 771
 - integer operands 130
 - numeric operands 130
 - precedence of operation 138
 - with arithmetic operators 129
 - with concatenation operator 131
 - without operators 129
 - extended dynamic SQL
 - description 4
 - external
 - function 439, 455
 - EXTERNAL clause
 - CREATE PROCEDURE (External) 507
 - in CREATE FUNCTION (External Scalar) 449
 - in CREATE FUNCTION (External Table) 465
 - in DECLARE PROCEDURE 604
 - EXTERNAL NAME clause
 - CREATE PROCEDURE (External) 507

EXTERNAL NAME clause (*continued*)
 in CREATE FUNCTION (External Scalar) 449
 in CREATE FUNCTION (External Table) 465
 in DECLARE PROCEDURE 604
 external-program-name
 description 46

F

FETCH FIRST clause
 of select-statement 353
 FETCH statement 645, 650
 fetch-first-clause 353
 file reference
 variable 117, 118
 FIRST clause
 in FETCH statement 646
 FLOAT
 data type for CREATE FUNCTION (External Scalar) 442
 data type for CREATE FUNCTION (External Table) 458
 data type for CREATE FUNCTION (Sourced) 472
 data type for CREATE FUNCTION (SQL Scalar) 481
 data type for CREATE FUNCTION (SQL Table) 490
 data type for CREATE PROCEDURE (External) 504
 data type for CREATE PROCEDURE (SQL) 516
 data type for CREATE TABLE 533
 data type for DECLARE PROCEDURE 601
 FLOAT function 235
 floating point
 constants 99
 numbers 61
 FLOOR function 236
 FOR BIT DATA clause
 CREATE FUNCTION (External Scalar) 442
 CREATE FUNCTION (External Table) 458
 CREATE FUNCTION (Sourced) 472
 CREATE FUNCTION (SQL Scalar) 481
 CREATE FUNCTION (SQL Table) 490
 CREATE PROCEDURE (External) 504
 CREATE PROCEDURE (SQL) 516
 CREATE TABLE statement 536
 DECLARE PROCEDURE statement 601
 DECLARE VARIABLE statement 609
 FOR clause
 CREATE ALIAS statement 425
 FOR COLUMN clause
 ALTER TABLE statement 376
 CREATE TABLE statement 532
 CREATE VIEW statement 570
 in DECLARE GLOBAL TEMPORARY TABLE statement 588

FOR FETCH ONLY clause
 of select-statement 355
 FOR MIXED DATA clause
 CREATE FUNCTION (External Scalar) 442
 CREATE FUNCTION (External Table) 458
 CREATE FUNCTION (Sourced) 472
 CREATE FUNCTION (SQL Scalar) 481
 CREATE FUNCTION (SQL Table) 490
 CREATE PROCEDURE (External) 504
 CREATE PROCEDURE (SQL) 516
 CREATE TABLE statement 536
 DECLARE PROCEDURE statement 601
 DECLARE VARIABLE statement 609
 FOR READ ONLY clause
 of select-statement 355
 FOR ROWS clause
 FETCH statement 647
 SET RESULT SETS statement 750
 FOR SBCS DATA clause
 CREATE FUNCTION (External Scalar) 442
 CREATE FUNCTION (External Table) 458
 CREATE FUNCTION (Sourced) 472
 CREATE FUNCTION (SQL Scalar) 481
 CREATE FUNCTION (SQL Table) 490
 CREATE PROCEDURE (External) 504
 CREATE PROCEDURE (SQL) 516
 CREATE TABLE statement 536
 DECLARE PROCEDURE statement 601
 DECLARE VARIABLE statement 609
 FOR UPDATE OF clause
 of select-statement 354
 foreign key 7
 FOREIGN KEY clause
 of ALTER TABLE statement 385
 of CREATE TABLE statement 547
 FORTRAN
 SQLCA (SQL communication area) 831
 FREE LOCATOR statement 651
 FROM clause
 correlation clause 335
 correlation-clause 613
 DELETE statement 613
 joined-table 338
 nested table expression 335
 of subselect 335
 PREPARE statement 693
 REVOKE (Distinct Type Privileges) statement 707
 REVOKE (function or procedure privileges) statement 713
 REVOKE (package privileges) statement 715
 REVOKE (table privileges) statement 718

FROM clause (*continued*)
 table reference 335
 fullselect 346
 used in CREATE VIEW statement 571
 function 178
 best fit 125
 built-in 123
 column 123, 168
 AVG 169
 COUNT 170
 COUNT_BIG 172
 MAX 173
 MIN 174
 STDDEV 175
 STDDEV_POP 175
 SUM 176
 VAR 177
 VAR_POP 177
 VARIANCE 177
 commenting 410
 creating 435, 439, 455, 469, 478, 487
 description 163
 dropping 635
 extending a built-in function 437
 external 123, 439, 455
 granting 659
 input parameters 436
 invocation 127
 locators 436
 name restrictions 435
 nesting 178
 overriding a built-in function 437
 resolution 124
 revoking 712
 scalar 123, 178
 ABS 179
 ABSVAL 179
 ACOS 180
 ANTILOG 181
 ASIN 182
 ATAN 183
 ATAN2 185
 ATANH 184
 BIGINT 186
 BLOB 187
 CEILING 189
 CHAR 190
 CHAR_LENGTH 195
 CHARACTER_LENGTH 195
 CLOB 196
 COALESCE 200
 CONCAT 201
 COS 202
 COSH 203
 COT 204
 CURDATE 205
 CURTIME 206
 DATE 207
 DAY 209
 DAYOFMONTH 210
 DAYOFWEEK 211
 DAYOFWEEK_ISO 212
 DAYOFYEAR 213
 DAYS 214
 DBCLOB 215
 DECIMAL 217

- function (*continued*)
 - scalar (*continued*)
 - DEGREES 220
 - DIFFERENCE 221
 - DIGITS 222
 - DLCOMMENT 223
 - DLLINKTYPE 224
 - DLURLCOMPLETE 225
 - DLURLPATH 226
 - DLURLPATHONLY 227
 - DLURLSCHEME 228
 - DLURLSERVER 229
 - DLVALUE 230
 - DOUBLE 232
 - DOUBLE_PRECISION 232
 - EXP 234
 - FLOAT 235
 - FLOOR 236
 - GRAPHIC 237
 - HASH 240
 - HEX 241
 - hour 242
 - IDENTITY_VAL_LOCAL 243
 - IFNULL 247
 - INTEGER 248
 - JULIAN_DAY 250
 - LAND 251
 - LCASE 252
 - LEFT 253
 - LENGTH 255
 - LN 256
 - LNOT 257
 - LOCATE 258
 - LOG 259
 - LOG10 259
 - LOR 260
 - LOWER 261
 - LTRIM 262
 - MAX 263
 - MICROSECOND 264
 - MIDNIGHT_SECONDS 265
 - MIN 266
 - MINUTE 267
 - MOD 268
 - MONTH 270
 - NODENAME 271
 - NODENUMBER 272
 - NOW 273
 - NULLIF 274
 - PARTITION 275
 - PI 276
 - POSITION 277
 - POSSTR 277
 - POWER 279
 - QUARTER 280
 - RADIANS 281
 - RAND 282
 - REAL 283
 - ROUND 284
 - ROWID 286
 - RRN 287
 - RTRIM 288
 - SECOND 289
 - SIGN 290
 - SIN 291
 - SINH 292
 - SMALLINT 293

- function (*continued*)
 - scalar (*continued*)
 - SOUNDEX 294
 - SPACE 295
 - SQRT 296
 - STRIP 297
 - SUBSTR (or SUBSTRING) 298
 - TAN 301
 - TANH 302
 - TIME 303
 - TIMESTAMP 304
 - TIMESTAMPDIFF 306
 - TRANSLATE 307
 - TRIM 309
 - TRUNCATE 311
 - UCASE 313
 - UPPER 314
 - VALUE 315
 - VARCHAR 316
 - VARGRAPHIC 320
 - WEEK 323
 - WEEK_ISO 324
 - XOR 325
 - YEAR 326
 - ZONED 327
 - sourced 123, 469
 - specific name 437
 - SQL 123, 478, 487
 - table 124
 - types 123
 - user-defined 123
 - FUNCTION clause 404
 - COMMENT statement 404, 409
 - DROP statement 633
 - GRANT (function or procedure privileges) statement 658
 - REVOKE (function or procedure privileges) statement 711
 - function reference
 - syntax 124
 - function resolution 53
 - function-name
 - description 47
 - in CREATE FUNCTION (External Scalar) 442
 - in CREATE FUNCTION (External Table) 458
 - in CREATE FUNCTION (Sourced) 472
 - in CREATE FUNCTION (SQL Scalar) 481
 - in CREATE FUNCTION (SQL Table) 490
 - in DROP statement 633
- functions
 - description 123

G

- GENERAL clause
 - CREATE PROCEDURE (External) 506
 - DECLARE PROCEDURE (External) 605
 - in CREATE FUNCTION (External Scalar) 450

- GENERAL WITH NULLS clause
 - CREATE PROCEDURE (External) 506
 - DECLARE PROCEDURE (External) 605
 - in CREATE FUNCTION (External Scalar) 451
- GENERATED
 - in ALTER TABLE statement 379
 - in CREATE TABLE statement 538
 - in DECLARE GLOBAL TEMPORARY TABLE statement 590
- GET DIAGNOSTICS statement 799, 801
 - description 801
- GO TO clause
 - WHENEVER statement 776
- GRANT (Distinct Type Privileges) statement 652, 654
- GRANT (function or procedure privileges) statement 655, 661
- GRANT (package privileges) statement 662, 664
- GRANT (table privileges) statement 664, 665, 668
- GRAPHIC
 - data type for CREATE FUNCTION (External Scalar) 442
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 534
 - data type for DECLARE PROCEDURE 601
 - function 237
- graphic constant
 - hexadecimal 100
- graphic string
 - assignment 84
 - constants 100
 - definition 64
- GROUP BY clause
 - of subselect 341
 - results with subselect 332

H

- HASH function 240
- HASHING
 - in CREATE TABLE statement 550
- HAVING clause
 - of subselect 343
 - results with subselect 332
- held connection state 30
- HEX function 241
- hexadecimal constants 99, 102
- HOLD clause 578
 - COMMIT statement 413
 - ROLLBACK statement 721

HOLD LOCATOR statement 669, 670
 host identifier 44
 host structure
 description 120
 host structure arrays
 description 122
 host variable
 DECLARE VARIABLE statement 609
 description 48, 114
 in CALL statement 399
 indicator variable 115
 LOB file reference 118
 LOB locator 117
 SELECT INTO statement 728
 statement string 643
 substitution for parameter markers 640
 host-identifier
 in host variable 48
 host-label
 description 48
 in WHENEVER statement 776
 host-structure-array
 in FETCH statement 648
 in INSERT statement 677
 in SET RESULT SETS statement 750
 host-variable
 description
 in Java 116
 in CALL statement 397, 398
 in CONNECT (Type 1)
 statement 416, 417
 in CONNECT (Type 2)
 statement 421, 422
 in DECLARE VARIABLE
 statement 609
 in DESCRIBE TABLE statement 623
 in DISCONNECT statement 627
 in EXECUTE IMMEDIATE
 statement 643
 in EXECUTE statement 640
 in FETCH statement 647
 in FREE LOCATOR statement 651
 in HOLD LOCATOR statement 669
 in INSERT statement 677
 in OPEN statement 687
 in PREPARE statement 693
 in RELEASE statement 700
 in SELECT INTO statement 728
 in SET CONNECTION statement 730
 in VALUES INTO statement 773
 HOUR function 242

I

identifiers
 in SQL
 delimited 43
 description 43
 host 44
 ordinary 43
 system 43
 limits 41, 51, 52, 821

IDENTITY
 in ALTER TABLE statement 379
 in CREATE TABLE statement 539

IDENTITY (*continued*)
 in DECLARE GLOBAL TEMPORARY
 TABLE statement 590
 IDENTITY_VAL_LOCAL function 243
 IFNULL function 247
 ILE RPG/400
 SQLCA (SQL communication
 area) 833
 SQLDA (SQL descriptor area) 862

IMMEDIATE
 EXECUTE IMMEDIATE
 statement 643, 644

IN ASP clause
 CREATE SCHEMA statement 522

IN clause
 CREATE PROCEDURE
 (External) 504
 DECLARE PROCEDURE
 statement 601
 in CREATE PROCEDURE (SQL) 516

IN EXCLUSIVE clause
 in LOCK TABLE statement 684

IN predicate 153

IN SHARE MODE clause
 in LOCK TABLE statement 684

INCLUDE statement 671, 672

INCLUDING clause
 in CREATE TABLE statement 545
 in DECLARE GLOBAL TEMPORARY
 TABLE statement 594

INCREMENT BY clause
 ALTER TABLE statement 383
 in ALTER TABLE statement 380

index 11
 dropping 635, 636

INDEX clause 404
 COMMENT statement 404, 410
 CREATE INDEX statement 496
 DROP statement 635
 GRANT (table privileges)
 statement 665
 RENAME statement 704
 REVOKE (table privileges)
 statement 718

index-name
 description 48
 in CREATE INDEX statement 497
 in DROP statement 635
 in RENAME statement 704

indicator
 array 120
 variable 120, 643

infix operators 129

INFORMATION_SCHEMA
 _CATALOG_NAME view 983

INNER JOIN clause
 in FROM clause 339

INOUT clause
 CREATE PROCEDURE
 (External) 504
 DECLARE PROCEDURE
 statement 601
 in CREATE PROCEDURE (SQL) 516

INSENSITIVE clause
 in DECLARE CURSOR
 statement 577

INSERT clause
 GRANT (table privileges)
 statement 665
 REVOKE (table privileges)
 statement 718

insert rule with referential constraint 8

insert rules
 check constraint 678

INSERT statement 673, 680

INTEGER
 data type for CREATE FUNCTION
 (External Scalar) 442
 data type for CREATE FUNCTION
 (External Table) 458
 data type for CREATE FUNCTION
 (Sourced) 472
 data type for CREATE FUNCTION
 (SQL Scalar) 481
 data type for CREATE FUNCTION
 (SQL Table) 490
 data type for CREATE PROCEDURE
 (External) 504
 data type for CREATE PROCEDURE
 (SQL) 516
 data type for CREATE TABLE 532
 data type for DECLARE
 PROCEDURE 601

integer constants 99

INTEGER data type 61

INTEGER function 248

interactive entry of SQL statements 367

interactive SQL 4

INTO clause
 in FETCH statement 647, 648, 649
 in PREPARE statement 692
 in SELECT INTO statement 728
 in VALUES INTO statement 773

INTO DESCRIPTOR clause
 FETCH statement 647

INTO keyword
 DESCRIBE statement 618
 DESCRIBE TABLE statement 623
 INSERT statement 675

IS clause
 COMMENT statement 411
 LABEL statement 682

isolation level
 comparison 24
 CS 23
 cursor stability 23
 description 21
 NC 24
 no commit 24
 read stability
 phantom rows 22
 repeatable read 22
 RR 22
 RS 22
 set using SET TRANSACTION 755
 uncommitted read (UR) 23

ISOLATION LEVEL clause
 SET TRANSACTION statement 755

isolation-clause 357
 in DELETE statement 614
 in INSERT statement 677
 in SELECT INTO statement 728
 in UPDATE statement 767

J

- JAVA clause
 - CREATE PROCEDURE (External) 506
 - DECLARE PROCEDURE (External) 605
 - in CREATE FUNCTION (External Scalar) 451
- Java Database Connectivity (JDBC) 4
- JOIN clause
 - in FROM clause 339
- JULIAN_DAY function 250

K

- KEEP LOCKS 357
- key
 - ALTER TABLE statement 384
 - composite 6
 - CREATE TABLE statement 546
 - foreign 7
 - parent 7
 - primary 6
 - primary index 6
 - unique 6
 - unique index 6

L

- LABEL statement 681, 683
- labeled duration 134
- LABELS
 - in catalog tables 681
 - in USING clause
 - DESCRIBE statement 619
 - DESCRIBE TABLE statement 624
 - PREPARE statement 692
- LAND function 251
- LANGID clause
 - in SET OPTION statement 742
- LANGUAGE clause
 - CREATE PROCEDURE (External) 505
 - in CREATE FUNCTION (External Scalar) 444
 - in CREATE FUNCTION (External Table) 460
 - in CREATE FUNCTION (SQL Scalar) 482
 - in CREATE FUNCTION (SQL Table) 491
 - in CREATE PROCEDURE (SQL) 516
 - in DECLARE PROCEDURE statement 602
- large integers 61
- large object (LOB)
 - data type 65
 - description 65
 - file reference variable 118
 - locator 66
 - locator variable 117
- LAST clause
 - in FETCH statement 646
- LCASE function 252
- LEFT EXCEPTION JOIN clause
 - in FROM clause 339

- LEFT function 253
- LEFT JOIN clause
 - in FROM clause 339
- LEFT OUTER JOIN clause
 - in FROM clause 339
- LENGTH function 255
- LIKE clause
 - in CREATE TABLE statement 543
 - in DECLARE GLOBAL TEMPORARY TABLE statement 592
- LIKE predicate 155
- limits
 - database manager 823
 - DataLink 823
 - datetime 823
 - identifier 51, 52, 821
 - in SQL 821
 - numeric 821
 - string 822
- literals 99
- LN function 256
- LNOT function 257
- LOB
 - data type 65
 - description 65
 - file reference variable 118
 - locator 66
 - locator variable 117
- LOCAL CHECK OPTION clause
 - CREATE VIEW statement 572
- LOCATE function 258
- locator
 - declaring host variable 117
 - description 66
 - FREE LOCATOR statement 651
 - HOLD LOCATOR statement 669
- LOCK TABLE statement 684, 685
- locking
 - COMMIT statement 413
 - during UPDATE 769
 - LOCK TABLE statement 684
 - table spaces 684
- locks
 - exclusive 22
 - share 22
- LOG function 259
- LOG10 function 259
- logical operator 161
- LONG VARCHAR
 - data type for CREATE TABLE 552
- LONG VARGRAPHIC
 - data type for CREATE TABLE 552
- LOR function 260
- LOWER function 261
- LTRIM function 262

M

- MAX
 - column function 173
 - scalar function 263
- MAXVALUE clause
 - in ALTER TABLE statement 380, 383
- MESSAGE_LENGTH
 - GET DIAGNOSTICS statement 800
- MESSAGE_OCTET_LENGTH
 - GET DIAGNOSTICS statement 800

- MESSAGE_TEXT
 - GET DIAGNOSTICS statement 800
- MICROSECOND function 264
- MIDNIGHT_SECONDS function 265
- MIN
 - column function 174
 - scalar function 266
- MINUTE function 267
- MINVALUE clause
 - in ALTER TABLE statement 380, 383
- mixed data
 - description 63
 - in LIKE predicates 156
 - in string assignments 84
- MOD function 268
- MODE
 - IN EXCLUSIVE MODE clause
 - LOCK TABLE statement 684
 - IN SHARE MODE clause
 - LOCK TABLE statement 684
- MODIFIES SQL DATA clause
 - CREATE PROCEDURE (External) 508
 - in CREATE FUNCTION (External Scalar) 445
 - in CREATE FUNCTION (External Table) 461
 - in CREATE FUNCTION (SQL Scalar) 483
 - in CREATE FUNCTION (SQL Table) 492
 - in CREATE PROCEDURE (SQL) 517
 - in DECLARE PROCEDURE 603
- MONTH function 270
- multiplication operator 129

N

- name
 - exposed 336
 - for SQL statements 607
 - in INCLUDE statement 671
 - subselect 331
- name qualification 53
 - default schema 54
- NAMES
 - in USING clause
 - DESCRIBE statement 619
 - DESCRIBE TABLE statement 624
 - PREPARE statement 692
- NAMING clause
 - in SET OPTION statement 743
- naming conventions in SQL 45
- NC (no commit) 24
- nested programs 777
- nested table expression 335
- NEXT clause
 - in FETCH statement 646
- NO ACTION delete rule
 - in ALTER TABLE statement 386
 - in CREATE TABLE statement 548
- NO ACTION update rule
 - in ALTER TABLE statement 387
 - in CREATE TABLE statement 548
- NO CACHE clause
 - in ALTER TABLE statement 380, 383
- no commit 24

NO COMMIT clause
 SET TRANSACTION statement 755

NO CYCLE clause
 in ALTER TABLE statement 381, 383

NO ORDER clause
 in ALTER TABLE statement 381, 383

NO SQL clause
 CREATE PROCEDURE
 (External) 508
 in CREATE FUNCTION (External
 Scalar) 445
 in CREATE FUNCTION (External
 Table) 461
 in DECLARE PROCEDURE 603

nodegroup
 definition 6
 in CREATE TABLE statement 549

nodegroup-name 48

NODENAME function 271

NODENUMBER function 272

NONE clause
 SET RESULT SETS statement 751

nonexecutable statement 365, 366

nonrepeatable read 25

NOT FOUND clause
 WHENEVER statement 776

NOT LOGGED clause
 in DECLARE GLOBAL TEMPORARY
 TABLE statement 595

NOT NULL clause
 ALTER TABLE statement 381
 CREATE TABLE statement 542
 in DECLARE GLOBAL TEMPORARY
 TABLE statement 588

NOW function 273

NUL-terminated string variables
 allowed 62

NULL
 in CAST specification 145
 in SET transition-variable
 statement 759
 in SET variable statement 761
 in UPDATE statement 766
 in VALUES INTO statement 773
 in VALUES statement 771
 keyword SET NULL delete rule
 description 8
 in ALTER TABLE statement 386
 in CREATE TABLE statement 548
 keyword SET NULL update rule
 in ALTER TABLE statement 387

NULL clause
 ALTER TABLE statement 378
 in CALL statement 398
 in INSERT statement 676

NULL predicate 160

null value in SQL
 assignment 81
 defined 61
 in grouping expressions 341
 in result columns 333
 specified by indicator variable 115

null value, SQL
 assigned to host variable 728

NULLIF function 274

numbers 61

numbers (*continued*)
 default decimal separator
 character 62

numeric
 assignments 82
 comparisons 89
 data type 61
 data types
 default decimal separator
 character 62
 string representation 62
 limits 821

NUMERIC
 data type for CREATE FUNCTION
 (External Scalar) 442
 data type for CREATE FUNCTION
 (External Table) 458
 data type for CREATE FUNCTION
 (Sourced) 472
 data type for CREATE FUNCTION
 (SQL Scalar) 481
 data type for CREATE FUNCTION
 (SQL Table) 490
 data type for CREATE PROCEDURE
 (External) 504
 data type for CREATE PROCEDURE
 (SQL) 516
 data type for CREATE TABLE 533
 data type for DECLARE
 PROCEDURE 601

O

object table 110

ON clause
 CREATE INDEX statement 497

ON COMMIT clause
 in DECLARE GLOBAL TEMPORARY
 TABLE statement 595

ON DISTINCT TYPE clause
 REVOKE (Distinct Type Privileges)
 statement 707

ON PACKAGE clause
 GRANT (package privileges)
 statement 663
 REVOKE (package privileges)
 statement 715

ON ROLLBACK clause
 in DECLARE GLOBAL TEMPORARY
 TABLE statement 596

ON TABLE clause
 GRANT (table privileges)
 statement 666
 REVOKE (table privileges)
 statement 718

ON TYPE clause
 GRANT (Distinct Type Privileges)
 statement 653

open state of cursor 649

OPEN statement 686, 690

operand
 date and time 134
 decimal 130
 distinct type 131
 floating point 131
 integer 130
 numeric 130

operation
 assignment 80, 83, 85, 86
 comparison 89, 92
 description 80

operators 129
 arithmetic 129

OPTIMIZE clause 356

OPTLOB clause
 in SET OPTION statement 743

OR
 truth table 161

ORDER BY clause
 of select-statement 351

ORDER clause
 in ALTER TABLE statement 381, 383

order of evaluation 138

ordinary identifier
 in SQL 43
 in system names 43

OUT clause
 CREATE PROCEDURE
 (External) 504
 DECLARE PROCEDURE
 statement 601
 in CREATE PROCEDURE (SQL) 516

outer join
See also LEFT OUTER JOIN clause
See RIGHT OUTER JOIN clause

OUTPUT clause
 in SET OPTION statement 743

OVRDBF (Override with Data Base
 file) 55

ownership 36

P

package
 description 13
 dropping 635
 in DRDA 26

PACKAGE clause 404
 COMMENT statement 404
 DROP statement 635
 LABEL statement 682

package view
 SYSPACKAGE 911

package-name 48
 in DROP statement 635
 in LABEL statement 682
 in REVOKE (package privileges)
 statement 715

PARAMETER clause
 COMMENT statement 410

parameter marker
 in EXECUTE statement 640
 in OPEN statement 687
 in PREPARE statement 694
 replacement 641, 688
 rules 694
 typed 694
 untyped 694
 usage in expressions, predicates and
 functions 694

parameter-marker
 in CAST specification 145

- parameter-name
 - CREATE PROCEDURE (External) 504
 - description 49
 - in CREATE PROCEDURE (SQL) 516
 - in DECLARE PROCEDURE 601
- PARAMETERS view 984
- parent key 7
- parent row 7
- parent table 7
- parentheses
 - with UNION 346
- PARTITION function 275
- partitioning key
 - definition 6
 - in CREATE TABLE statement 549
- password
 - in CONNECT (Type 1) statement 417
 - in CONNECT (Type 2) statement 422
- path
 - function resolution 125
- phantom row 25
- PI function 276
- PL/I
 - application program
 - varying-length string variables 62
 - host structure arrays 122
 - host variable 114, 120
 - SQLCA (SQL communication area) 832
 - SQLDA (SQL descriptor area) 861
- POSITION function 277
- POSTR function 277
- POWER function 279
- precedence
 - level 138
 - operation 138
- precision of a number 61
- predicate
 - basic 148
 - BETWEEN 151
 - description 147
 - EXISTS 152
 - IN 153
 - LIKE 155
 - NULL 160
 - quantified 149
- prefix operator 129
- PREPARE statement 691, 699
- prepared SQL statement
 - dynamically prepared by
 - PREPARE 691, 698
 - executing 640, 642
 - identifying by DECLARE 607
 - obtaining information
 - by INTO with PREPARE 620, 625
 - with DESCRIBE 618
 - with DESCRIBE TABLE 623
 - with SQLDA 847
 - SQLDA provides information 847
 - statements allowed 836
- primary index 6
- primary key 6
- PRIMARY KEY clause
 - ALTER TABLE statement 381, 384
 - CREATE TABLE statement 542, 546

- PRIOR clause
 - in FETCH statement 646
- privileges
 - description 36
- procedure
 - choosing parameter data types 500
 - commenting 411
 - creating 500, 501, 512
 - defining 598
 - dropping 636
 - granting 660
 - locators 500
 - RELEASE statement 700
 - revoking 713
 - ROLLBACK 720
 - signature 500
 - specific name 500
- PROCEDURE clause 404
- COMMENT statement 404
- DROP statement 635
- procedure-name
 - CREATE PROCEDURE (External) 504
 - description 49
 - in CALL statement 397
 - in CREATE PROCEDURE (SQL) 516
 - in DECLARE PROCEDURE 601
 - in DROP statement 635

- procedures 14
- SET CONNECTION statement 730
- PUBLIC clause
 - GRANT (table privileges) statement 666
 - in GRANT (Distinct Type Privileges) statement 653
 - in GRANT (function or procedure privileges) statement 660
 - in GRANT (package privileges) statement 663
 - in REVOKE (table privileges) statement 718
 - REVOKE (Distinct Type Privileges) statement 707
 - REVOKE (function or procedure privileges) statement 713
 - REVOKE (package privileges) statement 716

Q

- qualification of column names 108
- quantified predicate 149
- QUARTER function 280
- query 329, 358
- question mark (?)
 - See parameter marker

R

- RADIANS function 281
- RAND function 282
- RDBCNNMTH clause
 - in SET OPTION statement 743
- READ COMMITTED clause
 - SET TRANSACTION statement 755
- read stability 22

- READ UNCOMMITTED clause
 - SET TRANSACTION statement 755
- read-only-clause 355
- READS SQL DATA clause
 - CREATE PROCEDURE (External) 508
 - in CREATE FUNCTION (External Scalar) 445
 - in CREATE FUNCTION (External Table) 461
 - in CREATE FUNCTION (SQL Scalar) 483
 - in CREATE FUNCTION (SQL Table) 492
 - in CREATE PROCEDURE (SQL) 517
 - in DECLARE PROCEDURE 603
- REAL
 - data type for CREATE FUNCTION (External Scalar) 442
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 533
 - data type for DECLARE PROCEDURE 601
- REAL function 283
- recovery 15
- REFERENCES clause
 - ALTER TABLE statement 382, 385
 - CREATE TABLE statement 543, 547
 - GRANT (table privileges) statement 665
 - REVOKE (table privileges) statement 718
- referential constraint 7
- referential cycle 7
- referential integrity 7
 - delete rules 615
 - update rules 768
- REFERENTIAL_CONSTRAINTS view 988
- referential-constraint clause
 - of ALTER TABLE statement 385
 - of CREATE TABLE statement 547
- related information 1011
- relational database 1
- RELATIVE clause
 - in FETCH statement 577, 646
- RELEASE SAVEPOINT statement 702
- RELEASE statement 700, 701
- release-pending connection state 30
- remote unit of work 27
 - mixed environment 836
- RENAME statement 703, 705
- renaming SQL objects 703
- repeatable read 22
- REPEATABLE READ clause
 - SET TRANSACTION statement 755

reserved words 43, 1009
 RESET clause
 CONNECT (Type 1) statement 417
 CONNECT (Type 2) statement 422
 RESTART clause
 in ALTER TABLE statement 383
 RESTRICT clause
 DROP statement 633, 637
 in DROP COLUMN of ALTER TABLE statement 384
 in DROP constraint of ALTER TABLE statement 388
 RESTRICT delete rule
 description 8
 in ALTER TABLE statement 386
 in CREATE TABLE statement 548
 RESTRICT update rule
 in ALTER TABLE statement 387
 in CREATE TABLE statement 548
 result columns of subselect 333
 RESULT SETS clause
 CREATE PROCEDURE (External) 507
 in CREATE PROCEDURE (SQL) 516
 in DECLARE PROCEDURE 601
 result table 5
 temporary 577
 result-expression
 in CASE specification 140
 RETURN clause 578
 RETURN_STATUS
 GET DIAGNOSTICS statement 799
 RETURNS clause
 in CREATE FUNCTION (External Scalar) 443
 in CREATE FUNCTION (External Table) 459
 in CREATE FUNCTION (SQL Scalar) 482
 in CREATE FUNCTION (SQL Table) 491
 RETURNS NULL ON NULL INPUT clause
 CREATE FUNCTION statement 446, 462, 483, 492
 REVOKE (Distinct Type Privileges) statement 706, 707
 REVOKE (function or procedure privileges) statement 708, 714
 REVOKE (package privileges) statement 715, 716
 REVOKE (table privileges) statement 717
 REXX
 host variable 114
 RIGHT EXCEPTION JOIN clause
 in FROM clause 339
 RIGHT JOIN clause
 in FROM clause 339
 RIGHT OUTER JOIN clause
 in FROM clause 339
 rollback
 definition 17, 18
 description 17, 18
 ROLLBACK
 effect on SET TRANSACTION 756
 ROLLBACK statement 720, 723

ROUND function 284
 ROUTINES view 989
 row
 deleting 612
 dependent 7
 descendent 7
 inserting 673
 parent 7
 self-referencing 7
 ROW clause
 in UPDATE statement 766
 Row ID
 assignment 87
 data type
 description 73
 ROW_COUNT
 GET DIAGNOSTICS statement 799
 row-storage-area
 in FETCH statement 649
 row-subselect
 in SET transition-variable statement 759
 in SET variable statement 761
 in UPDATE statement 767
 in VALUES INTO statement 773
 in VALUES statement 771
 ROWID
 data type for CREATE FUNCTION (External Scalar) 442
 data type for CREATE FUNCTION (External Table) 458
 data type for CREATE FUNCTION (Sourced) 472
 data type for CREATE FUNCTION (SQL Scalar) 481
 data type for CREATE FUNCTION (SQL Table) 490
 data type for CREATE PROCEDURE (External) 504
 data type for CREATE PROCEDURE (SQL) 516
 data type for CREATE TABLE 535
 DECLARE PROCEDURE statement 601
 ROWID function 286
 ROWS clause
 INSERT statement 677
 RPG
 application program
 host variable 120
 varying-length string variables not allowed 62
 host structure arrays 122
 host variable 114
 integers 61
 RPG/400
 SQLCA (SQL communication area) 832
 RR (repeatable read) 22
 RRN function 287
 RS (read stability) 22
 RTRIM function 288
 rules
 names in SQL 45
 system name generation 553
 table name generation 553
 run-time authorization ID 58

S
 savepoint
 RELEASE SAVEPOINT statement 702
 ROLLBACK statement 720
 SAVEPOINT statement 724
 SAVEPOINT LEVEL clause
 CREATE PROCEDURE (External) 509
 CREATE PROCEDURE (SQL) 518
 SAVEPOINT statement 724, 725
 savepoint-name
 in RELEASE SAVEPOINT statement 702
 in SAVEPOINT statement 724
 SBCS data 63
 scalar function 123
 See function
 scalar-subselect 330
 scale of data
 comparisons in SQL 89
 conversion of numbers in SQL 82
 determined by SQLLEN variable 850
 in results of arithmetic operations 130
 in SQL 61
 schema
 description 5
 dropping 636, 637
 SCHEMA clause
 DROP statement 636
 schema-name
 definition 49
 in CREATE SCHEMA statement 522
 in DROP statement 636
 SCHEMATA view 997
 SCROLL clause
 in DECLARE CURSOR statement 577
 search condition
 description 161
 in JOIN clause 338
 order of evaluation 161
 with DELETE 614
 with HAVING 343
 with UPDATE 767
 with WHERE 340
 search-condition
 in CASE specification 141
 in UPDATE statement 767
 searched-when-clause
 in CASE specification 140
 SECOND function 289
 SELECT clause
 as syntax component 331
 GRANT (table privileges) statement 665
 REVOKE (table privileges) statement 718
 SELECT INTO statement 727, 729
 select list
 application 332
 notation 331
 SELECT statement 726
 fullselect 346
 subselect 330

- select-statement
 - in DECLARE CURSOR statement 578
 - used in INSERT statement 676
- self-referencing row 7
- self-referencing table 7
- SERIALIZABLE clause
 - SET TRANSACTION statement 755
- server 25, 840
- server-name
 - description 49
 - in CONNECT (Type 1) statement 416
 - in CONNECT (Type 2) statement 421
 - in DISCONNECT statement 627
 - in RELEASE statement 700
 - in SET CONNECTION statement 730
- SET clause
 - UPDATE statement 765
- SET CONNECTION statement 730, 732
- SET DATA TYPE clause
 - ALTER TABLE statement 382
- SET DEFAULT delete rule
 - description 8
 - in ALTER TABLE statement 386
 - in CREATE TABLE statement 548
- SET DEFAULT update rule
 - in ALTER TABLE statement 387
- SET default-clause
 - ALTER TABLE statement 383
- SET GENERATED ALWAYS clause
 - ALTER TABLE statement 383
- SET GENERATED BY DEFAULT clause
 - ALTER TABLE statement 383
- SET NOT NULL clause
 - ALTER TABLE statement 383
- SET NULL delete rule
 - description 8
 - in ALTER TABLE statement 386
 - in CREATE TABLE statement 548
- SET NULL update rule
 - in ALTER TABLE statement 387
- SET OPTION statement 733, 746
- SET PATH statement 747
- SET RESULT SETS statement 750, 752
- SET SCHEMA statement 753
- SET TRANSACTION statement 755, 757
- SET transition-variable statement 758
- SET variable statement 760
- SHARE
 - IN SHARE MODE clause
 - LOCK TABLE statement 684
- share locks 22
- SHARE MODE clause
 - in LOCK TABLE statement 684
- shift-in character 85
 - not truncated by assignments 84
- SIGN function 290
- simple-when-clause
 - in CASE specification 140
- SIN function 291
- single row select 727
- single-byte character
 - in LIKE predicates 156
- single-precision floating-point 61
- SINH function 292
- small integers 61
- SMALLINT
 - data type for CREATE FUNCTION (External Scalar) 442
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 532
 - data type for DECLARE PROCEDURE 601
- SMALLINT data type 61
- SMALLINT function 293
- SOME quantified predicate 149
- sort sequence 35
- SOUNDEX function 294
- sourced
 - function 469
- SPACE function 295
- special register 105
 - CURRENT DATE 105
 - CURRENT PATH 105
 - CURRENT SCHEMA 106
 - CURRENT SERVER 106
 - CURRENT TIME 106
 - CURRENT TIMESTAMP 107
 - CURRENT TIMEZONE 107
 - CURRENT_DATE 105
 - CURRENT_PATH 105
 - CURRENT_SERVER 106
 - CURRENT_TIME 106
 - CURRENT_TIMESTAMP 107
 - CURRENT_TIMEZONE 107
 - in CALL statement 398, 399
 - USER 107
- SPECIFIC clause
 - COMMENT statement 410, 411
 - CREATE PROCEDURE (External) 507
 - DROP statement 635, 636
 - GRANT statement 659, 660
 - in CREATE FUNCTION (External Scalar) 444
 - in CREATE FUNCTION (External Table) 460
 - in CREATE FUNCTION (Sourced) 475
 - in CREATE FUNCTION (SQL Scalar) 482
 - in CREATE FUNCTION (SQL Table) 491
 - in CREATE PROCEDURE (SQL) 517
 - in DECLARE PROCEDURE 602
 - REVOKE statement 712, 713
- specific-name
 - description 49
 - in COMMENT statement 410, 411
 - in CREATE FUNCTION (Sourced) 475
 - in DROP statement 635, 636
- specific-name (*continued*)
 - in GRANT statement 659, 660
 - in REVOKE statement 712, 713
- SQL
 - function 478, 487
- SQL (structured query language)
 - dynamic SQL 3
 - extended dynamic SQL 4
 - static SQL 3
- SQL (Structured Query language)
 - interactive SQL facility 4
- SQL (Structured Query Language) 39, 425, 626, 628, 638, 665, 705, 770, 801
 - assignment operation 80
 - assignments and comparisons 80
 - binary large object (BLOB) 65
 - binary strings 65
 - bind 3
 - call level interface (CLI) 4
 - character large object (CLOB) 65
 - character strings 62
 - characters 39
 - comparison operation 80
 - constants 99
 - data types 59
 - dates and times 67
 - double-byte character large object (DBCLOB) 65
 - dynamic
 - statements allowed 836
 - Embedded SQL for Java (SQLJ) 4
 - escape character 43
 - identifiers 43
 - Java Database Connectivity (JDBC) 4
 - large object (LOB) 65
 - limits 821
 - naming conventions 45
 - null value 61
 - numbers 61
 - Open Database Connectivity (ODBC) 4
 - tokens 41
 - variable names used 45
- SQL clause
 - CREATE PROCEDURE (External) 505
 - DECLARE PROCEDURE (External) 604
 - in CREATE FUNCTION (External Scalar) 450
- SQL path 53
 - function resolution 125
 - SET PATH 747
 - SET SCHEMA 753
- SQL server mode
 - threads 20
- SQL statement
 - CREATE FUNCTION (external scalar) 454
 - CREATE FUNCTION (Sourced) 469
 - CREATE FUNCTION (SQL Table) 487
 - CREATE PROCEDURE (External) 501
 - CREATE PROCEDURE (SQL) 512
- SQL statements
 - ALTER TABLE 370, 393

SQL statements (*continued*)

- BEGIN DECLARE SECTION 394, 395
- CALL 396, 401
- characteristics 835
- CLOSE 402, 403
- COMMENT 404, 412
- COMMIT 413, 415
- CONNECT (Type 1) 416, 420
- CONNECT (Type 2) 421, 425
- CONNECT differences 845
- CREATE ALIAS 425, 427
- CREATE DISTINCT TYPE 428
- CREATE FUNCTION (External Scalar) 439
- CREATE FUNCTION (External Table) 455
- CREATE FUNCTION (SQL Scalar) 478
- CREATE INDEX 496
- CREATE PROCEDURE (SQL) 520
- CREATE SCHEMA 521, 525
- CREATE TABLE 525
- CREATE TRIGGER 556
- CREATE VIEW 569, 575
- data access indication 838
- DECLARE CURSOR 576, 582
- DECLARE GLOBAL TEMPORARY TABLE 583
- DECLARE GLOBAL TEMPORARY TABLE statement 597
- DECLARE PROCEDURE 598, 606
- DECLARE STATEMENT 607, 608
- DECLARE VARIABLE 609, 611
- DELETE 612, 617
- DESCRIBE 618, 622
- DESCRIBE TABLE 623, 626
- DISCONNECT 627, 628
- DROP 629, 638
- END DECLARE SECTION 639
- EXECUTE 640, 642
- EXECUTE IMMEDIATE 643, 644
- FETCH 645, 650
- FREE LOCATOR 651
- GET DIAGNOSTICS 799, 801
- GRANT (Distinct Type Privileges) 652, 654
- GRANT (function or procedure privileges) 655, 661
- GRANT (package privileges) 662, 664
- GRANT (table privileges) 664, 668
- HOLD LOCATOR 669, 670
- INCLUDE 671, 672
- INSERT 673, 680
- LABEL 681, 683
- LOCK TABLE 684, 685
- names for 607
- OPEN 686, 690
- PREPARE 691, 699
- prepared 3
- RELEASE 700, 701
- RELEASE SAVEPOINT 702
- RENAME 703, 705
- REVOKE (Distinct Type Privileges) 706, 707

SQL statements (*continued*)

- REVOKE (function or procedure privileges) 708, 714
- REVOKE (package privileges) 715, 716
- REVOKE (table privileges) 717
- ROLLBACK 720, 723
- SAVEPOINT 724, 725
- SELECT 726
- SELECT INTO 727, 729
- SET CONNECTION 730, 732
- SET OPTION 733, 746
- SET PATH 747
- SET RESULT SETS 750, 752
- SET SCHEMA 753
- SET TRANSACTION 755, 757
- SET transition-variable 758
- SET variable 760
- UPDATE 762, 770
- VALUES 771
- VALUES INTO 773
- WHENEVER 776, 779
- SQL_FEATURES view 998
- SQL_LANGUAGES table 999
- SQL_SIZING view 1000
- SQL-label
 - description 50
- SQL-parameter-name
 - description 50
- SQL-parametere-name
 - in GET DIAGNOSTICS statement 799
- SQL-variable-name
 - description 50
 - in GET DIAGNOSTICS statement 799
- SQLCA (SQL communication area)
 - C 831
 - COBOL 831
 - contents 825
 - description 825
 - entry changed by UPDATE 768
 - FORTRAN 831
 - ILE RPG/400 833
 - PL/I 832
 - RPG/400 832
- SQLCA (SQL communication area) clause
 - INCLUDE statement 671
- SQLCODE 368
- SQLCOLPRIVILEGES view 950
- SQLCOLUMNS view 951
- SQLCURRULE clause
 - in SET OPTION statement 743
- SQLD field of SQLDA 619, 624, 849
- SQLDA (SQL descriptor area)
 - C 857
 - COBOL 860
 - contents 847
 - ILE COBOL 860
 - ILE RPG/400 862
 - PL/I 861
- SQLDA (SQL descriptor area) clause
 - INCLUDE statement 671
- SQLDABC field of SQLDA 619, 624, 849
- SQLDAID field of SQLDA 618, 624, 849
- SQLDATA field of SQLDA 856
- SQLDATALEN field of SQLDA 853
- SQLERRMC field of SQLCA
 - values for CONNECT 830
 - values for SET CONNECTION 830
- SQLERROR clause
 - WHENEVER statement 776
- SQLFOREIGNKEYS view 956
- SQLIND field of SQLDA 850
- SQLLEN field of SQLDA 850, 854
- SQLLONGLEN field of SQLDA 853
- SQLN field of SQLDA 618, 623, 849
- SQLNAME field of SQLDA 850, 853, 856
- SQLPATH clause
 - in SET OPTION statement 744
- SQLPRIMARYKEYS view 957
- SQLPROCEDURECOLUMNS view 958
- SQLPROCEDURES view 962
- SQLSCHEMAS view 963
- SQLSPECIALCOLUMNS view 964
- SQLSTATE
 - description 368
- SQLSTATISTICS view 966
- SQLTABLEPRIVILEGES view 967
- SQLTABLES view 968
- SQLTYPE
 - unsupported 856
- SQLTYPE field of SQLDA 850, 854
- SQLTYPEINFO view 969
- SQLUDTS view 974
- SQLVAR field of SQLDA 619, 624, 852
- SQLWARNING clause
 - WHENEVER statement 776
- SQRT function 296
- SRTSEQ clause
 - in SET OPTION statement 744
- START WITH clause
 - in ALTER TABLE statement 379
- statement string 643
- statement-name
 - description 50
 - in DECLARE CURSOR statement 579
 - in DECLARE STATEMENT statement 607
 - in DESCRIBE statement 618
 - in EXECUTE statement 640
 - in PREPARE statement 692
- states
 - SQL connection 30
- static select 366
- static SQL 3, 365
 - use of SQL path 53
- STDDEV function 175
- STDDEV_POP function 175
- string
 - assignment 83
 - columns 62
 - constant
 - binary 102
 - character 99
 - graphic 100
 - hexadecimal 99, 102
 - limitations on use of 67
 - limits 822
 - variable
 - CLOB 62
 - DBCLOB 64

- string (*continued*)
 - variable (*continued*)
 - fixed-length 62
 - varying-length 62
- string delimiter 41, 99, 102
- string-expression
 - in EXECUTE IMMEDIATE statement 643
 - in PREPARE statement 693
- STRIP function 297
- subquery
 - description 111, 346
 - in HAVING clause 343
- subselect 330
 - in CREATE VIEW statement 330
- substitution character 33
- SUBSTR function 298
- SUBSTRING function 298
- subtraction operator 129
- SUM function 176
- synonym for qualifying a column name 108
- SYSCATALOGS view 885
- SYSCHKCST view 887
- SYSCOLUMNS view 888
- SYSCST view 897
- SYSCSTCOL view 898
- SYSCSTDEP view 899
- SYSFUNCS view 900
- SYSINDEXES view 906
- SYSJARCONTENTS view 907
- SYSJAROBJECTS view 908
- SYSKEYCST view 909
- SYSKEYS view 910
- SYSPACKAGE view 911
- SYSPARMS table 913
- SYSPROCS view 917
- SYSREFCST view 922
- SYSROUTINEDEP view 923
- SYSROUTINES table 924
- SYSTABLES view 932
- system column name 6, 12, 532, 570, 588, 619, 625
- system identifier 43
- SYSTEM NAME clause
 - RENAME statement 703
- system name generation
 - rules 553
- SYSTEM NAMES
 - in USING clause
 - DESCRIBE statement 619
 - DESCRIBE TABLE statement 624
 - PREPARE statement 692
- system path 747
- system table name 5
- system-column-name 553
 - description 50
 - in ALTER TABLE statement 376
 - in CREATE TABLE statement 532
 - in CREATE VIEW statement 570
 - in DECLARE GLOBAL TEMPORARY TABLE statement 588
- system-object-name
 - definition 50
- SYSTRIGCOL view 934
- SYSTRIGDEP view 935
- SYSTRIGGERS view 936

- SYSTRIGUPD view 940
- SYSTYPES table 941
- SYSVIEWDEP view 946
- SYSVIEWS view 948

T

- table
 - altering 370
 - creating 525
 - definition 5
 - dependent 7
 - descendent 7
 - designator 110, 287
 - distributed 6
 - dropping 636, 637
 - global temporary 583
 - parent 7
 - primary key 6
 - self-referencing 7
 - system table name 5
 - temporary 688
- TABLE clause
 - COMMENT statement 411
 - DROP statement 637
 - LABEL statement 682
 - RENAME statement 703
- table function 124
 - FROM clause
 - of subselect 336
- table name generation
 - rules 553
- TABLE_CONSTRAINTS view 1001
- table-name
 - description 50
 - in ALTER TABLE statement 376
 - in CREATE ALIAS statement 425
 - in CREATE INDEX statement 497
 - in CREATE TABLE statement 532, 547
 - in DECLARE GLOBAL TEMPORARY TABLE statement 587
 - in DELETE statement 613
 - in DROP statement 637
 - in GRANT (table privileges) statement 666
 - in INSERT statement 675
 - in LABEL statement 682
 - in LOCK TABLE statement 684
 - in REFERENCES clause of ALTER TABLE statement 385
 - in RENAME statement 703
 - in REVOKE (table privileges) statement 718
 - in UPDATE statement 765
- TABLES view 1002
- TAN function 301
- TANH function 302
- temporary
 - result table 577
- temporary tables in OPEN 688
- TEXT clause
 - LABEL statement 682
- TGTRLS clause
 - in SET OPTION statement 744
- thread safety 20

- time
 - arithmetic operations 137
 - duration 134
 - strings 69
- TIME
 - assignment 86
 - data type 68
 - data type for CREATE TABLE 535
 - function 303
- timestamp
 - arithmetic operations 138
 - duration 134
 - strings 72
- TIMESTAMP
 - assignment 86
 - data type 68
 - data type for CREATE TABLE 535
 - function 304
- TIMESTAMPDIFF
 - function 306
- TIMFMT clause
 - in SET OPTION statement 745
- TIMSEP clause
 - in SET OPTION statement 745
- tokens in SQL 41
- transition table 560
- transition variable 560
- TRANSLATE function 307
- trigger 9
 - creating 556
 - delete rules 615
 - dropping 637
 - RELEASE statement 700
 - ROLLBACK 720
 - SET CONNECTION statement 730
 - setting isolation level 756
 - update rules 768
- TRIGGER clause
 - COMMENT statement 404, 411
 - DROP statement 637
- trigger-name
 - description 50
 - in DROP statement 637
- TRIM function 309
- TRUNCATE function 311
- truncation of numbers 82
- truth table 161
- truth valued logic 161
- type
 - dropping 633
- TYPE clause
 - DROP statement 633

U

- UCASE function 313
- UCS-2 (universal coded character set)
 - description 65
- UCS-2 graphic constant
 - hexadecimal 101
- UDF (user-defined function) 123
 - external 123
 - sourced 123
 - SQL 123
- unary
 - minus 129
 - plus 129

- uncommitted read 23
- unconnected state 30
- undefined reference 111
- UNION ALL clause
 - of fullselect 346
- UNION clause
 - of fullselect 346
 - with duplicate rows 346
- UNIQUE clause
 - ALTER TABLE statement 382, 384
 - CREATE INDEX statement 497
 - CREATE TABLE statement 543, 546
 - in SAVEPOINT statement 724
- unique index 6
 - update rules 768
- unique key 6
- unit of work
 - COMMIT 413
 - ending
 - closes cursors 688
 - COMMIT 413
 - referring to prepared statements 691
 - ROLLBACK 720
- UPDATE
 - in ON UPDATE clause of ALTER TABLE statement 387
 - in ON UPDATE clause of CREATE TABLE statement 548
- UPDATE clause 354
 - GRANT (table privileges) statement 665
 - REVOKE (table privileges) statement 718
- update rules 768
 - check constraint 768
 - checking of unique constraints 768
 - effect of commitment control 768
 - referential integrity 768
 - trigger 768
 - views with WITH CHECK OPTION 768
- UPDATE statement 762, 770
- UPPER function 314
- UR (uncommitted read) 23
- USAGE clause
 - GRANT (Distinct Type Privileges) statement 653
 - REVOKE (Distinct Type Privileges) statement 706
- USER clause
 - ALTER TABLE statement 377, 378
 - CONNECT (Type 1) statement 417
 - CONNECT (Type 2) statement 422
 - CREATE TABLE statement 538
 - DECLARE GLOBAL TEMPORARY TABLE statement 590
- USER special register 107
- USER_DEFINED_TYPES view 1003
- user-defined function 123
 - external 123
 - sourced 123
 - SQL 123
- user-defined types (UDTs)
 - data types
 - description 73
- USING clause
 - CONNECT (Type 1) statement 417

- USING clause (*continued*)
 - CONNECT (Type 2) statement 422
 - DESCRIBE statement 619
 - DESCRIBE TABLE statement 624
 - EXECUTE statement 640
 - in CREATE TABLE statement 545
 - in DECLARE GLOBAL TEMPORARY TABLE statement 594
 - OPEN statement 686
 - PREPARE statement 692
- USING DESCRIPTOR clause
 - CALL statement 399
 - EXECUTE statement 640
 - OPEN statement 687
- USING HASHING
 - in CREATE TABLE statement 550
- USRPRF clause
 - in SET OPTION statement 746

V

- VALUE function 315
- VALUES clause
 - INSERT statement 676, 677
- VALUES INTO statement 773
- VALUES statement 771
- VAR function 177
- VAR_POP function 177
- VARCHAR
 - data type for CREATE FUNCTION (External Scalar) 442
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 533
 - data type for DECLARE PROCEDURE 601
 - function 316
- VARGRAPHIC
 - data type for CREATE FUNCTION (External Scalar) 442
 - data type for CREATE FUNCTION (External Table) 458
 - data type for CREATE FUNCTION (Sourced) 472
 - data type for CREATE FUNCTION (SQL Scalar) 481
 - data type for CREATE FUNCTION (SQL Table) 490
 - data type for CREATE PROCEDURE (External) 504
 - data type for CREATE PROCEDURE (SQL) 516
 - data type for CREATE TABLE 534
 - data type for DECLARE PROCEDURE 601
 - function 320

- variable
 - file reference 117, 118
- VARIANCE function 177
- view
 - catalog 881
 - creating 569
 - deletable 573
 - dropping 637
 - insertable 573
 - read-only 573
 - updatable 573
 - updating with WITH CHECK OPTION views 768
- VIEW clause
 - CREATE VIEW statement 569
 - DROP statement 637
- view-name
 - description 51
 - in CREATE ALIAS statement 425
 - in CREATE VIEW statement 570
 - in DELETE statement 613
 - in DROP statement 637
 - in GRANT (table privileges) statement 666
 - in INSERT statement 675
 - in LABEL statement 682
 - in RENAME statement 703
 - in REVOKE (table privileges) statement 718
 - in UPDATE statement 765
- VIEWS view 1007

W

- WEEK function 323
- WEEK_ISO function 324
- WHENEVER statement 776, 779
- WHERE clause
 - DELETE statement 614
 - of subselect 340
 - UPDATE statement 767
- WHERE CURRENT OF clause
 - DELETE statement 614
 - UPDATE statement 767
- WHERE NOT NULL clause
 - in CREATE INDEX statement 497
- WITH CASCADED CHECK OPTION clause
 - CREATE VIEW statement 571
- WITH CHECK OPTION clause
 - CREATE VIEW statement 571
 - effect on update 768
- WITH CHECK OPTION clause of CREATE VIEW statement
 - UPDATE rules 768
- WITH clause 357
 - UPDATE statement 614, 677, 767
- WITH COMPARISONS
 - CREATE DISTINCT TYPE statement 430
- WITH DATA DICTIONARY clause
 - CREATE SCHEMA statement 522
- WITH DEFAULT clause
 - CREATE TABLE statement 536
 - in DECLARE GLOBAL TEMPORARY TABLE statement 588

- WITH DISTINCT VALUES clause
 - CREATE INDEX statement 498
- WITH GRANT OPTION clause
 - in GRANT (Distinct Type Privileges) statement 653
 - in GRANT (function or procedure privileges) statement 660
 - in GRANT (package privileges) statement 663
 - in GRANT (table privileges) statement 666
- WITH HOLD clause
 - in DECLARE CURSOR statement 578
- WITH LOCAL CHECK OPTION clause
 - CREATE VIEW statement 572
- WITH REPLACE clause
 - in DECLARE GLOBAL TEMPORARY TABLE statement 595
- WITH RETURN clause
 - in DECLARE CURSOR statement 578
- words
 - reserved 43, 1009
- WORK clause
 - in COMMIT statement 413
 - ROLLBACK statement 721

X

- XOR function 325

Y

- YEAR function 326

Z

- ZONED function 327



Printed in U.S.A.